OLIVEIRA, F.S. 2014. Reinforcement learning for business modeling. In Wang, J. (ed.) *Encyclopedia of business analytics and optimization*. Hershey: IGI Global [online], chapter 181, pages 2010-2019. Available from: https://doi.org/10.4018/978-1-4666-5202-6.ch181

Reinforcement learning for business modeling.

OLIVEIRA, F.S.

2014

Reinforcement Learning for Business Modeling

INTRODUCTION

Inter-temporal decision making is an important but difficult class of problems that analyze the interdependence between sequences of decisions in time. These tend to be very complex problems with very important applications to real world problems such as, for instance, product differentiation (e.g., Hsu & Wang, 2004), portfolio optimization (e.g., Bae et al., 2013; Kraft & Steffensen, 2013), inventory management (e.g., Murphy and Oliveira, 2010), futures and options valuation (e.g., Gulpinar and Oliveira, 2012), among many others. In all the models listed so far, if there is uncertainty in the parameters characterized by known distributions models which are solved using different variations of stochastic dynamic programming.

However, there are planning problems in which the information about the uncertain parameters is very limited and either the transition probabilities between states are unknown or the rewards received in each state are difficult to estimate, or both. Such issues with incomplete information can arise due to lack of information about consumer behavior, lack of knowledge about the competitors' behavior, or due to uncertainty associated with regulatory policy, for example.

One methodology that is able to derive dynamic optimal policies in the context of limited information about transition probabilities or rewards is reinforcement learning (Sutton, 1988; Weiss, 1995; Bertsekas and Tsitsiklis, 1996). By using an algorithm of reinforcement learning, an agent plays a game in the extensive form with incomplete information where, at the end of each stage, the player only observes the outcome of the game (the transition among states) and the reward he receives. Applications of reinforcement learning algorithms include, among many others, scheduling problems (Zhang et al., 2011), and Raju et al. (2006).

The reinforcement learning algorithms can approximate the best policy to play within a certain environment without building an explicit model of it. In the first type of model presented (the *n*-armed bandit), an agent decides how to play in the next iteration given the expected utility (profit) of each possible action. Other reinforcement learning algorithms, such as temporal differences or Q-learning, are more complex. The agent evaluates each action (in the possible states of the world) taking into account its possible repercussions on the future behavior of the system.

This chapter summarizes the reinforcement learning theory emphasizing its relationship with dynamic programming (reinforcement learning algorithms may replace dynamic programming when a full model of the environment is *not* available) and analyzing its limitations as a tool for modeling human and organizational behavior.

THE REINFORCEMENT LEARNING PROBLEM

Before proceeding, this section presents the terminology used in models of learning, which follows Singh (1994). The first concept presented is the *Markov property* (Howard, 1971). Let $P\left(s_{t+1}=s' \mid s_t, s_{t-1}, ..., s_0\right),\ \forall s', s_t, s_{t-1}, ..., s_0$ represent the transition probability from state s_t to a state s', where s_t is the state of a dynamic system at time *t*. If the state has the Markov Property, it contains all the information to determine the transition probabilities and, in this case, the model transition dynamics is $P\left(s_{t+1}=s' \mid s_t\right)$. If every state has the Markov Property, the environment and the task as a whole also have the Markov Property. A Markovian decision process is characterized by a tuple (*policy*, *reward function*, *value function*, *model of the environment*).

A *policy* $\pi : A(s) \rightarrow a$ is a rule of behavior that transforms states into actions. A *reward function* $u_{ij}^{a_t}$ defines the utility that an agent receives from choosing an action a_t in a certain state

i, at a given time t. The ultimate goal of an agent is to maximize the total reward received in the long-run, his *Return* (R_t). If the horizon is finite with T stages, then $R_t = u^{a_t}_{s_t,s_{t+1}} + u^{a_{t+1}}_{s_{t+1},s_{t+2}} \ldots + u^{a_T}_{s_T,s_{T+1}}$ If the time horizon is infinite, the return is the discounted sum (where $0 \leq \gamma \leq 1$ is the *discount* parameter) of each one of the rewards $R_t = \sum_{k=0}^{+\infty} \gamma^k u_{t+k+1}$.

