Autonomous University of Barcelona

Master thesis

Solving an Optimization Problem for Product Delivery with Reinforcement Learning and Deep Neural Networks

---

Autor:
Daniel Salgado Rojo

Director:
Toni Lozano Bagén[1]

Master's degree in Modelling for Science and Engineering
Mathematics Department[1]

Friday 22nd June, 2018

UAB Universitat Autònoma de Barcelona

GRUPO AIA

“Artificial Intelligence is the new electricity.”

Andrew Ng

Data Scientist and co-founder of Coursera

Abstract

This master thesis is motivated by a real world problem for Product Delivery (PD) - an optimization problem which combines inventory control and vehicle routing - inspired by a project from the company, Grupo AIA, where I have been doing an internship from March to June 2018. The solution proposed to the client uses classical constraint optimization techniques which tend to be slow when finding optimal solutions and do not scale properly when the number of shops and trucks used to deliver a product increases.

Machine Learning (ML) has become a very popular field in Data Science due to the increase in computation power in recent years. It is usually said that ML techniques divide in two types, Supervised Learning and Unsupervised Learning. However, this is not a complete classification. Apart from these two types, we must distinguish another one which is very different from those two: Reinforcement Learning (RL). RL consists of techniques that have been used for decades in the field of Artificial Intelligence for many applications in fields such as robotics and industrial automation [1], health and medicine [2, 3], Media and Advertising [4, 5, 6], Finance [7], text, speech and dialog systems [8, 9], and so forth.

RL provides a nice framework to model a large variety of stochastic optimization problems [10]. Nevertheless, classical approaches to large RL problems suffer from three curses of dimensionality: explosions in state and action spaces, and a large number of possible next states of an action due to stochasticity [11, 12]. There is very few literature about the application of Reinforcement Learning to a PD optimization framework. The only paper we have found that focus on the practical application of RL to the domain of PD is from S. Proper and P. Tadepalli (2006) [12]. In that paper they propose a variant of a classical RL technique called ASH-learning, where they use "tabular linear functions" (TLF) to learn the so-called H-values, which are then used to decide how to control the product delivery system of interest. Proper and Tadepalli show the results of controlling a simplistic and discretised system of 5 shops and 4 trucks with ASH-learning, where the three curses of dimensionality are present, and the results are successful for that particular example with an small number of trucks and shops. However, in practical situations, the number of shops and trucks may be so large (for instance, lets say 30 shops and about 7 to 10 trucks) that the explosion of the dimensionality of the state and action spaces would make those classical RL techniques impractical.

In this thesis [13] we present a novel approach for solving product delivery problems by means of Reinforcement Learning and Deep Neural Networks (DNN), a field also referred to as Deep Reinforcement Learning (DRL). The idea is that the nonlinearity and complexity of DNN should be better for learning to solve complex optimization problems than TLF, and the tabular functions in general that have been used so far in classical RL. Moreover, we expect that DNN could be the key to solve some of the curses of dimensionality such as the explosion of the state-action spaces; in the framework of PD, we expect them to scale better than classical approaches to systems with a large number of shops and trucks. In addition, we have developed an OpenAI gym environment for our PD problem which is available in a GitHub repository [14] [1].

[1] Open AI gym is a standard open library containing environments for RL algorithms, and its standards are usually

followed by the research community.

Acknowledgements

To do.

Nomenclature

The units we specify here are the ones that will be used mostly for simulations.

Symbol	**Units**	**Description**
t	day	Time.
s	—	System's state.
$\mathcal{S}$	—	Set of all possible states.
a	—	Action.
$\mathcal{A}$	—	Set of all actions.
$\mathcal{A}(s)$	—	Set of possible (not forbidden) actions from state s.
Λ	—	Total number of actions (cardinality of $\mathcal{A}$).
r	—	Reward after some action (scalar).
R	—	Function of rewards (function) or reward random variable, depending on the context.
T	—	Transition probability function.
π	—	A policy.
Q^π	—	Q-value (or state-action value) function with regards to some policy π.
V^π	—	V-value (or state value) function with regards to some policy π.
τ	day	Length (number of days) of an episode.
$\mathcal{G}$	—	Weighted and directed graph that defines the connections between shops and the depot.
$W_\mathcal{G} = (w_{ij})_{i,j}$	—	Weights for the edges of the model graph $\mathcal{G}$.
$A_\mathcal{G}$	—	Adjacency matrix of the model graph $\mathcal{G}$.
$\mathcal{N}$	—	The set of shops (modelled as tanks) in the system.
n	—	Number of shops in the system.
C_i	—	Maximum stock capacity of the i-th shop.
c_i	—	Current stock of the i-th shop.
$x^{(i)} = \frac{c_i}{C_i}$	—	Fraction of available stock in the i-th shop.
q_i	—	Expected total consumption of stock (after the end of the current day) for the i-th shop.

$\mathcal{K}$	—	The set of trucks in the system.
k	—	Number of trucks in the system.
L_i	—	Maximum load capacity of the i-th truck.
l_i	—	Current load of the i-th truck.
p_i	—	Id (number) of the shop or depot (if $i = n$) where the i-th truck is located.
$\lambda^{(i)}$	—	Quantity of product that the i-th truck is going to deliver to a given shop.
V	—	Number of shops that a given truck can at most visit every day.
$V_{\max}$	—	Maximum number of shops (among all trucks) that a truck can visit every day.
E	—	An episode, a sequence of (state, action, reward) random variables.
E_j	—	A simulated j-th episode, a sequence of observed (state, action, reward).
γ	—	Discount factor (for the discounted rewards).
α	—	A learning rate.
R_t^γ	—	Discounted average reward with discount factor γ.
$\mathcal{P}_{ss'}^a$	—	Transition probability of going from state s to s' after taking action a.
$\mathcal{R}_{ss'}^a$	—	Expected reward obtained after going from state s to s' by taking action a.
J (J_θ)	—	Cost function (dependent on parameters θ).
J_{costs}	—	Cost function term due to transport costs (or other costs).
J_{levels}	—	Cost function to model the *wellness* of shops.
J_{extra}	—	Extra contributions to the total cost function J (different from J_{costs} and J_{levels}).
μ_i	—	Weights for each J_i contribution to the total cost function J.
b	—	Minimum level of stock for a given shop (in percentage). Expected consumption of stock in the next 12 hours.
c	—	Danger level of stock for a given shop (in percentage). Expected consumption of stock in the next 36 hours.
e	—	Maximum level of stock for a given shop (in percentage).
π_ε	—	ε-greedy policy.
Uniform(a, b)	—	A uniform probability distribution in the interval (a, b).
X	—	A train dataset.
y	—	An array with the labels (targets) associated to some train dataset.
w_{jk}^l	—	The weight of the NN edge that connects the k-th neuron from the $(l-1)$-th with the j-th neuron from the l-th layer.

b_j^l	—	Bias parameter of neuron j in layer l.
z_j^l	—	Weighted input to neuron j in layer l.
a_j^l	—	Output of neuron j in layer l after applying the activation function to z_j^l.
δ_j^l	—	δ-error of neuron j in layer l.
σ	—	An activation function in general.
$H(p, q)$	—	Cross entropy loss between two probability distributions p, q.
π_m	—	Minimum stock policy.

Shortcut	**Description**
MDP	Markov Decision Process
RL	Reinforcement Learning
PG	Policy Gradient
ANN, NN	Artificial Neural Network
DNN	Deep Neural Network
GD	Gradient Descent
SGD	Stochastic Gradient Descent
ML	Machine Learning
SL	Supervised Learning
UL	Unsupervised Learning
AI	Artificial Intelligence
DL	Deep Learning
DRL	Deep Reinforcement Learning
TLF	Tabular Linear Function
ASH-learning	After-State H-learning

Contents

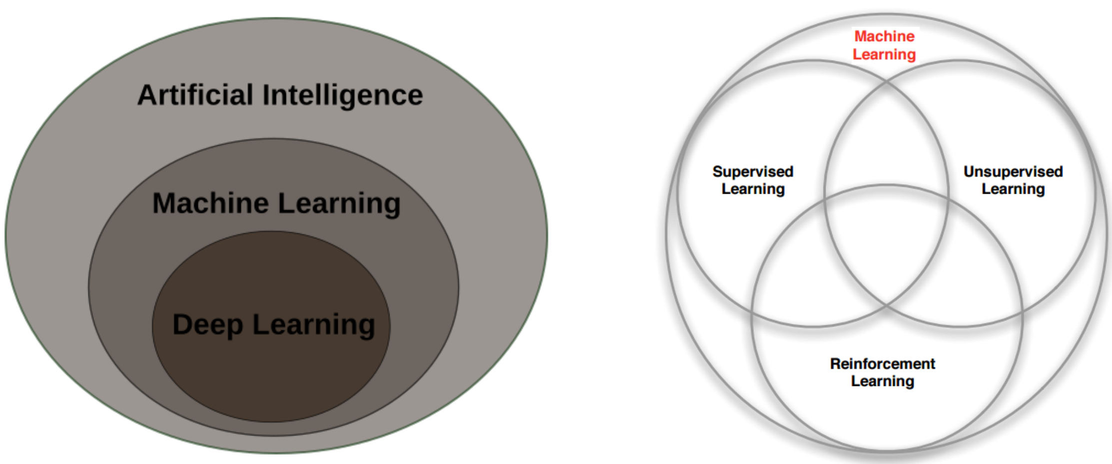
Chapter 1

Introduction

1.1 Artificial Intelligence, Machine Learning and Deep Learning

Artificial Intelligence (AI) and Machine Learning (ML) are two very hot buzzwords by the time of writing this thesis, and often seem to be used as if they were synonyms. Both terms appear very frequently when the topic is *Big Data* or *Data Science* in general, but they are not exactly the same.

