Multi-Agent Reinforcement Learning

Introduction

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer
Slides by: Leonard Hinckeldey

Multi-Agent Reinforcement Learning: Foundations and Modern Approaches

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer

Published by MIT Press
(print version will be released on 17 December 2024)

This lecture is based on *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches* by Stefano V. Albrecht, Filippos Christianos and Lukas Schäfer

The book can be downloaded for free at www.marl-book.com.

Lecture Outline

Part 1: Multi-Agent Reinforcement Introduction

- Multi-agent systems
- Advantages of MARL vs SARL in multi-agent systems
- Challenges of MARL

Part 2: Reinforcement Learning Basics

- Markov-decision process
- Discounted returns
- Dynamic programming and temporal-difference learning

Part 1: Multi-Agent Reinforcement Introduction

What is MARL?

Multi-agent reinforcement learning (MARL) is about finding optimal decision policies for two or more artificial agents interacting in a common environment.

- Applying reinforcement learning (RL) algorithms to multi-agent systems.
- The aim is to learn optimal policies for two or more agents independently.

MARL Applications

Figure: Competitive games

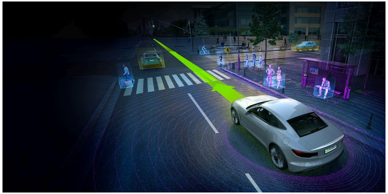


Figure: Autonomous driving

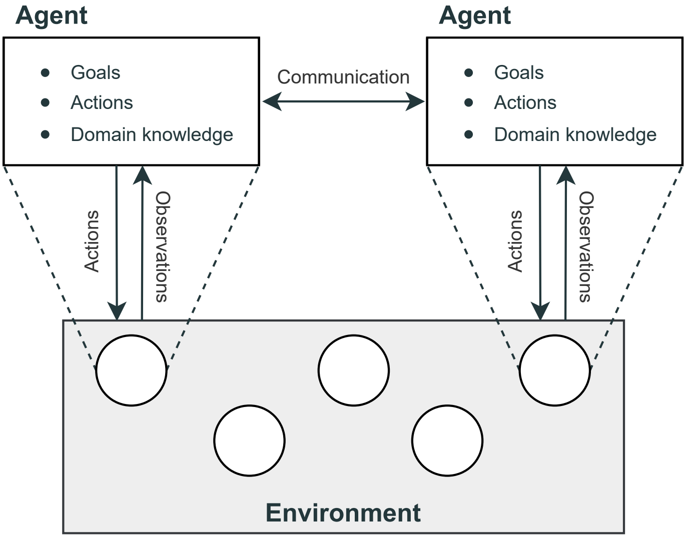
Figure: Multi-robot warehouse management

Figure: Automated trading in electronic markets

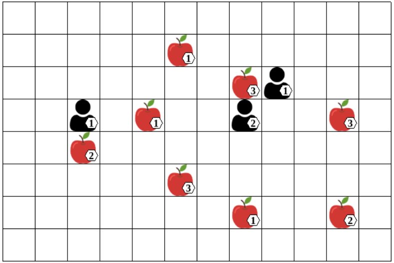
Multi-agent systems consist of:

- **An environment:** The environment is a physical or virtual world whose state evolves and is influenced by the agents' actions within the environment.
- **Agents:** An agent is an entity which receives information about the state of the environment and can choose actions to influence the state. Agents are goal-directed, e.g. maximizing returns.