A *value function*, *V(s)*, specifies the value of the state *s*, i.e., the total value of rewards an agent expects to receive until the end of the horizon by entering that state immediately. When the problem has discrete state spaces and actions, and if the agent has a perfect knowledge of the transition probabilities and rewards, in order to compute the optimal policy there is no need to interact with the environment. The method used to compute the optimal policy is dynamic programming (e.g., Bertsekas, 2000).

A *model* of the environment is the set of state transition probabilities and transition rewards associated with every action in each state. This model is just optional in reinforcement learning but it is essential for dynamic programming. One of the advantages of reinforcement learning is that it enables an agent to have forward-looking behaviour without having an explicit or complete model of the environment. However, an agent is required to interact a very large number of times with the environment.

An algorithm that works *on-line* learns the value function and controls a real environment at the same time. On-line algorithms present a trade-off between *exploration* (an attempt to improve the knowledge possessed at a certain time) and *exploitation* (an attempt to profit from the knowledge of the environment). In order to increase its knowledge of the environment an agent may have to choose sub-optimal actions. If an agent chooses only optimal actions, he may end up exploiting a local optimum, not converging to the *global optimum* solution. *Off-line* algorithms use simulated experience with a model of the environment and do not face the

exploitation vs. exploration trade-off. These algorithms learn the control policy before applying it to the real world. Note that the distinction between on-line and off-line algorithms, and real versus simulated world, is somewhat misleading. An algorithm that does not require a model of the environment can solve a problem in which a model is available. In this case, a player uses the model to simulate the real world.

Next, dynamic programming and three main approaches to reinforcement learning are discussed, namely the *n*-armed bandit, the temporal differences and the Q-learning algorithms. As seen in the previous presentation, reinforcement learning and dynamic programming can be used to solve the same type of problems, as both aim to computing the *optimal policies* given that the environment follows a Markov process. Therefore, this chapter describes reinforcement learning as a generic problem of which dynamic programming is a special case: this approach follows Sutton and Barto (1998), Bertsekas and Tsitsiklis (1996) and Kaelbling et al. (1996).

DYNAMIC PROGRAMMING

Bellman (1957) defined dynamic programming as the mathematical theory of multi-stage decision processes, and defined *decision process* as a system (possibly stochastic) in which the decision-maker has a choice of transformations that may be applied to the system at any time. He also distinguishes single-stage (only one decision has to be made) from multi-stage decision processes (where the decision-maker has to take a sequence of decisions). A decision maker solves a dynamic programming problem when he understands the *structure* of the optimal policy. This means that the decision-maker understands the system characteristics, which determine the decisions at any particular stage of the process. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal

policy with regard to the state resulting from the first decision (this is the *Principle of Optimality*).

Therefore, the dynamic programming problem is the one of computing the *optimal policies*, given that the environment follows a Markovian process and a known model of the environment. Consequently, it implies the solution of two related problems: prediction (policy evaluation) and control.

The *prediction* problem consists of estimating the value of each state of the environment for a certain policy π. Let E_π represent the expected value of a policy π, and let $\pi(s,a)$ represents the probability of executing action *a* in state *s*, following a policy π. Further, let $P^a_{ss'}$ stand for the probability of the system moving from state s to state s', conditional on the agent choosing action *a*. Then the *state-value function* $V^\pi(s)$ represents the expected return of state *s*, following a policy π such that $V^\pi(s) = E_\pi\{R_t \setminus s_t = s\}$, or equivalently $V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left(u^a_{ss'} + \gamma V^\pi(s')\right)$.