Consider the Venn diagram from Figure 1.1a. On the one hand, Artificial Intelligence is the broader concept of machines being able to carry out tasks in a way that we would consider “smart” (e.g., following rules that a human would usually follow). On the other hand, Machine Learning would be the field of study of algorithms and methods that give computers the ability to learn without being explicitly programmed (e.g. using data) - Arthur Samuel, 1959 [15].



(a) (b)

Figure 1.1: (a) Venn diagram to distinguish Artificial Intelligence, Machine Learning and Deep Learning concepts. (b) Venn diagram to illustrate the three main subfields of Machine Learning in terms of the kind of supervision needed for training; adapted from [16].

AI and ML have been around for a long time. On the one hand, the Greek myths contain stories of mechanical men designed to mimic our own behaviour [17]. On the other hand, statistical models such as linear regression, random forests, principal components analysis, and so on, are examples of ML techniques that have been present for a long time [18].

However, the quantity of *data* and the computing power are crucial ingredients on which ML depends to succeed. That is why it has not been until the 21st century, with the increase of computing power and the advances in technology, that ML has become so popular.

"*In 2006, Geoffrey Hinton et al. published a paper [19] showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision (>98%). They branded this technique "Deep Learning." Training a deep neural net was widely considered impossible at the time, and most researchers had abandoned the idea since the 1990s. This paper revived the interest of the scientific community and before long many new papers demonstrated that Deep Learning was not only possible, but capable of mind-blowing achievements that no other Machine Learning technique could hope to match (with the help of tremendous computing power and great amounts of data). This enthusiasm soon extended to many other areas of Machine Learning*" [20] [1]

In twelve years from then, Machine Learning is contributing directly to most of the high-tech products, ranking our web search results, powering our smartphone's speech, footprint or face recognition, and recommending videos (e.g. YouTube) or songs, among others.

One of the most recent and astonishing events related to Machine Learning and Artificial Intelligence so far, has been Alpha Go [21], an AI software developed by Deep Mind[2] in 2015, that in March 2016 was able to beat Mr Lee Sedol, winner of 18 world titles and considered to be the greatest player of Go of the past decade. The game of Go was originated in China about 5000 years ago [22], and although the rules of the game are simple, the complexity of this game leads to an astonishing number of 10^{170} possible board configurations, much more than the number of atoms in the known universe (estimated to be around $\approx 10^{80}, 10^{90}$). To have an idea, Go is about a *googol* [3] times more complex than chess [4].

In October 2016, Deep Mind presented Alpha Go Zero. The original Alpha Go partially learned by having as input many Go real matches, whereas Alpha Go Zero was able to learn without previous knowledge, just playing against itself!. *In doing so, it surpassed the performance of all previous versions, hence becoming arguably the strongest Go (AI) player of all time* [21]. For this amazing achievement, Deep Mind published another *Nature* paper in 2017 [23].

At this point it is worth to talk about the main subfields of Machine Learning. ML systems can be classified according to the amount and type of supervision that they get during training. It is usually said that ML techniques divide in two types, Supervised Learning (SL) and Unsupervised Learning (UL). However, this is not a complete classification. In SL, the training data that one feeds to the algorithms in order to *train them* includes the desired solutions, called *labels* - e.g. regression models where one has an observation $x \in \mathbb{R}^n$ and a desired output y, approximated by the model via $f(x)$ for some f. On the contrary, in UL the training data is *unlabelled*, and algorithms belonging to this type of learning are aimed to extract information from data without explicitly knowing the solution a priori - usual examples include Principal Component Analysis (dimensionality reduction) and clustering methods.

Nevertheless, apart from these two types, we must distinguish another one which is very different from those two (see Figure 1.1b): Reinforcement Learning (RL). In RL, instead of having an initial training dataset from which to learn, the learning system called an *agent* interacts in an environment

[1] At this point the curious reader may consider to have a brief look at Chapter 5 to have some notions about what a Deep Neural Network is.

[2] *DeepMind Technologies Limited* is a British artificial intelligence company founded in September 2010 and acquired by Google in 2014.

[3] A *googol* is 10^{100}.

[4] I highly recommend to the reader to watch the Alpha Go movie to see the importance and the impact of Alpha Go in the field of AI, and the game of Go. Netflix: `https://www.netflix.com/es-en/title/80190844`.

and it is responsible to select and perform actions, getting *rewards* or *penalties* in return. The agent must learn by itself the best strategy (the so-called *policy*) to get the most reward over time. A policy determines (either in a probabilistic or deterministic way) what action the agent should take when it is in a given situation [5].

To fix ideas, Alpha Go is based on Reinforcement Learning techniques, but also used Supervised Learning, since it was initially trained by Go matches played by professionals. On the contrary, Alpha Go Zero is purely based in Reinforcement Learning, since it only learns by itself by interacting with an environment, the Go board where it plays against itself.

In this project we are going to focus on Reinforcement Learning and the application of Deep Neural Networks as a tool. In the next section we introduce the problem that motivates the work done in this thesis.

1.2 Goal optimization problem

In order to fix ideas consider the following example: imagine we are a petrol company which owns some number of petrol stations and we are paying a transport company to bring our product to the stations with their trucks in order to refuel them (see Figure 1.2).

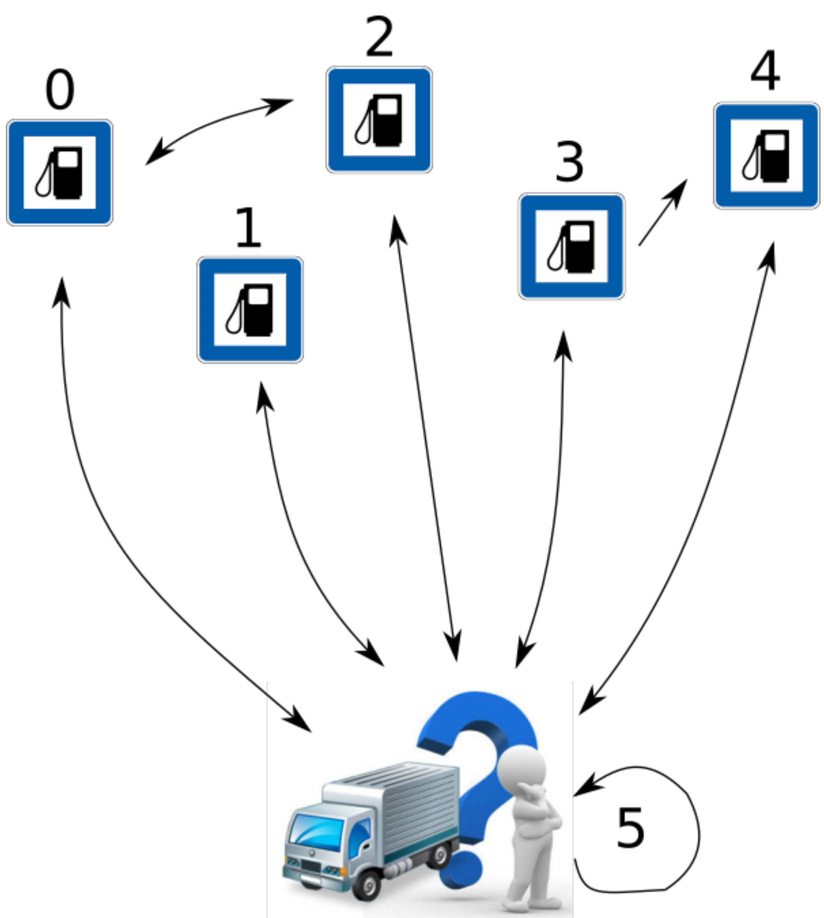


Figure 1.2: Product-delivery problem representation. Truck image taken from [24].

[5]In chapter 2 we formalize the mathematics and concepts of RL.

The goal is to minimize the total amount of money (lets say per month or per year) that the petrol company has to pay to the transport company, ensuring that petrol is always available for costumers (among other constraints).

Note that this problem can be generalized to the situation in which we have a company that is selling some product, and we have to pay to a transport company to distribute it from the loading dock so that it is available in all of our shops at any time. In addition, as we will explain in Chapter 3, there could be not only quantitative things to optimize (such as costs) but also qualitative things such as the "*wellness*" of shops in terms of the quantity of product available in a given time.