Multi-Agent Systems

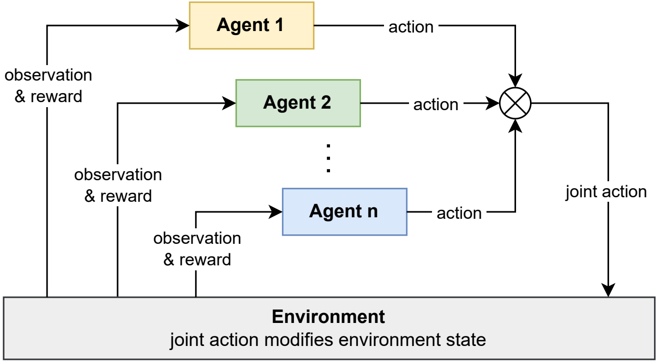


Level Based Foraging Example



- 3 agents with varying skill levels.
- Goal: to collect all apples.
- Items can be collected if a group of one or more agents are located next to the item and the sum of all agents levels is greater or equal to the item level.
- Action space A is $\{up, down, left, right, collect, noop\}$

MARL for solving Multi-Agent Systems



- Goal: learn optimal policies for a set of agents in a multi-agent system.
- Each agent chooses an action based on its policy; the actions of all agents together form a joint action.
- The environment transitions to a new state based on the joint action.

Why MARL?

Why should we use MARL to find optimal solutions to multi-agent systems rather than controlling multiple ’agents’ using a single-agent RL (SARL) algorithm? I.e. one agent controlling the actions of all agents.

Decomposing a larger problem

- For example, controlling 3 agents each with 6 actions (see LBF example), the action space becomes $6^3 = 216$.
- Using MARL, we decompose this into three more tractable problems.

Decentralized decision making

- There are many real-world scenarios where it is beneficial for each agent to make decisions independently.
- E.g. for autonomous driving is impractical for frequent long-distance data exchanges between a central agent and the vehicle.

Challenges of MARL

While it might be advantageous to use MARL in certain settings, several challenges arise or are amplified in MARL compared to SARL.

- Non-stationarity caused by learning agents
- Optimality of policies and equilibrium selection
- Multi-agent credit assignment
- Scaling in number of agents

We will explore these challenges more thoroughly in upcoming lectures.

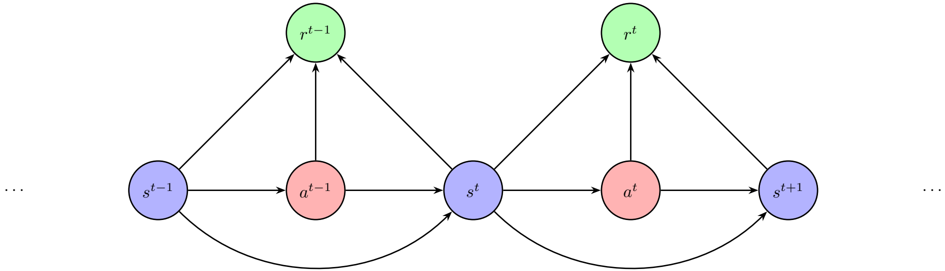
Part 2: Reinforcement Learning Basics

Back to RL Basics

RL Problem	=	Decision Process Model *e.g. MDP, POMDP, multi-armed bandit*	+	Learning Objective *e.g. discounted return with specific discount factor*

- RL algorithms learn solutions for sequential decision problems via repeated environment **interaction**.
- The **goal** is to learn an optimal decision policy for a specific objective within an environment.
- A sequential decision process is usually defined more formally as a Markov decision process (MDP).

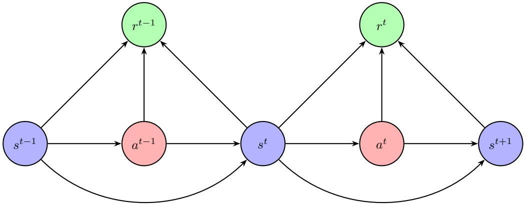
MDP Interaction



MDP continued

A MDP can be defined as a 5 tuple $(S, A, \mathcal{R}, \mathcal{T}, \mu)$:

- S: Finite set of states with subset of terminal states $\bar{S} \subset S$
- A: Finite set of actions.
- $\mathcal{R}$: Reward function $\mathcal{R} : S \times A \times S \to \mathbb{R}$
- $\mathcal{T}$: State transition function $\mathcal{T} : S \times A \times S \to [0, 1]$
- μ: Initial state distribution $\mu : S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$ and $\forall s \in \hat{S} : \mu(s) = 0$



MDP assumptions

Markov Property

- Future states are temporally independent of past states and actions, given the current state and action.
- $Pr(s^{t+1}, r^t | s^t, a^t, s^{t-1}, a^{t-1}, ..., s^0, a^0) = Pr(s^{t+1}, r^t | s^t, a^t)$

Full Observability

- The agent knows the entire state of the world.
- In practice, this is often violated, and we have Partially Observable MDPs (POMDP).