The control problem takes into account that a given policy influences the value of a given action. Thus, let the *action-value function* $Q^\pi(s,a)$ represent the expected return of an action *a,* in state *s,* following policy π. Then, $Q^\pi(s,a) = E_\pi\{R_t \setminus s_t = s, a_t = a\}$ represents the action-value function, which is equivalent to $Q^\pi(s,a) = \sum_{s'} P^a_{ss'} \left(u^a_{ss'} + \gamma V^\pi(s')\right)$.

It is then intuitive that a value function imposes a partial order on the space of policies. A policy π is better than or equal to a policy π' if and only if its expected value is higher than or equal to the expected return of policy π'. More formally, $\pi \succ \pi' \Leftrightarrow V^\pi(s) \geq V^{\pi'}(s)$, for all $s \in S$.

As it is possible to have a partial ordering on the space of policies, it is also possible to define optimal value functions. The optimal state-value function $V^*(s) = \max_\pi V^\pi(s), \forall s \in S$ represents the maximum expected value of a state *s*. Therefore, the optimal action-value function $Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in S, a \in A(s)$ represents the maximum expected value of an action *a* in state *s*. Consequently, an optimal policy π^* gives the association between states of the world and actions that maximise the expected return, $Q^*(s,a): \ \pi^*(s) = \arg\max_a \sum_{s'} P^a_{ss'} \left\{ u^a_{ss'} + \gamma V^\pi(s') \right\}$, $\forall s \in S$.

In the infinite horizon the value iteration and the policy iteration algorithms, first developed by Howard (1960), can approximate the optimal policy and the optimal value functions. The *value iteration* algorithm is a successive approximation method, in the space of the value functions *V(s)*, that converges to the optimal value function, in the limit. The *policy iteration* algorithm is an approximation method, in the space of policies, which converges to the optimal policy, in the limit (Ross, 1983; Bertsekas, 2000).

N-ARMED BANDIT

In the *n*-armed bandit problem (see Sutton and Barto, 1998) a player needs to choose between *n* possible actions. After each action, the player receives a reward. A player's objective is to choose the actions that maximize the overall reward in a certain period. Usually, this is a stochastic problem (the rewards are generated by a stationary stochastic process) where each action has a certain expected value (the expected value of the rewards received) unknown to the player. The player's task is to find the action with the highest expected reward and to execute it as often as possible. In order to learn the expected reward associated with each action, an agent is

required to try non-optimal actions during a few iterations (exploration) instead of repeatedly choosing the action with the highest expected value at a certain time (exploitation).

Let $Q_t(a)$ stand for the expected reward of action *a*, at stage *t*, and let *a* represent one of the possible *n* actions available to the agent. Further, let u_{t+1}^a stand for the reward received by executing action *a* at iteration *t+1*. An agent attempts to estimate the true value of *a*, *Q*(*a*), computing on-line equation (1).

$$Q_{t+1}(a) = Q_t(a) + \alpha.\left[u_{t+1}^a - Q_t(a)\right],\ \forall a\,. \tag{1}$$

Equation (1) represents an exponential smoothing of past rewards with a weight-factor α (also known as learning rate) such that $0 \le \alpha \le 1$. The learning algorithm converges with probability one for the true action value only if the learning parameter decays over time, let us say $\alpha_t = \frac{1}{t}$. However, only in problems with stationary rewards is this condition valid. The conditions for convergence with probability one are presented in equation (2), i.e., the learning rate converges to zero but slowly, meaning that the agent can learn fast enough to revise his initial expectations and slowly enough not to incorporate the effects of random events into his expectations.

$$\sum_{t=1}^{+\infty} \alpha_t(a) = +\infty \text{ and } \sum_{t=1}^{+\infty} \alpha_t^2(a) < +\infty\,. \tag{2}$$

Arthur (1991) using this type of algorithm to capture bounded rationality has found that computer algorithms (after calibration) learned in a very similar way to humans, in the sense that they also deviate from perfect-rationality. Roth and Erev (1995) used an *n*-armed bandit algorithm to capture the effect of experience, and learning, in human behavior.