1.3 Overview

The thesis is structured in 6 main chapters that we briefly describe now:

- Chapter 2 introduces the basic mathematical formalism for Reinforcement Learning, which are the Markov Decision Processes.
- Chapter 3 develops the mathematical model for product delivery that we are interested in by using the formalism of RL introduced in chapter 2 (assumptions, constraints, etc).
- Chapter 4 introduces Q-learning, one of the many algorithms that exist in classical Reinforcement Learning, and then we apply it to solve a product delivery problem.
- Chapter 5 is focused on learning about Deep Neural Networks. The first part is aimed to formalize the mathematics of DNN and explain how they are trained. The second part consists on applying a DNN to a classification problem we have created, which is related to the product delivery problem we want to solve, and which has served as an example to learn DNN in a practical way.
- Finally in Chapter 6 we introduce Policy Gradient, another RL algorithm which in this case we implement using Deep Neural Networks. The goal of applying Deep Neural Networks instead of classical RL techniques, is that the former is expected to scale to large problem sizes (big number of shops and trucks), whereas the latter scales very poorly and training is very slow. In this chapter we get introduced to the world of Deep Reinforcement Learning.

Chapter 2

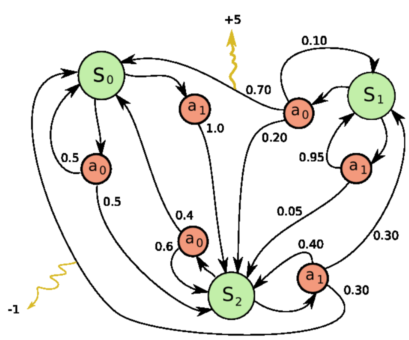
Markov Decision Processes and Reinforcement Learning

Markov Decision Processes (MDP) [25] are the fundamental mathematical formalism for *decision-theoretic planning* (DTP) [26], reinforcement learning (RL) [27, 28, 29] and other learning problems in stochastic domains [30].

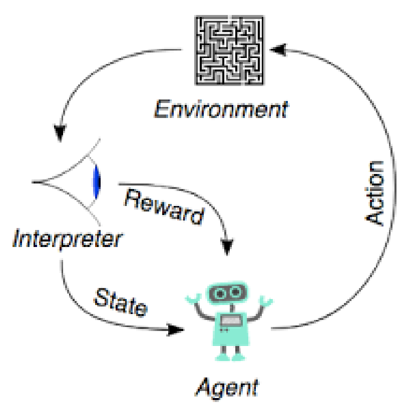
In this chapter we introduce the basic concepts about Markov Decision Processes and Reinforcement Learning, focusing on the approaches that are adaptable to solve our goal optimization problem of product delivery. We mainly follow the first chapter in [30].

2.1 Mathematical formalism of Markov Decision Processes

In MDP models an *environment* is modelled as a set of *states* and *actions* that can be performed to control the system's state. The goal is to control the system by means of some *policy*, in such a way that some performance criteria is maximized (or minimized). MDP are commonly used to model problems such as stochastic planning problems, learning robot control and game playing.



(a)



(b)

Figure 2.1: (a) *"Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)"* [31]. (b) *"The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent"* [32].

RL is a general class of algorithms whose goal is to make an *agent* learn how to behave in an environment, where the only feedback consists of a scalar *reward* signal [30]. The goal of the agent is to perform actions that maximize rewards in the long run.

In Figure 2.1a we see the directed graph representation of a simple MDP with three states (green nodes) and some actions (orange nodes). In a given state, only some actions can be performed, and each can lead to one or more states according to different probabilities. Transitions between states and the intermediate actions are represented by directed edges. In some state transitions, a positive or negative reward can be obtained (orange arrows) if we think of a MDP in the context of Reinforcement Learning (see Figure 2.1b).

2.1.1 Markov Decision Processes

The definition of MDPs we present at the end of this section consists of states, actions, transitions between states and a reward function. Although general MDPs may have infinite state and action spaces, we focus on discrete MDP problems with finite state and action spaces.

States

We denote $\mathcal{S} = \{s^1, ..., s^N\}$ the set of possible states that can define the environment in a particular instant, and $|\mathcal{S}| = N$. Each $s \in \mathcal{S}$ is going to be considered as a tuple of *features* or properties that uniquely determine the environment state in a certain situation. For instance, in *the tic-tac-toe* game (see Figure 2.2 on the right), a state could be a tuple of 9 components corresponding to each cell of the game's board and it may contain either 0, 1 or 2 (or any other three symbols) depending on if the cell is empty or occupied by one of the players.

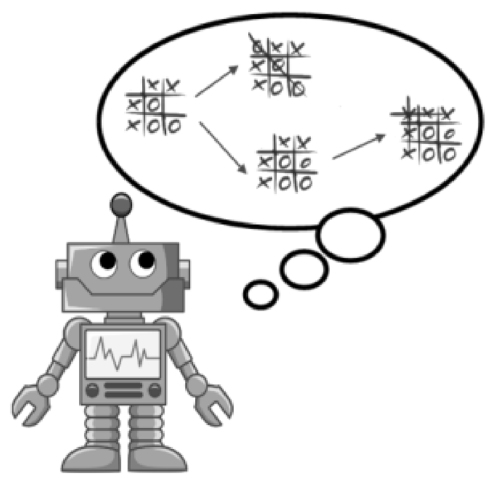


Figure 2.2: Representation of an AI agent for tic-tac-toe game. States and possible actions for a particular case are also shown. Picture from [33].

Actions

We denote $\mathcal{A} = \{a^1, ..., a^\Lambda\}$ the set of actions that can be applied to control the environment by changing its current state; $|\mathcal{A}| = \Lambda$. Usually not all actions are going to be applicable when the system is in a particular state $s \in \mathcal{S}$. Thus, we define $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of actions that are applicable in state s.

The transition operator

If an action $a \in \mathcal{A}$ is applied in a state $s \in \mathcal{S}$, the system makes a transition from s to a new state $s' \in \mathcal{S}$ based on a probability distribution over the set of possible transitions [30]. We define the transition function T as

$$\begin{aligned} T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} &\longrightarrow [0, 1] \\ (s, a, s') &\longmapsto P(s'|s, a) \end{aligned}$$

Hence, $T(s, a, s')$ is the probability of a system in a state s to make a transition to a new state s' after being applied action a.

There are some conditions to be satisfied so that T defines a proper probability distribution over possible next states:

i) For all $s, s' \in \mathcal{S}, a \in \mathcal{A}$, $0 \leq T(s, a, s') \leq 1$,

ii) To model the fact that some actions are not applicable when being in some states, one sets $T(s, a, s') = 0$ for all triples (s, a, s') with $s', s \in \mathcal{S}$ so that $a \notin \mathcal{A}(s)$.

iii) For all $s \in \mathcal{S}, a \in A$, $\sum_{s' \in \mathcal{S}} T(s, a, s') = 1$.

For talking about the *order* in which actions occur one defines a discrete *global clock*, $t \in \{0, 1, 2, ...\}$, so that s_t, a_t denote the state and action at time t, respectively.

Definition 2.1. *The system being controlled is* Markovian *if the result of an action does not depend on the previous actions and visited states, but only depends on the current state, i.e. [30]*

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1}). \tag{2.1}$$

In our context one assumes that the system being controlled is Markovian.

The rewards function

In classical optimization problems one seeks a *cost function* and aims to either maximize or minimize it under some set of conditions. In the context of RL, one talks about *rewards*, and the goal is to make an agent learn how to maximize the rewards obtained. For the type of problems we are interested to solve, we define the reward function R as

$$\begin{aligned} R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} &\longrightarrow \mathbb{R} \\ (s, a, s') &\longmapsto R(s, a, s') \end{aligned}$$

Thus, R is a scalar feedback signal which can be interpreted as a *punishment*, if negative, or a *reward*, if positive.

For the *tic-tac-toe* example, one could assign either high positive or negative rewards, or zero rewards, to actions that (when being in a given state) lead to win or lose the game, or having a draw, respectively. In this situation, the goal of the agent is to reach positive valued states, which means winning the game. Sometimes the problem is more complex and it is common to assign non-zero reward to combinations of states and actions that are good or bad under some criteria. For instance, having two aligned pieces in *tic-tac-toe* is good, so we may assign a positive reward for these situations.

In conclusion, rewards are used to give a *direction* in which way the MDP system should be controlled [30]. More concretely, the system should be controlled by the agent taking actions that lead to more (positive) rewards over time.

The Markov Decision Process

With all this stuff, we can now give the definition of a *Markov decision process.*

Definition 2.2. *A* ***Markov decision process*** *(MDP) is a tuple* $(\mathcal{S}, \mathcal{A}, T, R)$ *in which* $\mathcal{S}$ *is a finite set of states,* $\mathcal{A}$ *a finite set of actions,* T *a transition probability function* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *and* R *a reward function,* $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$*. One says that the pair* T, R *define the* model *of the MDP.*

This definition of MDP allows us to model *episodic tasks* and *continuing tasks*:

- In episodic tasks, there is the notion of *episodes* of some length τ where the goal is to take the agent from some starting state to a *goal* state. In these cases, one can distinguish between *fixed horizon tasks* in which each episode consists of a fixed number of steps, or *indefinite horizon* tasks in which each episode has an end but episodes can have arbitrary length [30]. An example for the former type would be *tic-tac-toe* and for the latter, *chess* board game. Another example for indefinite horizon tasks would be a robot trying to learn how to run or walk; at the beginning it will fall down early (so that episodes will be short), but if it is learning properly, at some point it will be able to run or walk forever.