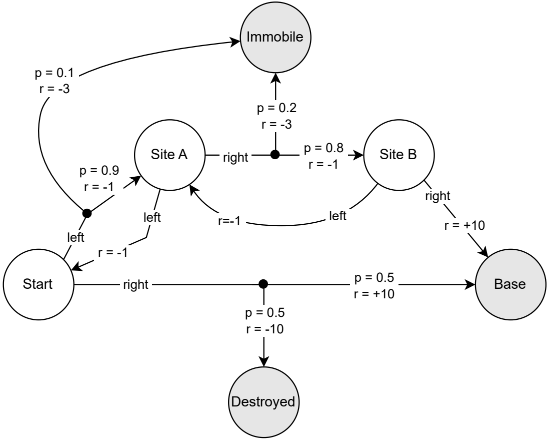
Stationarity

- The dynamics of the environment are assumed to be stationary.
- i.e. $\mathcal{T}$ and $\mathcal{R}$ remain constant through time.

No Knowledge of MDP Dynamics

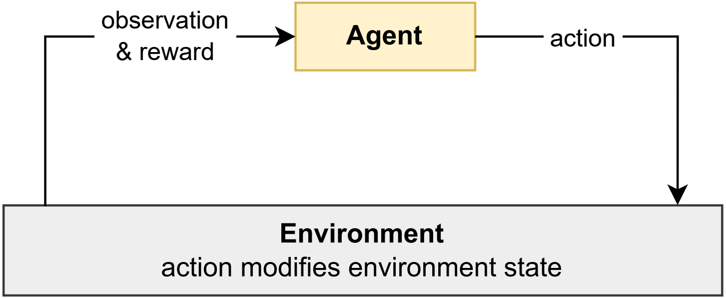
- In RL it is assumed that the agent has knowledge only of the action and state spaces (A, S)
- The transition and reward function $(\mathcal{T}, \mathcal{R})$ are assumed to be unknown.

Mars Rover MDP Example



- *Start* is the initial state s^0
- Two possible actions $A = \{right, left\}$
- Goal is to get to *Base*
- Rewards given by $\mathcal{R}(s, a, s')$ are shown as *r*.
- State transition probabilities given by $\mathcal{T}(s, a, s)$, are shown as *p*

RL for optimizing decision-making in MDPs



Value-Based RL

These methods indirectly update the policy by approximating value functions.

Policy-Based RL

These methods update a parameterized policy function directly.

Expected Discounted Returns

Using RL, we aim to maximize the expected sum of **returns**.

- Returns (u) are the sum of rewards received through time
- Typically, we **discount** returns as this ensures finite (discounted) returns (if discount factor $\gamma < 1$)

$$\mathbb{E}_\pi[u_t] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r^t \right]$$

- γ is the discount factor, such that $\gamma \in [0, 1]$
- π is the behavior policy, that determines which actions are chosen. (We discuss these further in slide 16)

State-Value Functions

State-value functions $V^\pi(s)$, give the 'value' of state s, when following the policy π.

$$V^\pi(s) = \mathbb{E}_\pi \left[u^t | s^t = s \right]$$

The return (u) can be recursively defined as:

$$\begin{aligned} u^t &= r^t + \gamma(r^{t+1} + \gamma r^{t+2} + \ldots) \\ &= r^t + \gamma u^{t+1} \end{aligned}$$