The *n*-armed bandit algorithm has also been used to model electricity markets. Nicolaisen et al. (2001), Bunn and Oliveira (2001, 2003) have simulated a wholesale electricity market

using agent-based computational learning. These models aimed to analyzing market power issues in electricity markets by simulating the interaction between different generation companies, and by capturing the players' learning behavior. Next we present the problems of forecasting and controlling in reinforcement learning theory.

TEMPORAL-DIFFERENCES ALGORITHM

Sutton (1988) presents the temporal-difference learning algorithm as a new method for forecasting. The temporal differences algorithm started, back in the 50s, with Samuel's (1959) checkers player and, so far, it was in the games research area that the temporal-differences algorithm achieved its highest success: a Gammon player (Tesauro, 1994) and a Chess player (Baxter et al., 2000). The temporal-difference learning can also be applied to solve control problems (Tesauro, 1992). Sutton (1988) has defended that whereas conventional forecasting methods use the difference between predicted and actual outcomes as their learning signal the temporal-difference algorithm looks at the difference between successive predictions: with the temporal-difference algorithms learning occurs whenever there is a change in predictions. In the multi-step prediction problems (where a forecast of an outcome at time $t+lag$ is made at time t), the temporal difference methods learn to correct the predictions (for some *lag*), at time t, taking into account the forecasting errors at time *lag-k*, for every $k \in [0, lag-t-1]$. In addition, temporal-difference algorithms can deal with the *credit assignment problem* (to identify, in the set of actions that a player played before a certain outcome, the action which is the responsible for that outcome).

Let $V(s_t)$ stand for the value of state s_t, u_{t+1} represent the reward of received after executing action a_t, at state s_t (that resulted in the transition to state s_{t+1}), and Γ denote the

trajectory s_0, s_1, … , s_n. At the end of the episode, the expected value of each state s_k in the trajectory Γ is updated by the rule $V(s_k) := V(s_k) + \alpha\left[u_{t+1} + \gamma V(s_{k+1}) - V(s_k)\right]$, in which *k*, *k+1*, … represents the index of each state in Γ, $0 \leq \gamma \leq 1$ and $0 \leq \alpha \leq 1$ denote, respectively, the *discount* parameter and the learning rate, and := is an operator used to represent an iterative process. This means that the value of a certain variable is numerically approximated by iterations of the algorithm on the previous value of the same variable.

The temporal-difference algorithm is a *bootstrapping method* as it updates the values of the weights based on previous existing predictions. Equation $d_k \equiv u_{k+1} + \gamma V(s_{k+1}) - V(s_k)$ defines the temporal differences, which represent the difference between the new $\left[u_{k+1} + \gamma V(s_{k+1})\right]$ and the old predicted value of *s*, $V(s_k)$.

The temporal-difference algorithm only updates the values of the states in the trajectory leading to a given outcome, given the values of its successors, and the reward associated with the transition departing from that state. This is a first-reward-based learning algorithm (and thus known as TD(0)), as it uses the value of the next state as a proxy for the value of all the other rewards, i.e., the forecast for the value of each state *k* is based on the one-step Bellman equation $V^{\pi}(s_k) = E\left[u_{t+1} + \gamma V^{\pi}(s_{k+1})\right]$.

The *n*-step Bellman equation $V^{\pi}(s_k) = E\left[\sum_{i=1}^{n}\left(\gamma^{i-1}u_{k+i}\right) + \gamma^n V^{\pi}(s_{k+n})\right]$ can approximate the expected value of a state s_k. In this case, the expected value of *n* future rewards and of the value of the state reached after *n* steps is used to forecast the value of a certain state *s* at stage *k* (s_k), under a policy π (Bertsekas and Tsitsiklis, 1996).