  In each episode the initial state of the system is initialized with some *initial state distribution* $I : \mathcal{S} \to [0, 1]$.

- In continuing or *infinite horizon* tasks the system does not end unless done it in purpose (theoretically it never ends). A possible example would be a software dedicated to sustain the energy supply for a city by controlling the power plants.

2.1.2 Policies

Given an MDP $(\mathcal{S}, \mathcal{A}, T, R)$, in order to fix ideas we define a deterministic policy by a function $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions that are applicable in that state, i.e., $\pi(s) = a \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$ [1].

A policy π can be used to make evolve, i.e. to control, a MDP system in the following way:

- Starting from a initial state $s_0 \in \mathcal{S}$, the next action the agent will do is taken as $a_0 = \pi(s_0)$.
- After the action is performed by the agent, according to the transition probability function T and the reward function R, a transition is made from s_0 to some state s_1, with probability $T(s_0, a, s_1)$ and an obtained reward $r_0 = R(s_0, a_0, s_1)$.
- By iterating this process, one obtains a sequence $s_0, a_0, r_0, s_1, a_1, r_1, ...$ of state-action-reward triples over $\mathcal{S} \times \mathcal{A} \times \mathbb{R}$ which constitute a trajectory (or path) of the MDP.

In this way, a policy fully determines the behaviour of an agent.

If the task is episodic, the sequence of state-action-reward triples ends in a finite number of iterations τ, the system is restarted and the process starts again with a new sampled initial state. If the task is continuing, the sequence can be extended indefinitely ($\tau = \infty$) [30].

[1] In a general framework, policies are considered to be stochastic functions $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\mathcal{A})$ is the set of probability measures on $\mathcal{A}$.

2.1.3 Formalizing Markov Decision Processes for episodic tasks

So far we have introduced MDPs via a more intuitive but less formal approach than we could have done. Following the usual notation from probability theory, from now we use upper case letters to denote random variables and the associated lower case letters to denote the values taken by these random variables.

First we need to generalize the concept of deterministic policy we considered before to stochastic policies, which are probability distributions $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ over actions given states [2]:

$$\pi(s, a) := \pi(a|s) = P(A_t = a|S_t = s). \tag{2.2}$$

Note that in time t, A_t is a random variable which take values in the action's space $\mathcal{A}$ and S_t is a state random variable that takes values in the states' space $\mathcal{S}$. However, we remark that MDP policies depend only on the current state and not on the past; in other words, policies are *stationary* (time-independent): $A_t \sim \pi(\cdot|S_t)$, $\forall t \geq 0$.

Given an MDP $(\mathcal{S}, \mathcal{A}, T, R)$ to model an episodic task with episodes of length τ and a (stochastic) policy π, we define an episode of this MDP by a sequence of random variables

$$E = \{S_0, A_0, R_0, S_1, A_1, R_1, ..., S_{\tau-1}, A_{\tau-1}, R_{\tau-1}, S_\tau\} \tag{2.3}$$

where R_t is a random variable whose values are the rewards, denoted r_t, obtained after observing S_t, which would take some value s_t, action A_t, which would take some value a_t, and the new state S_{t+1} that would take some value s_{t+1}. In particular, $r_t = R(s_t, a_t, s_{t+1})$, and we may write $R_t = R(S_t, A_t, S_{t+1})$.

2.1.4 Optimality Criteria and Discounting

The core problem of MDPs is to find an optimal policy π^* to control the system "optimally". Therefore, we need a model for optimality with regard to policies.

Although there are several models of optimality for a MDP, we focus here on the so called *discounted average reward* criteria, which is applicable in both finite (episodic) and infinite (continuing) horizon tasks.

The discounted sum of rewards received or *return* by the agent starting from state S_t is defined as

$$R_t^\gamma = \sum_{k=0}^{\tau-t} \gamma^k R_{t+k} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^{\tau-t} R_\tau \tag{2.4}$$

where $\tau < \infty$ if the task is episodic, and $\tau = \infty$ if it is continuing, and $\gamma \in [0, 1]$ is a discount factor for future rewards ($\gamma < 1$ if the task is continuing).

The factor γ determines the importance of future rewards:

- For $\gamma = 0$ (and taking $\gamma^0 = 1$) the only term that survives in equation (2.4) is the one corresponding to the present reward. In this case the agent will be *myopic* (short-sighted) by only considering current rewards.

[2] By convention, being in some state s, if $a \notin \mathcal{A}(s)$, then $\pi(a|s) = 0$.

- On the contrary, for $\gamma \approx 1$ the discount factor will make the agent strive for a long-term high reward [34]. For instance, if $\gamma = 0.95$, rewards 13 steps far into the future count approximately half as much as immediate rewards ($0.95^{13} \approx 0.51$), while for $\gamma = 0.99$, rewards start counting half as much immediate ones from the 69th step ($0.99^{69} \approx 0.499$) [20].

- For episodic tasks and the case $\gamma = 1$, R_t^γ is the sum of rewards at each step of an episode.

The goal of the discounted average reward criteria in the context of MDP is to find a policy π^* that maximizes the expected return $\mathbb{E}_\pi[R_t^\gamma]$. Intuitively this quantity is the total (discounted) reward that one expects to obtain from time t until the end of the episode. The expectation $\mathbb{E}_\pi$ denotes the expected value when the MDP is controlled by policy π.

We say we have *solved* and MDP if we have found some optimal policy. We formalize more this idea in the coming sections, after defining the so-called value functions.

2.1.5 Value Functions, optimal policies and Bellman equations

In order to link the criteria of optimality introduced in the precedent subsection to the policies, one considers the so-called *value functions*, which are a way to quantify "how good" it is for the agent being in a certain state (*state-value function* V^π) or making a certain action when being in some particular state (*action-value function* Q^π).

Value Functions and optimal policies

Definition 2.3. *The* ***value of a state*** s ***under policy*** π*, denoted* $V^\pi(s)$ *is the expected return when starting in state* s *and following* π *thereafter [30]. If we consider the discounted average reward criteria, we have the following expression:*

$$V^\pi(s) = \mathbb{E}_\pi\left[R_t^\gamma | S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t} \gamma^k R_{t+k} | S_t = s\right] \tag{2.5}$$

where $\tau < \infty$ *if the task is episodic, and* $\tau = \infty$ *if it is continuing.*

Definition 2.4. *The* ***state-action value of*** s***,***a ***under policy*** π*, denoted* $Q^\pi(s,a)$ *is the expected return when starting in state* s*, taking action* a *and thereafter following* π *[30]. For the discounted average reward criteria we have the following expression:*

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[R_t^\gamma | S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t} \gamma^k R_{t+k} | S_t = s, A_t = a\right] \tag{2.6}$$

where $\tau < \infty$ *if the task is episodic, and* $\tau = \infty$ *if it is continuing.*

The main goal of a given MDP is to find the policy that receives the most reward. This is equivalent to find the policy that maximizes the value functions for each possible state.

Definition 2.5. *A policy* π^* *is said to be* ***optimal*** *if it is such that* $V^{\pi^*}(s) \geq V^\pi(s)$ *for all* $s \in \mathcal{S}$ *and all policies* π*.* $V^* := V^{\pi^*}$ *is called the optimal value function. One may write*

$$V^*(s) = \max_\pi V^\pi(s). \tag{2.7}$$

Similarly one defines the optimal Q-value, Q^* by

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{2.8}$$

To illustrate how value functions can be used in practice; assume we know the optimal Q-values for each sate-action pair (i.e., $Q^*(s,a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$), or an algorithm able to estimate them. Then, one can greedily select an optimal action using the greedy Q-policy π_Q defined as

$$\pi_Q(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q^*(s,a') \\ 0 & \text{otherwise} \end{cases}, \qquad \forall s \in \mathcal{S}. \tag{2.9}$$

and it is an optimal policy according to definition 2.5. By convention, given $s \in \mathcal{S}$, $Q^*(s,a), Q^\pi(s,a) = -\infty$ if $a \notin \mathcal{A}(s)$.

Bellman equations

Richard Bellman derived the so-called *Bellman equations* [35], which allow us to solve MDPs practically. The Bellman equations are essential in RL and are necessary to understand how many RL algorithms work.