Therefore:

$$V^\pi(s) = \mathbb{E}_\pi \left[r^t + \gamma u^{t+1} | s^t = s \right]$$

Deriving the Bellman Equation

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}\left[r^{t} + \gamma u^{t+1} \mid s^{t} = s\right] \\
&= \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} \mathcal{T}(s' \mid s, a)\left[\mathcal{R}(s, a, s') + \gamma \mathbb{E}_{\pi}\left[u^{t+1} \mid s^{t+1} = s'\right]\right] \\
&= \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} \mathcal{T}(s' \mid s, a)\left[\mathcal{R}(s, a, s') + \gamma V^{\pi}(s')\right]
\end{aligned}
$$

The last equation is known as the Bellman equation in honor of Richard Bellman.

- The equation states that the value of being in state s while following a fixed policy π is equivalent to the immediate reward ($\mathcal{R}(s, a, s') \rightarrow r$) received when taking action a in state s ($\pi(a \mid s)$), and the subsequent state's **expected** value.

State-Action Value Function

State-Action value function $Q^\pi(s, a)$ are an extension on the *State* value functions. They condition the expected return on the current state and the action taken.

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[u^t | s^t = s, a^t = a \right]$$

The *state-action* value Bellman equation is therefore:

$$Q^\pi(s, a) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} \mathcal{T}(s'|s, a) \left[\mathcal{R}(s, a, s') + \gamma Q^\pi(s') \right]$$

Greedy Policies

A policy might act *greedily* i.e. choosing actions which maximize the immediate reward and the value of the next state.

Greedy π using *state value* functions:

$$\pi(s) = \arg\max_{a \in A} \sum_{s',r} \mathcal{T}(s', r | s, a) \left[r + \gamma V(s') \right]$$

Or using the *state-action value function*:

$$\pi(s) = \arg\max_{a \in A} Q(s, a)$$

Optimal Greedy Policy

A greedy policy with respect to a value function becomes optimal only when using the **optimal value function.**

An **optimal value function** for a MDP can be defined as:

$$V^*(s) = \max_{\pi'} V^{\pi'}(s), \quad \forall s \in S$$
$$Q^*(s,a) = \max_{\pi'} Q^{\pi'}(s,a), \quad \forall s \in S, a \in A$$

Therefore, the optimal policy is:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a)$$

Dynamic Programming

- Dynamic Programming (DP) is a family of algorithms to compute **optimal value functions** and **optimal policies in MDPs** (Bellman 1957; Howard 1960).
- In DP, we **assume complete knowledge** of the MDP, including the transition and reward function $(\mathcal{T}, \mathcal{R})$.
- Given complete knowledge, we can use the **Bellman equation** to find optimal value functions and policies.

Policy Iteration

Policy iteration, is a DP algorithm that alternates between 2 tasks:

- **Policy evaluation**: computing value function V^π for current policy π.
- **Policy improvement**: improve current policy π with respect to V^π.

$$\pi^0 \rightarrow V^{\pi^0} \rightarrow \pi^1 \rightarrow V^{\pi^1} \rightarrow \pi^2 \rightarrow \ldots \rightarrow V^* \rightarrow \pi^*$$

Policy Iteration Pseudo Code

Algorithm Policy Iteration

Initialize π randomly, initialise $V(s)$ arbitrarily for all $s \in S$ except $V(\text{terminal}) = 0$
repeat
  Policy Evaluation:
  repeat
    for each state s in S **do**
      $V(s) \leftarrow \sum_{s'} \mathcal{T}(s'|s, \pi(s))[\mathcal{R}(s, \pi(s), s') + \gamma V(s')]$
  until $V(s)$ converges for all $s \in S$
  Policy Improvement:
  policy_stable $\leftarrow$ true
  for each state s in S **do**
    old_action $\leftarrow \pi(s)$
    $\boxed{\pi(s) \leftarrow \operatorname{arg\,max}_a \sum_{s'} \mathcal{T}(s'|s, a)[\mathcal{R}(s, a, s') + \gamma V(s')]}$
    if old_action $\neq \pi(s)$ **then**
      policy_stable $\leftarrow$ false
until policy_stable **return** V, π

Value Iteration

- **Value Iteration** uses the **Bellman optimality equation**.
- This combines iterative policy evaluation and improvement into one single update equation.