Moreover, this means that instead of having a one-step updating function it is possible to have a two-step, three-step, or generically an *n*-step updating function:

$$V(s_k) := V(s_k) + \alpha\left[u_{k+1} + \gamma u_{k+2} + \ldots + \gamma^{n-1} u_{k+n} + \gamma^n V(s_{k+n}) - V(s_k)\right].$$

Consequently, the difference between a one-step and an *n*-step TD method is that an agent computes a new estimate of the true value of the state by looking *n*-steps into the future. If there is no preference on the number of steps, instead of looking several steps ahead, an agent can use instead the weighted average of possible multi-step Bellman equations.

The Temporal Differences (λ), TD(λ), algorithm in equation (3) is a particular way of averaging the multi-step Bellman equations. This average includes all the *n*-step equations weighted proportionally to λ^{n-1}, where $0 \le \lambda \le 1$. Equation (3) can be approximated by equation (4) representing the updating function of the TD(λ) algorithm.

$$V^{\pi}(s_k) := V^{\pi}(s_k) + E\left[\sum_{m=k}^{+\infty} (\gamma\lambda)^{m-k} d_m\right]. \tag{3}$$

$$V^{\pi}(s_k) := V^{\pi}(s_k) + \alpha\sum_{m=k}^{+\infty} (\gamma\lambda)^{m-k} d_m . \tag{4}$$

Equations (3) and (4) are appropriate for off-line problems (in which the value of every state is updated after the trajectory reaches a terminal state), however, they are not adequate for problems where a terminal state does not exist, or in which there are trajectories that may be infinite. In order to apply temporal-differences methods to these types of problems, an on-line TD(λ) has to be defined instead.

In the on-line TD(λ) algorithm there is a memory variable associated with each state, known as *eligible trace*. This variable determines how the process of updating the state-value equation takes into account a certain d_k difference. Eligibility traces are memory parameters,

associated with a certain state, that enable the solution of the *temporal credit assignment problem* by giving more credit, i.e., higher trace, to the states visited more recently.

The eligible trace for a state *s* at time *t* is denoted as $e(s) \in R^+$:

$$e(s) := \begin{cases} \gamma\lambda e(s) & \text{if } s \neq s_t \\ 1+\gamma\lambda e(s) & \text{if } s = s_t \end{cases}, \qquad \forall s \in S,$$

where s_t is the state of the environment at time *t*. The eligibility trace decays for all states at a rate $\gamma\lambda$, except for the last state visited where the eligibility trace is incremented by one unit. λ is the *trace decay parameter*, it determines how different states are assigned a certain prediction error, given the discount rate. Let $d_k \equiv u_{k+1} + \gamma V^{\pi}(s_{k+1}) - V^{\pi}(s_k)$ and let $e_k(s)$ represent the eligibility trace of state *s* at time *k*. Then, in the on-line TD(λ) algorithm, for every *s*, equation $V^{\pi}(s_k) := V^{\pi}(s_k) + \alpha . e_k(s) . d_k$ approximates the state-value function.

THE Q-LEARNING ALGORITHM

In the control problem an agent chooses the action, in a given state, that gives him the optimal, or almost optimal, reward. The way an agent deals with exploitation classifies his learning behavior as on-policy or off-policy. Off-policy algorithms may update the value of a given state based on actions other than the one actually executed during the episode. During a given trajectory Γ an agent may choose a certain action a', at state *s*, (because he is exploring), but when updating the estimated value of *s*, the agent may assume that in future repetitions of the game (also known as episodes) he will choose the action with the higher expected value. On the other hand, on-policy algorithms update the value of a state strictly based on the experience gained from executing some policy. In this case, if in a state *s* the agent chooses an action a' (because he is exploring)

then, when updating the expected value of this state, the agent uses the expected value of action a' (and the value of the state reached by executing it).

The distinction between these two types of policy evaluation has important implications both on the policies derived and on the convergence properties of the algorithms applied (Singh et al., 2000). The policies derived using on-policy algorithms are *safer* than policies derived using off-policy algorithms. While on-policy algorithms take into account the effects of exploration on the overall reward of the policy, off-policy algorithms take into account only the reward of the potentially optimal policy. Hence, in simulated environments, the off-policy algorithms can find the optimal policy, but their use in real-world problems may have disastrous consequences.