Before deriving these equations, we introduce some useful notation to simplify things:

- We denote $\mathcal{P}^a_{ss'} = T(s,a,s')$ the transition probability of going from state s to s' after taking action a.
- Similarly, $\mathcal{R}^a_{ss'} = \mathbb{E}\left[R_t | S_t = s, A_t = a, S_{t+1} = s'\right]$ denotes the expected reward obtained after transitioning from state s to s' by taking action a. [3]

Theorem 2.1 (Bellman equations)**.** *Given an MDP $(\mathcal{S}, \mathcal{A}, T, R)$ and a policy π we can consider an episode as defined in equation* (2.3)*. Then the value functions V^π and Q^π satisfy the following equations:*

$$V^\pi(s) = \mathbb{E}_\pi\left[R_t + \gamma V^\pi(S_{t+1}) | S_t = s\right] \tag{2.10}$$

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[R_t + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \tag{2.11}$$

for all $s \in \mathcal{S}, a \in \mathcal{A}$.

Proof. We proof only equation (2.10) and (2.11) would be proven similarly.

Given $s \in \mathcal{S}$, from the definition of V^π we may split the term inside the expectation in two terms:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[R^\gamma_t | S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t} \gamma^k R_{t+k} | S_t = s\right] = \mathbb{E}_\pi\left[R_t + \sum_{k=1}^{\tau-t} \gamma^k R_{t+k} | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_t + \gamma \sum_{k=0}^{\tau-t-1} \gamma^k R_{t+k+1} | S_t = s\right] = \mathbb{E}_\pi\left[R_t + \gamma R^\gamma_{t+1} | S_t = s\right]. \end{aligned}$$

[3] In our case this would correspond to the value r_t, but we retain the expectation for a general case where the reward may contain a noise term.

Now, the linearity of the conditional expectation allows us to decompose $V^\pi(s)$ as follows

$$V^\pi(s) = \mathbb{E}_\pi\left[R_t | S_t = s\right] + \mathbb{E}_\pi\left[\gamma R^\gamma_{t+1} | S_t = s\right].$$

Lets see how these two terms look like more explicitly using the notation for $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$.

The first term is simply

$$\mathbb{E}_\pi\left[R_t | S_t = s\right] = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}\mathcal{R}^a_{ss'}. \tag{2.12}$$

For the second term we need to work a bit:

$$\mathbb{E}_\pi\left[\gamma R^\gamma_{t+1} | S_t = s\right] = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}\gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t-1} \gamma^k R_{t+k+1} | S_{t+1} = s'\right] = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}\gamma V^\pi(s'), \tag{2.13}$$

because if we denote $t' = t+1$ and remember the definition of V^π we have that

$$\mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t-1} \gamma^k R_{t+k+1} | S_{t+1} = s'\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\tau-t'} \gamma^k R_{t'+k} | S_{t'} = s'\right] = V^\pi(s').$$

Finally, if we combine equations (2.12) and (2.13) we arrive to

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right) = \mathbb{E}_\pi\left[R_t + \gamma V^\pi(S_{t+1}) | S_t = s\right]. \tag{2.14}$$

□

Note that Bellman equations give us a recursive way to compute value functions, and they are the base to derive algorithms such as the Q-learning algorithm that we will explain in chapter 4. Concretely, from Bellman equations one can obtain the following optimal state-action value equation[4]:

$$Q^*(s,a) = \sum_{s'\in\mathcal{S}} \mathcal{R}^a_{ss'} + \gamma\mathcal{P}^a_{ss'} \max_{a'\in\mathcal{A}} Q^*(s',a'),\ s\in\mathcal{S}, a\in\mathcal{A}. \tag{2.15}$$

2.2 Reinforcement Learning

Once we have defined MDPs, policies, optimality criteria and value functions, the next step is to consider the question of how to compute optimal policies.

Model-free and model-based algorithms

Although there are several approaches, there are usually two distinguished types of algorithms with regards to the available information about the MDP model: *model-based* and *model-free*.

Model-based algorithms assume that a model of the MDP is given, i.e. that a transition and a reward function pair (T, R) is known. This class of algorithms is also known as Dynamic Programming (DP).

[4]See for example [36] or search for the corresponding video lecture in YouTube.

On the contrary, model-free algorithms, which are under the general name of Reinforcement Learning, do not rely on the availability of a perfect model. "Instead, they rely on *interaction* with the environment, i.e. a *simulation* of the policy thereby generating *samples* of state transitions and rewards" [30]. Then, for instance, these samples can be used to estimate the Q-values (or the V-values) for each visited state-action pair $(s, a), s \in \mathcal{S}, a \in \mathcal{A}(s)$, by means of some iterative algorithm. Since a model of the MDP is not known a priori, the agent has to learn how to behave by experience. Thus, the agent has to *explore* the MDP to learn about its environment and how to behave in order to maximize the rewards obtained. "This naturally induces an *exploration-exploitation* trade-off which has to be balanced to obtain an optimal policy" [30].

As we are going to see during the following chapters, for the product delivery problem we introduced in section 1.2, we will have a model for the rewards function but we do not know anything about the transition function of our system modelled as a MDP. For this reason, from now on we assume to be in the model-free, RL, framework where the MDP model (either T, R or both) is initially unknown.

Value and Policy estimation algorithms

So far, for the sake of simplicity we have assumed MDPs with finite state and action spaces. There are some problems where the state or/and the action spaces are continuous so that one can consider that $\mathcal{S} \subseteq \mathbb{R}^d$ and $\mathcal{A} \subseteq \mathbb{R}^m$, for some integers $d, m \geq 1$. In these cases the concepts introduced in section 2.1 are practically the same but with another formalism (see for example Chapter 7 in [30]).

Generally one can distinguish between two RL methods for finding optimal policies: value-approximation and policy-approximation algorithms.

On the one hand, in value-approximation algorithms, experience samples are used to update a value function (such as Q or V) that gives an approximation of the current or the optimal policy, and from these estimations one can define an (approximated) optimal policy. For instance, usually one considers the greedy policy defined in (2.9). The difference between having finite or continuous state-action spaces is that in the first case, value functions are tabular functions[5], and in the continuous case [6] one would consider some parametrized functions Q_θ, V_φ with respect to some set of parameters θ, φ, and from those functions define a policy.

On the other hand, policy-approximation algorithms focus directly on finding optimal policies by improving them as the system interacts with the environment. This approach usually corresponds to the context of continuous state-action spaces where it is considered a parametrized function π_θ, and the goal is to find the set of parameters θ^* such that $\pi^* := \pi_{\theta^*}$ is an optimal policy.

When considering parametrized functions one assumes enough differentiability with respect to the parameters so that one can perform the desired optimization procedures on them.

In this work we are going to focus mainly on two RL approaches:

- In chapter 4, we consider a product delivery framework where states and actions are finite so that we can consider tabular-based value-approximation algorithms such as the so-called

[5] For instance, the Q-value function would be a table with one row per state and one column per possible action. And the V function would just be an array with as many components as states.

[6] Also in the finite case but with very large state and/or action spaces.

Q-learning, the one we will focus on.

- In chapter 6, we consider a more general framework where the state space is continuous and we resort to policy-approximation algorithms where neural networks are used as parametrized policy functions π_θ, where the parameters θ are the weights of the Neural Network (see chapter 5 for more details on Neural Networks). The algorithm considered there is the so called Policy-Gradient, which in general words consists in updating parameters θ taking the direction of the gradient of π_θ scaled in proportion to the discounted average reward obtained (in this way, one achieves to make better states more probable, and worse states less probable).

Chapter 3

Problem approach

In this chapter we present the model for a product delivery system according to the theory of MDPs and Reinforcement Learning explained in chapter 2.

Since this thesis has been motivated by a real world problem, our approach is based on several assumptions and concepts that are particularized to that concrete real problem. However, the ideas we consider could apply more generally to any product delivery problem just changing constraints and some assumptions.

3.1 Problem description and constraints

A single product is to be delivered to several shops from a depot using several trucks (see Figure 3.1). We call *unload* the fact that a truck delivers (unloads) some quantity of product to a shop; unloads can be either *simple*, if a truck only delivers to one shop and then returns to the depot, or *shared* if a truck delivers to more than one shop before returning to the depot. Trucks can only be loaded in the depot.

Moreover, some initial restrictions and assumptions are considered:

1. Trucks leave the depot fully loaded (lets say at 8 in the morning) and return completely empty (lets say at the end of the day, i.e., 23:59:59).

2. Trucks perform at most one *unload* (simple or shared) every day. Hence, it is possible that some day some of the trucks do not go to any shop.

3. A truck can not visit the same shop to perform a *unload* more than one time during the same day.

4. Trucks can perform shared unloads of at most $V \geq 1$ shops. In general V may depend on the truck.

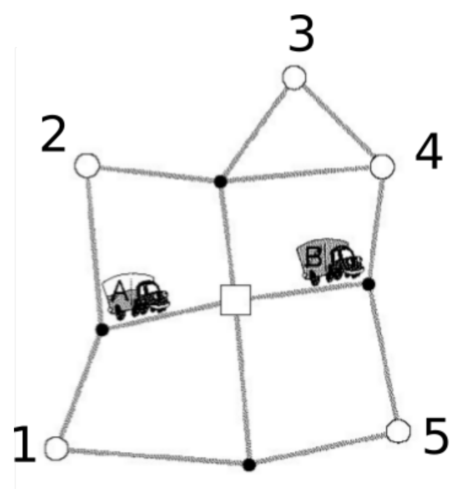


Figure 3.1: Product-delivery domain: the square in the centre is the depot (load doc) and white circles are the shops.. Adapted from [12].