Bellman Optimality Equation as update operator:

$$V^{k+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} \mathcal{T}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^k(s')], \quad \forall s \in S$$

- The max operator makes this the Bellman *optimality* equation.
- The equation expresses the value of a state as the maximum expected return achievable by taking the best action and then following the optimal policy thereafter.

Value Iteration Pseudo Code

Algorithm Value Iteration

Initialize: $V(s) = 0, \forall s \in S$

repeat

$\quad \boxed{\forall s \in S : V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma V(s')\right]}$

until V converged **return** optimal policy π^* with:

$\forall s \in S : \arg\max_{a \in A} \sum_{s' \in S} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma V(s')\right]$

Temporal-Difference (TD) learning is a family of RL algorithms which learn optimal policies and value functions based on data collected through environment **interactions**.

- These algorithms learn which actions yield the best returns by **trial and error** and exploring different actions and states

Some **advantages** of this include:

- No need for a **model** of the environment's **reward** and **transition** function $(\mathcal{R}, \mathcal{T})$.
- They can learn **online**, updating the policy while interacting with the environment.

Temporal Difference Update

The update for Temporal Difference learning relies exclusively on value functions.

$$V(s^t) \leftarrow V(s^t) + \alpha \left[\mathcal{X} - V(s^t)\right]$$

or

$$Q(s^t, a^t) \leftarrow (s^t, a^t) + \alpha \left[\mathcal{X} - (s^t, a^t)\right]$$

Where $\mathcal{X}$ is the update target, serving as an estimate of the current state value. α is the learning rate.

Temporal Difference Update

In SARSA (a basic TD algorithm), we use the experience tuple $(s^t, a^t, r^t, s^{t+1}, a^{t+1})$, to construct a target:

$$\mathcal{X} = r^t + \gamma Q(s^{t+1}, a^{t+1})$$

(The immediate reward plus the discounted value of the next state) - note the resemblance of the Bellman equation.

$$Q^\pi(s, a) = \sum_{a \in A} \pi(a \mid s) \sum_{s' \in S} \mathcal{T}(s' \mid s, a) \boxed{[\mathcal{R}(s, a, s') + \gamma Q^\pi(s', a')]}$$

SARSA

The SARSA update rule thus becomes:

$$Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha[r^t + \gamma Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t)]$$

- Note the TD error $r^t + \gamma Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t)$.
- The TD update is **bootstrapped**
- Using the value **estimates** of the next state ($Q(s^{t+1}, a^{t+1})$) to update the current state value ($Q(s^t, a^t)$)

SARSA Pseudo Code

Algorithm SARSA

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$
for every episode **do**
    Observe initial state s^0
    With probability ϵ: choose random action $a^0 \in A$
    Otherwise: choose action $a^0 \in \arg\max_a Q(s^0, a)$
    for $t = 0, 1, 2, \ldots$ **do**
        Apply action a^t, observe reward r' and next state s^{t+1}
        With probability ϵ: choose random action $a^{t+1} \in A$
        Otherwise: choose action $a^{t+1} \in \arg\max_a Q(s^{t+1}, a)$
        $\boxed{Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha[r^t + \gamma Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t)]}$

Convergence of SARSA

SARSA is guaranteed to converge to the optimal **state-value function**, for all $S \in S$ and $a \in A$, if:

- All *state-action* pairs are explored infinitely many times.

$$\forall s \in S, a \in A : \sum_{k=0}^{\infty} \mathbb{I}(s,a) \to \infty$$

- The learning rate α is reduced over time according to the "standard stochastic approximation condition".

$$\forall s \in S, a \in A : \sum_{k=0}^{\infty} \alpha_k(s,a) \to \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \alpha_k(s,a)^2 < \infty$$

ϵ-Greedy Policies

Using a **greedy policy** would **violate the convergence condition** of SARSA (infinite exploration of S and A).