However, if the parameters of a given off-policy algorithm are such that it always follows a greedy policy, and never explores, it may have good performances in real world environments. Nonetheless, in this case, there is no distinction between on-policy and off-policy algorithms, as this distinction relies on the manner the algorithms use exploration.

The convergence properties of off-policy and on-policy algorithms are also different. In an off-policy algorithm, the updating function does not take into account the policy used. Therefore, convergence (e.g., the *Q*-learning presented below) requires the algorithm to try each action (in every state) infinitely often. On the contrary, the on-policy algorithm learns the value of the actions actually taken, and converges to the optimum only if, in the limit, it chooses the optimal actions. Hence, in this case, convergence occurs only if the algorithm tries each action (in every state) infinitely often and if, in the limit, the learning algorithm is greedy with probability one.

Q-learning and Sarsa are two different algorithms for on-line control problems. While the former uses off-policy learning, the latter applies on-policy learning. The rest of this section presents these two algorithms in order to illustrate and discuss the application of reinforcement learning to the control problem.

Q-learning is a temporal-differences off-policy algorithm used in the control problem. Again, let $P_{ss'}^{a}$ stand for the state transition probabilities, representing the probability of the system moving from state s to state s', conditional on the agent choosing action *a*.

Furthermore, let $Q^{*}(s,a)$ be an optimal *Q*-value, i.e., the maximum expected value associated with action *a*, in state *s*, then $Q^{*}(s,a)=\sum_{s'\in S} P_{ss'}^{a}\left[u_{ss'}^{a}+\gamma V^{*}(s')\right],\ \forall s\in S$. $Q^{*}(s,a)$ represents the expected value of an action *a*, in a state *s*, assuming that the optimal policy is followed within trajectory Γ. Equation (5) represents the Bellman's equation (actually a set of equations) used to compute the optimal value of each state, given the expected value of every action. By combining $Q^{*}(s,a)$ and $V^{*}(s)$ the expected value of an action *a* can be defined, for a state *s*, as a function of the actions in the following states within Γ, equation (6).

$$V^{*}(s)=\max_{a} Q^{*}(s,a),\ \forall s\in S. \tag{5}$$

$$Q^{*}(s,a)=\sum_{s'\in S} P_{ss'}^{a}\left[u_{ss'}^{a}+\gamma \max_{b} Q(s',b)\right],\ \forall s\in S. \tag{6}$$

In equation (6) it is clear that the expected value of an action *a*, in a state *s*, assumes that the optimal policy is followed in the future. Watkins and Dayan (1992) show that the one-step *Q*-learning converges to the optimal action-values with probability one if every action-value is tried infinitely often (in the discrete case).

The value iteration algorithm in equations (7) and (8) approximates the optimal *Q*-factors. This updating process follows the assumption that the algorithm, in the future iterations, follows the optimal policy.

$$Q(s,a) \coloneqq Q(s,a) + \alpha.e(s,a).\left[u_{ss'}^{a} + \max_{b} \gamma Q(s',b) - Q(s,a)\right],\ \forall s \in S,\ a \in A(S) \tag{7}$$

$$e(s,a) \coloneqq \begin{cases} \gamma\lambda e(s,a) \\ 1 + \gamma\lambda e(s,a) \end{cases} \text{ if } \begin{matrix} (s,a) \neq (s_t,a_t) \\ (s,a) = (s_t,a_t) \end{matrix}. \tag{8}$$

An on-policy control algorithm, such as Sarsa, computes the expected value of a policy taking into account the possibility of explorative behavior (an explorative action may be non-optimal at a certain stage). In the Sarsa (λ) algorithm, the action value function $Q(s,a)$ is approximated by the updating equation $Q(s,a) \coloneqq Q(s,a) + \alpha.e(s,a).\left[u_{ss'}^{a} + \gamma Q(s',b) - Q(s,a)\right]$, $\forall s \in S$, $a \in A(S)$, where the eligible trace is computed using equation (8).