5. The price that the company owning the shops has to pay to the transport company is given by some cost function J_{costs} which may depend on the distance travelled by the trucks from the depot to the corresponding shops, the quantity of product delivered and other contributions imposed by the transport company (for example, impose an additional cost if that day is a holiday, an additional cost depending on the number of shops visited for *shared* unloads, etc.).

Another assumption in our model is that the company that owns the shops has some criteria to decide if the quantity of stock available in a shop is good or bad. For instance, it is clearly bad having zero stock in some shop.

Lets consider a particular shop and imagine that it stores the product in a cilindric container we will refer to as *tank* (see Figure 3.2). There are considered several levels of stock:

1. **Minimum capacity or zero level**: when there is no product in stock at all.

2. **Minimum level**: below this level it is considered as having a break of stock (high risk). Thus, we consider that below this level is like having no product in stock.

3. **Danger level**: from this level to the minimum level it is considered that we are taking a moderate risk, so that the shop should receive a *unload* of product soon. The smaller the value of stock below this level, the higher the risk of having a break of stock.

4. **Maximum level**: the desired maximum stock. This level can be overpassed in some quantity but never surpassing the maximum capacity level. The higher the value of stock above this level, the higher the risk of having a break of stock.

5. **Maximum capacity level**: when it is physically impossible to put more product in the container (i.e., however much we compress the product inside the container, we cannot fit more product).

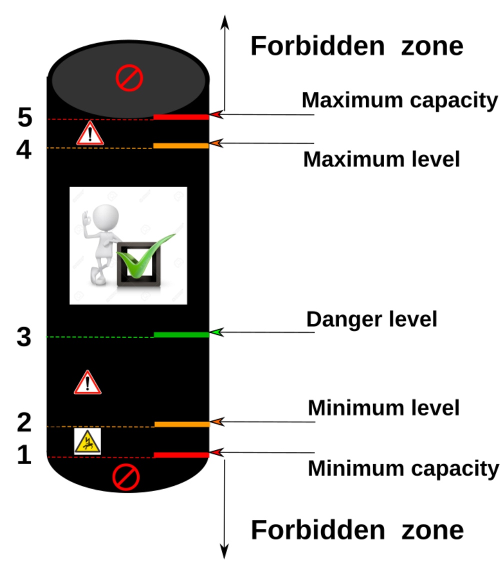


Figure 3.2: Stock levels for an abstract storage container (*tank*) of a given shop.

The ideal and desired state of stock - **optimal zone**- is one between the danger level (3.) and the maximum level (4.). We call the zone between the danger level and the minimum level **danger zone**; the zone between the minimum capacity and the minimum level **bottom risky zone** and the one between the maximum level and the maximum capacity **top risky zone**.

For us, the minimum level and the danger level will correspond to the expected consumption of product in 12 and 36 hours, respectively.

3.2 Mathematical model

To formalize the model we use the Markov Decision Processes (MDP) nomenclature introduced in chapter 2.

Our system of shops and trucks can be modelled as a weighted and directed graph $\mathcal{G}$, a set of Trucks $\mathcal{K}$ and a set of Tanks $\mathcal{N}$ (the abstract storage containers of each shop, as in Figure 3.2). $\mathcal{G}$ is determined by an adjacency matrix $A_\mathcal{G}$ and a matrix of weights $W_\mathcal{G}$, which are defined as follows:

$$(A_\mathcal{G})_{ij} = \begin{cases} 1 & \text{if there is an edge from shop } i \text{ to shop } j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$$(W_\mathcal{G})_{ij} = \begin{cases} w_{ij} \in \mathbb{R}^+ & \text{if there is an edge from shop } i \text{ to shop } j \\ \infty & \text{otherwise} \end{cases} \tag{3.2}$$

Hence, weights are used to quantify the cost of going from a shop to another, and this cost is infinity when it is not possible to go from some shop to some other. Note that a component of $W_\mathcal{G}$ is ∞ if and only if the corresponding component of matrix $A_\mathcal{G}$ is zero.

As an example, the following matrix is the adjacency matrix of the system of 5 shops from Figure 1.2. The depot is considered to be the 6th node.

$$A_\mathcal{G} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

We consider n tanks and k trucks. Thus, $|\mathcal{K}| = k$ and $|\mathcal{N}| = n$.

If $V_{\max}$ is the maximum number of shops that any of the k trucks of the system can visit each day; then, we obtain a sequence of time labels that define a clock for our product delivery system

$$t \in \{t_{0,0}, ..., t_{0,V_{\max}}, ..., t_{\tau-1,0}, ..., t_{\tau-1,V_{\max}}\},$$

such that the first index indicates the day and the second is aimed to divide a day in $V_{\max}+1$ parts, and τ is some "last day" to consider. When the second index is zero, we assume that all trucks are fully loaded in the depot. We assume that it is only at the end of the day when the total amount of product in each tank is reduced by some amount according to what was consumed during all that day.

Each $N_i \in \mathcal{N}$ has the following features:

- An integer id, $i \in \{0, ..., n-1\}$,
- A maximum load (maximum capacity) C_i,
- The current load, $c_i = c_i(t)$,

- A discrete partition of the interval $[0, C_i]$ with d_i parts: $h_\beta = [\beta \cdot C_i/d_i, (\beta+1) \cdot C_i/d_i]$, $\beta = 0, ..., d_i - 1$.

Each $K_i \in \mathcal{K}$ has the following features:

- An integer id, $i \in \{0, ..., k-1\}$,
- A maximum load (maximum capacity) L_i,
- The current load, $l_i = l_i(t)$,
- Its current position: characterized by the tank's id where the truck is located, we call it $p_i = p_i(t)$. If $i = n$, the truck is at the depot.
- A discrete partition of the interval $[0, L_i]$ with m_i parts: $f_\alpha = [\alpha \cdot L_i/m_i, (\alpha+1) \cdot L_i/m_i]$, $\alpha = 0, ..., m_i - 1$.

  This leads to a discrete set of possible "deliveries" that a truck can do when it delivers product to some tank: $\{\lambda_0^{(i)}, ..., \lambda_{m_i-1}^{(i)}\}$, where $\lambda_j^{(i)} = (j+1) \cdot L_i/m_i$, $j = 0, ..., m_i - 1$.

3.2.1 States: simple approach

Following the notation introduced above, we define the state of the system s_t in a given instant t by a tuple of three tuples consisting of the positions $(p_0, ..., p_{k-1})$ of each truck, the current load of each truck $(l_0, ..., l_{n-1})$ and the current load of each tank, $(c_0, ..., c_{n-1})$ [1].

In equation (3.6) we have the real (continuous) states of the system, and in equation (3.4) its discretized version.

Real state of the system

$$s = ((p_0, ..., p_{k-1}), (l_0, ..., l_{k-1}), (c_0, ..., c_{n-1})) \in \mathcal{S} \subseteq \mathbb{N}^k \times \mathbb{R}^k \times \mathbb{R}^n \tag{3.3}$$

Discrete state of the system

$$s = ((p_0, ..., p_{k-1}), (f_{\alpha_0}, ..., f_{\alpha_{k-1}}), (h_{\beta_0}, ..., h_{\beta_{n-1}})) \tag{3.4}$$

with $0 \leq \alpha_0, ..., \alpha_{k-1} < m_i$, $0 \leq \beta_0, ..., \beta_{n-1} < d_i$; and f_{α_j}, h_{β_j} are such that $l_j \in f_{\alpha_j}$ and $c_j \in h_{\alpha_j}$, where j varies between 0 and $k-1$ and between 0 and $n-1$ respectively.

For the discrete case, the total number of possible states is:

$$|\mathcal{S}| = (n+1)^k \times \prod_{i=0}^{k-1} m_i \times \prod_{i=0}^{n-1} d_i \tag{3.5}$$

3.2.2 States: complex approach

Now we consider a more general case where the consumption rates of the tanks are added as state variables, so that they could model the fact that consumption rates are not the same every day and allow them to be the predicted consumptions coming from another model.

[1] For simplicity we omit the dependencies on t.

Thus, the real state of the system in this case would be like

$$s = ((p_0, ..., p_{k-1}), (l_0, ..., l_{k-1}), (c_0, ..., c_{n-1}), (q_0, ..., q_{n-1})), \tag{3.6}$$

where q_i is the consumption of product for the next time (day) and for tank i. Concretely, it is the consumption after transitioning from state s at the end of some day d (corresponding to time $t_{d,V_{\max}}$) to some other state s' (corresponding to time $t_{d+1,0}$) through an action a.

Similarly, we could generalize even more this approach by considering not only the predictions of consumption for the next first day, but also the predictions for the coming $D > 1$ days. Therefore, the tuple vector $(q_0, ..., q_{n-1})$ will become a set of D tuples $(q_0^j, ..., q_{n-1}^j)$, $j = 1, ..., D$, that we can think as a $D \times n$ matrix.