- Intuitively, we must explore a wide range of states and actions to find state action combinations that yield high returns
- One solution is to add an **exploration** parameter $\epsilon \in [0, 1]$. This makes gives us a stochastic **epsilon-greedy** policy.

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if } a \in \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases}$$

- With probability $1 - \epsilon$, the policy chooses the greedy action, and with probability ϵ chooses an action uniformly at random.

Q-Learning

Q-learning is a popular TD algorithm which uses the Bellman optimality equation to update its value function estimates.

- By using the Bellman optimality equation, Q-learning directly learns the **optimal state-action value function**
- Q-learning is **off-policy**, meaning the policy followed to gather experiences is different from the optimized policy
- We use the ϵ-greedy policy to collect experiences

Q-Learning Update

The target in Q-learning uses the *max* operator to target the optimal Q-values directly.

$$\mathcal{X} = r^t + \gamma \max_{a' \in A} Q(s^{t+1}, a')$$

The Q-learning update is thus:

$$Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha \left[r^t + \gamma \max_{a' \in A} Q(s^{t+1}, a') - Q(s^t, a^t) \right]$$

Q-Learning Pseudo Code

Algorithm Q-Learning

```
Initialize Q(s, a) = 0 for all s ∈ S, a ∈ A
for every episode do
    for t = 0, 1, 2, ... do
        Observe current state s^t
        With probability ε: choose random action a^t ∈ A
        Otherwise: choose action a^t ∈ arg max_a Q(s^t, a)
        Apply action a^t, observe reward r^t and next state s^{t+1}
        Q(s^t, a^t) ← Q(s^t, a^t) + α [r^t + γ max_{a'} Q(s^{t+1}, a') − Q(s^t, a^t)]
```

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$

for every episode **do**

  for $t = 0, 1, 2, \ldots$ **do**

    Observe current state s^t

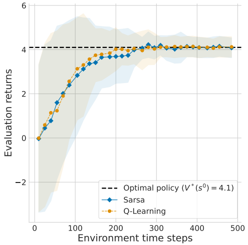
    With probability ϵ: choose random action $a^t \in A$

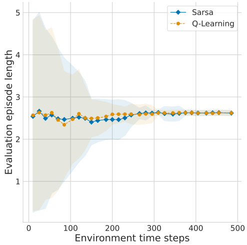
    Otherwise: choose action $a^t \in \arg\max_a Q(s^t, a)$

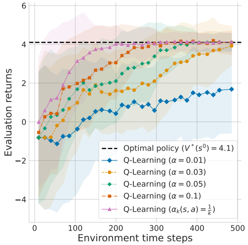
    Apply action a^t, observe reward r^t and next state s^{t+1}

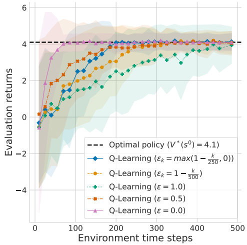
    $\boxed{Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha \left[r^t + \gamma \max_{a'} Q(s^{t+1}, a') - Q(s^t, a^t)\right]}$

Evaluating RL Algorithms









Y-axis:

- Average **discounted evaluation returns**. This shows us how our greedy policy would perform if we stopped learning at time step T.
- In some cases, **undiscounted returns** are easier to interpret and may be used instead.

X-axis:

- Cumulative training steps across episodes.
- Number of episodes can also be used. This might, however, distort the learning speed.

Summary

We covered:

- Multi-Agent Systems and the case for MARL
- MDPs
- Value Functions
- Dynamic Programming
- Temporal Difference Learning (SARSA and Q-Learning)

Next we'll cover:

- Games: Models of Multi-Agent Interaction

Sources:

Bellman, Richard. 1957. Dynamic Programming. Princeton University Press. Howard, Ronald A. 1960. Dynamic Programming and Markov Processes. John Wiley.