CONCLUSIONS AND FUTURE TRENDS

Reinforcement learning is useful in modelling problems where an agent lacks information about the uncertain parameters as it is able to work in settings where a model of the environment is not available.

A first shortcoming of reinforcement learning models is the assumption that an agent only learns by interacting with the environment. A second issue with reinforcement learning is the way it deals with the exploration versus exploitation problem. In all the approaches that have been analysed (*n*-armed bandit, *Q*-learning and Sarsa), this is a central issue as it influences the value of the different policies. In addition, if, as discussed, the Q-learning and the Sarsa algorithms present alternative ways of dealing with this problem, all of them still assume that a player is willing to try any action he wishes to evaluate, which is clearly a very strong

assumption when working in environments in which the cost of experimenting with possible actions is potentially very costly.

In reality, an agent learns by being told, by observing how others behave, by deduction, and by induction. I, therefore, envision that a way to improve the learning ability of these computer algorithms is to integrate these different types of learning methods into a coherent framework which allows computational learning to be more efficient and more human-like.

REFERENCES

Arthur. W. B. (1991). Designing Economic Agents that Act like Human Agents: A Behavioral Approach to Bounded Rationality. *The American Economic Review*, 81 (2), Papers and Proceedings of the Hundred and Third Annual Meeting of the American Economic Association, 353-359.

Bae G. I., Kim, W. C., and Mulvey, J. M. (2013). Dynamic Asset Allocation for Varied Financial Markets under Regime Switching Framework. European Journal of Operational Research, http://dx.doi.org/10.1016/j.ejor.2013.03.032.

Baxter, J., Tridgell, A., & Weaver, L. (2000). Learning to Play Chess using Temporal Differences. *Machine Learning*, 40 (3), 243 – 263.

Bellman, R. (1957). Dynamic programming, Princeton University Press.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). Neuro-Dynamic Programming. Athena Scientific, Belmont, Massachusetts.

Bertsekas, D. P. (2000). Dynamic Programming and Optimal Control. Vol. I and II, second edition. Athena Scientific, Belmont, Massachusetts.

Bunn, D. W. & Oliveira, F. S. (2001). Agent-Based Simulation: An Application to the New Electricity Trading Arrangements of England and Wales. *IEEE - Transactions on Evolutionary Computation*, 5 (5), 493-503.

Bunn, D. W., & Oliveira, F. S. (2003). Evaluating Individual Market Power in Electricity Markets via Agent-Based Simulation. *Annals of Operations Research*, 121 (1-4), 57-77, 2003.

Gulpinar, N., & Oliveira, F. S. (2012). Robust Trading in Spot and Forward Oligopolistic Markets. *International Journal of Production Economics*, 138 (1), 35-45.

Howard, R. A. (1960). Dynamic programming and Markov Processes. MIT Press.

Howard, R. A. (1971). Dynamic probabilistic systems. John Wiley.

Hsu, H.-M., & Wang, W.-P. (2004). Dynamic Programming for Delayed Product Differentiation. *European Journal of Operational Research*, 156, 183-193.

Kaelbling L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4, 237–285.

Kraft, H., & Steffensen, M. (2013). A Dynamic Programming Approach to Constrained Portfolios. *European Journal of Operational Research*, 229, 453-461.

Murphy, F., & Oliveira, F. S., (2010). Developing a Market-Based Approach to Managing the US Strategic Petroleum Reserve. *European Journal of Operational Research*, 206 (2), 488-495.

Nicolaisen J., Petrov, V., & Tesfatsion, L. (2001). Market Power and Efficiency in a Computational Electricity Market with Discriminatory Double-Auction Pricing. *IEEE Transactions on Evolutionary Computation*, 5 (5), 504- 523.