3.2.3 Actions

In a given state, an action consists in deciding the next location, either a shop or the depot, that each truck has to visit, and in the case of being a shop, how much product has to be delivered. Therefore, if p_i' denotes the new location where the i-th truck has to go and $\lambda^{(i)}$ the quantity of product to be unloaded, then

$$a = ((p_0', ..., p_{k-1}'), (\lambda^{(0)} ..., \lambda^{(k-1)})) \in \mathcal{A} \subseteq \mathbb{N}^k \times \mathbb{R}^k \tag{3.7}$$

The dimension of the actions space $\mathcal{A}$ when the truck loads are discretized satisfies:

$$|\mathcal{A}| \leq (n+1)^k \times \prod_{i=0}^{k-1} m_i \tag{3.8}$$

The inequality comes from the fact that not all actions need to be valid actions in a given state.

3.2.4 Rewards function

As we commented in section 2.1.1, the rewards function is a scalar score that allows the agent to know how good or bad it is to take an action in a given state.

In the product delivery problem we want to solve, we split the function of rewards in three main contributions (see eq. (3.15)):

$$R(s, a, s') = \mu_1 J_{\text{costs}} + \mu_2 J_{\text{levels}} + \sum_j \mu_{3,j} J_{\text{extra},j}, \qquad \mu_i \geq 0, \tag{3.9}$$

- The first term, $J_{\text{costs}}(s, a, s')$, would be the one representing the economical costs to pay to the transport company. In particular, $J_{\text{costs}} \leq 0$.
- The second term, $J_{\text{levels}}(s')$, should be a reward function that penalizes negatively in dangerous and forbidden regions of stock for each tank, and positively in the optimal region of stock according to Figure 3.2 and the qualitative importance of the stock levels explained in section 3.1.
- Finally, a third term of extra penalties $J_{\text{extra},j}(s, a, s')$ that the agent may need to learn additional rules or prohibitions (such as forbidden actions).

Since the range of values taken by the different J contributions can be very different, the coefficients μ_i allow to balance the importance of each contribution term and deal with different scales.

Transport and unloads costs contribution

If we assume that costs are proportional to the distance travelled by trucks and to the total amount of product unloaded, we can consider a function of the form

$$J_{\text{costs}}(s, a, s') = C_{\text{costs}} \sum_{i=0}^{k-1} w_{p_i, p'_i} \cdot \lambda^{(i)}, \tag{3.10}$$

for some constant C_{costs} that makes J_{costs} be dimensionless.

Levels of stock contribution

If $x^{(i)}(s')$ denotes the fraction between the current load c_i of tank i with respect to its maximum capacity C_i, then we consider the following decomposition for the levels of stock contribution to the total reward function:

$$J_{\text{levels}}(s') = \sum_{i=0}^{n-1} J^{(i)}_{\text{levels}}(x^{(i)}(s')), \tag{3.11}$$

where

$$J^{(i)}_{\text{levels}}(x) = \begin{cases} P_2 & \text{if } x < a \text{ or } x > f \\ C_{ab} \exp\left(\frac{\alpha_{ab}}{x}\right) & \text{if } a \leq x \leq b \\ m_{bc}x + n_{bc} & \text{if } b < x \leq c \\ m_{cd}x + n_{cd} & \text{if } c < x \leq d \\ m_{de}x + n_{de} & \text{if } d < x \leq e \\ C_{ef} \exp\left(\frac{\alpha_{ef}}{1-x}\right) & \text{if } e < x \leq f, \end{cases}$$

and the coefficients m_{ij}, n_{ij} and C_{ij} are determined such that the functions $J^{(i)}_{\text{levels}}(x)$ look like the one in Figure 3.3. As it is detailed in Appendix A, for the suggested function there are 6 free tunable parameters: b, c, e and M, P_1, P_2. In order to simplify notation, we have omitted the dependency of a, b, c, d, e, f on the shop identifier "i".

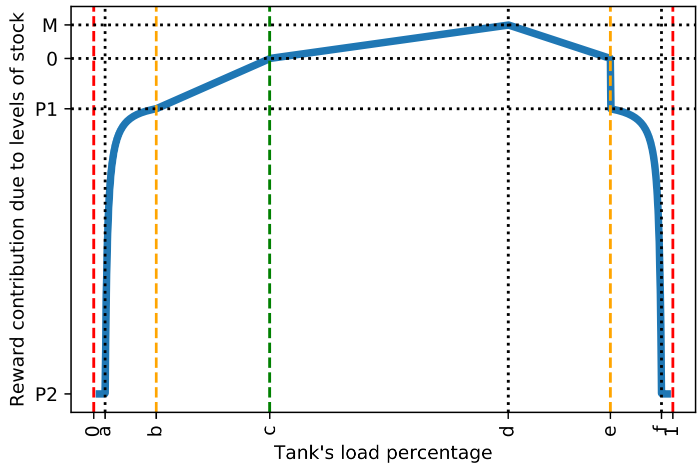


Figure 3.3: Representation of $J^{(i)}_{\text{levels}}$. It is a piecewise function dependent on some parameters P_1, P_2, M, a, b, c, d, e, f that are described in Appendix A.

Additional contributions

Some examples for additional contributions to the rewards function could be a $J_{\text{extra, costs}}$ term that could be included in the J_{costs} term to account for extra costs due to different prices for holidays, for the fact that shared unloads are cheaper or more expensive than simple unloads, etc.

Moreover, since during the simulations the agent does not know which actions are valid or not in a given state so that it may choose non-performable actions, we need to add an extra (negative) penalty to make the agent know that this action is "bad" in some sense.

For instance, the agent may at some point try to fill a tank with a quantity of product that would make the shop surpass its maximum stock capacity. For this reason, we may add a contribution to the reward of the form:

$$J_{\text{extra, forbidden}}(s, a, s') = -C_{\text{extra, forbidden}} \sum_{i=0}^{k-1} \mathbb{1}\,(\text{the action performed by truck } i \text{ is forbidden}) \tag{3.12}$$

3.3 A simplified (toy) model

For the simulations and computations we will present in the next chapters, otherwise noted we are going to restrict ourselves to a simplified version of the model we have introduced so far in this chapter. We are going to focus on the first states approach described in section 3.2.1.

As it can be deduced from Figure 3.4, the first important assumption we make is that the only connections available are the ones between shops and the depot. Thus, each truck can at most visit one shop every day so that we are restricting to the case where $V_{\max} = 1$ (the maximum number of visited shops by a truck).

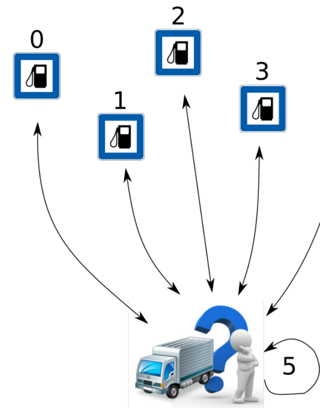


Figure 3.4: Simplified Product-delivery problem representation. Case $n = 5$ shops. Truck image from [24].

In this frame, de adjacency matrix that model our system's graph is the following:

$$A_{\mathcal{G}} = \begin{pmatrix} 0_{n\times n} & 1_n \\ 1_n^T & 1 \end{pmatrix}$$

where $0_{n\times n}$ is a matrix with n rows and n columns of zeros, and 1_n is a column vector with n components equal to 1.

3.3.1 States, actions and rewards

Since at most only one shop is visited by each truck, there is no need to include the current positions of the trucks in the system's state. We know that at the beginning of the day, each truck will visit one shop or stay at the depot and at the end of the day it will return to de depot to by filled up to be ready for the next day.

The first assumption we considered in section 3.1 requires that if a truck visits a shop, the current load of the truck - which under the assumptions we are making corresponds to the whole capacity of the truck- must fit in that shop. For this reason, we should consider a term on the reward function such as the one from equation (3.12) to help the agent avoid sending a truck to a shop that would be overfilled, and other possible forbidden actions. Moreover, now there is no need to include the current load of each truck in the system's state since it will be always the same; the same applies for the actions, where now only the new positions (shop or depot) for each truck are needed. In short,

$$s = (c_0, ..., c_{n-1}) \in \mathcal{S} \subseteq \mathbb{R}^n$$
$$a = (p'_0, ..., p'_{k-1}) \in \mathcal{A} \subseteq \mathbb{N}^k.$$

Moreover, lets assume for simplicity that all shops are discretized in d parts. Then, equations (C.11) and (C.11) for the cardinalities of $\mathcal{S}$ and $\mathcal{A}$ become,

$$|\mathcal{S}| = \prod_{i=0}^{n-1} d_i = d^n \tag{3.13}$$

$$|\mathcal{A}| \leq (n+1)^k \times \prod_{i=0}^{k-1} m_i = (n+1)^k. \tag{3.14}$$

In addition, we will consider simulations of two types:

On the one hand, we will consider cases where the transport and unload costs contributions are neglected (thus $\mu_1 = 0$), and just focus on the contribution made by the levels of stock criteria to see if the reinforcement learning algorithms we will present are able to learn how to maintain the shops alive, or even better, maintain their stocks around the optimal regions (remember Figure 3.2). For these cases, the function of rewards simplifies to

$$R(s, a, s') = \mu_2 J_{\text{levels}}(s') + \mu_{3,\text{forbidden}} J_{\text{extra, forbidden}}(s, a, s'), \tag{3.15}$$

where the two terms are given in equations (3.11) and (3.12), respectively.