Raju, C. V. L., Narahari, Y., & Ravukumar, K. (2006). Learning Dynamic Prices in Electronic Retail Markets with Customer Segmentation. *Annals of Operations Research*, 143, 57-75.

Ross, S. (1983). Introduction to Stochastic Dynamic Programming. Academic Press.

Roth, A. E., & Erev, I. (1995). Learning in Extensive-Form Games: Experimental Data and Simple Dynamic Models in the Intermediate Term. *Games and Economic Behavior*, 8, 164-212.

Samuel, A. L. (1959). The history of heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Learning*, 11 (11), 1203-1212

Singh, S. P. (1994). Learning to solve markovian decision processes. Doctor of Philosophy Dissertation, Department of Computer Science, University of Massachusetts.

Singh, S., Jaakkola, T., & Szespesvári, C. (2000). Convergence Results for Single-Step On-line Reinforcement Learning Algorithm. *Machine Learning*, 38 (3), 287 – 308.

Sutton, R. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3, 9-44.

Sutton, R. S., & Barto, A. G. (1998). Reinforcement Learning: An Introduction. MIT Press.

Tesauro, G. (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, 8, 257-277.

Tesauro, G. (1994). TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 6, 215-219.

Weiss, G. (1995). Adaptation and Learning in Multi-Agent Systems: Some Remarks and a Bibliography. *Adaptation and Learning in Multiagent Systems*, G. Weiss and S. Sen, Ed. Springer, 1-21.

Zhang, Z., Zheng, L., Hou, F., & Li, N. (2011). Semiconductor Final Test Scheduling with Sarsa((λ,k) algorithm. *European Journal of Operational Research*, 215, 446-458.

KEY TERMS & DEFINITIONS

Action-Value Function: it represents the expected return of an action in a given state by following a certain policy. It stands for the expected present value of all the rewards received during the planning horizon, by following this previously defined policy, starting from a given state.

Credit Assignment: it is the process of identifying among the set of actions chosen in an episode the ones which are responsible for the final outcome. And moreover, it is an attempt to identify the best and worse decisions chosen during an episode, so that the best decisions are reinforced and the worst decisions penalized.

Dynamic Programming: dynamic programming is the mathematical theory of multi-stage decision processes, aiming to compute the *optimal policy*, given that the environment follows a Markovian process.

Eligibility Trace: this is a parameter used to control the "memory" of the algorithm associated to a given state that enables the assignment of more credit, i.e., higher trace, to the states and actions performed more recently.

N-armed bandit: in this reinforcement learning algorithm the learner decides which action to choose in the search of the highest expected reward. This is a stochastic problem in which each action has a certain expected value unknown to the player. In order to learn the expected reward associated with each action an agent is required to try non-optimal actions during a few iterations (exploration) instead of repeatedly choosing the action with the highest expected value at a certain time (exploitation).

Q-Learning: In this reinforcement learning algorithm the agent evaluates each action (in the possible states of the world) taking into account its possible repercussions in the future behavior of the system. Q-learning is a temporal-differences off-policy algorithm used in the control problem. It is an algorithm for on-line control problems in which values of the states are updated after having observed the outcomes of the actions chosen in a given sequence. Q-learning uses *off-policy* learning.

Reinforcement learning: in the context of computational learning, it stands for a family of algorithms aimed at approximating the best policy to play in a certain environment (without building an explicit model of it) by increasing the probability of playing actions that improve the rewards received by the agent.

SARSA algorithm: it is a reinforcement-learning algorithm for the on-line control problem that uses *on-policy* learning. Sarsa, computes the expected value of a policy taking into account the possibility of explorative behavior (an explorative action may be non-optimal at a certain stage).

Temporal-differences algorithm: it is a forecasting method. Whereas the conventional forecasting methods use the difference between predicted and actual outcomes as their learning signal, the temporal-differences algorithm looks at the difference between successive predictions. Using temporal-difference methods learning occurs whenever there is a change in predictions.