On the other hand, we will consider cases where the transport costs make some contribution to the total reward. To do it we will add a term $\mu_1 J_{\text{costs}}$, $\mu_1 \neq 0$, to equation (3.15) and see if the algorithms are able to optimize and balance the trade-off between transport costs and the *wellness* of shops.

Chapter 4

Classical Reinforcement Learning

In this chapter we introduce Q-learning, one of the most popular value-based algorithms aimed to learn the optimal Q-values and from them define an optimal policy. Although Q-learning algorithm is a bit antiquate, it will serve us as an starting point to learn classical reinforcement learning by applying it to our product delivery problem.

4.1 The Q-learning algorithm

In practice, the value functions we introduced in section 2.1.5 are learned through the agent's interaction with the environment. If we focus on the Q-values, the popular algorithms that have been used more are Q-learning [37, 38] and SARSA [1] [39] from the so-called *temporal difference approach* [2], which are typically used when the model of the environment, i.e. the pair (T, R), is unknown.

Assuming we can simulate the system in `n_episodes` episodes of length τ by means of some exploitation-exploration strategy for selecting actions in a given state, this leads to a set of simulated episodes E_j that we can write as

$$E_j = (s_0^j, a_0^j, r_0^j, s_1^j, a_1^j, r_1^j, ..., s_{\tau-1}^j, a_{\tau-1}^j, r_{\tau-1}^j, s_\tau^j)$$

for $j \in \{1, ..., \texttt{n\_episodes}\}$.

The most exploitatory action-selection criteria would be the greedy one, which would consist on using equation (2.9) for the current learned Q-values if the current state is known (and otherwise choose an action randomly). The most exploratory criteria would be the one which selects an action completely randomly in every step.

An interesting compromise between the two action-selection extremes is the ε-greedy policy, we denote π_ε, which is widely used in practice [42], and it is defined as follows:

$$\pi_\varepsilon(s) = \begin{cases} \text{random action from } \mathcal{A}(s) & \text{if } p < \varepsilon \\ \arg\max\limits_{a \in \mathcal{A}(s)} Q(s, a) & \text{otherwise,} \end{cases} \tag{4.1}$$

where $p \in [0, 1]$ is a uniform random number drawn at each time step (of each episode).

[1] Which stands for State-Action-Reward-State-Action)

[2] See for example [30, 20] for an academic point of view, or the original papers [40, 41].

Policy (4.1) executes the greedy policy (2.9) with probability $1-\varepsilon$ and the random policy with probability ε. The use of this combination of policies gives a balance between exploration and exploitation that both guarantees convergence and often good performance [43].

In Algorithm 1 we present the pseudocode for Q-learning which is hopeful self-explanatory. The update rule for the Q-values is derived from the Bellman equations we introduced in section 2.1.5 (see [44] for more details). Concretely, it is derived from the optimality equation for Q we showed in eq. (2.15).

There is only one parameter we have not talked about yet, the learning rate α:

The learning rate or step size determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing, while a factor of 1 makes the agent consider only the most recent information. In fully deterministic environments, a learning rate of $\alpha = 1$ *is optimal. When the problem is stochastic, the algorithm converges under some technical conditions on the learning rate that required it to decrease to zero* (see Theorem 4.1) [34]. In general, the learning rate can be considered a function of the time step, the current state and the current chosen action.

Algorithm 1 Q-learning (train) algorithm

procedure Q-LEARNING(γ, $\alpha(a,s)$, τ , `n_episodes`)
    Q $\leftarrow$ 0 ▷ Initialise $Q(s,a)$ for all $(s,a) \in \mathcal{S}\times\mathcal{A}$.
    for $j \in \{1, ..., \texttt{n\_episodes}\}$ **do**
        $s_0^j \leftarrow$ `Random_choice`$(s \in \mathcal{S})$ ▷ The system is initialized to some initial state randomly
        for $t \in \{0, ..., \tau-1\}$ **do**
            Choose $a_t^j \in \mathcal{A}(s_t^j)$ based on the current Q and an exploration strategy (e.g. π_ε).
            Perform action a_t^j.
            Observe the new state s_{t+1}^j and the received reward r_t^j.
            Update Q with the following rule:

$$Q(s_t^j, a_t^j) \leftarrow Q(s_t^j, a_t^j) + \alpha_t^j(s_t^j, a_t^j)\left[r_t^j + \gamma \max_{a\in\mathcal{A}(s_{t+1}^j)} Q(s_{t+1}^j, a) - Q(s_t^j, a_t^j)\right]$$

        end for
    end for
end procedure

For when the problem we are dealing with is not completely deterministic, the following theorem gives us a criteria to ensure that the Q-learning algorithm will converge as long as the learning rate satisfies some conditions.

Theorem 4.1. *Given a finite MDP* $(\mathcal{S}, \mathcal{A}, T, R)$ *such that* $\mathcal{S}$, $\mathcal{A}$ *are finite,* $\gamma \in (0,1)$ *and* R *is deterministic and bounded, the Q-learning algorithm given by the update rule*

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t)\left[r_t + \gamma \max_{a\in\mathcal{A}(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{4.2}$$

converges with probability 1 to the optimal Q-value function as long as

$$\sum_{t=0}^{\infty} \alpha_t(s,a) = +\infty, \quad \sum_{t=0}^{\infty} \alpha_t(s,a)^2 < +\infty, \quad 0 \le \alpha_t(s,a) < 1 \qquad \textit{for all } (s,a) \in \mathcal{S}\times\mathcal{A} \tag{4.3}$$

Condition (4.3) *requires that all state-action pairs are visited infinitely often.*

Proof. The proof of this theorem can be found for example in [44, 45]. □

Example 4.1. The conditions for the learning rates that appear in Theorem 4.1; for the case of an episodic task they can be simply rewritten as

$$\sum_{j=0}^{\infty}\sum_{t=0}^{\tau-1}\alpha_t^j(s,a)=+\infty, \quad \sum_{j=0}^{\infty}\sum_{t=0}^{\tau-1}\alpha_t^j(s,a)^2<+\infty, \quad 0\le\alpha_t^j(s,a)<1 \qquad \textit{for all } (s,a)\in\mathcal{S}\times\mathcal{A} \tag{4.4}$$

where the index k denotes the episode.

As an example, consider a learning rate that decreases with the number of episodes but does not depend on t (when fixing an episode), neither on the state nor the action. For example, take $\alpha_t^j=\frac{\alpha_0}{1+\beta j}$, where $\alpha_0,\beta\in(0,1)$. We call α_0 the initial learning rate, and β the learning rate decay.

Note that α_t^j satisfies the conditions in (4.4) since:

$$\sum_{j=0}^{\infty}\sum_{t=0}^{\tau-1}\frac{\alpha_0}{1+\beta j}\sim\sum_{j=0}^{\infty}\frac{1}{1+j}\sim\sum_{n=1}^{\infty}\frac{1}{n}=+\infty,$$

and

$$\sum_{j=0}^{\infty}\sum_{t=0}^{\tau-1}\left(\frac{\alpha_0}{1+\beta j}\right)^2\sim\sum_{j=0}^{\infty}\frac{1}{(1+j)^2}\sim\sum_{n=1}^{\infty}\frac{1}{n^2}<+\infty.$$

4.2 Simulations

We have implemented the Q-learning algorithm 1 in Python [13], which has been tested for the simplified case considered in section 3.3. To make the algorithm computationally feasible we should resort to a discretization of the states and actions as we explained in section 3.2, since the tabular function $Q(s,a)$ that we encode as a python's dictionary structure [3] should not be arbitrarily large; otherwise, the Q-learning algorithm will not perform well in a reasonable amount of execution time.

At the beginning of each episode the state of the system, i.e. the initial stocks, are sampled via $c_i\sim\text{Uniform}(C_i/d_i,\frac{d_i-1}{d_i}C_i)$ for $i=0,1,...,n-1$.

As exploration strategy we use π_ε as defined in equation (4.1) with a parameter ε that starts from ε_0 and decreases until a value ε_{min} after each episode E_j as follows:

$$\varepsilon(E_j)=\max\left(\varepsilon_{\text{min}}\,,\,\frac{\varepsilon_0}{1+\varepsilon_{\text{decay}}(j-1)}\right),j\ge1, \tag{4.5}$$

where $\varepsilon_{\text{decay}}$ is a decay parameter that determines the rate at which ε decreases as the number of episodes increases.

The π_ϵ strategy taking the ε parameter according to equation (4.5) allows to be very exploratory at the beginning, and start being more and more exploitatory as the number of episodes increases. Moreover, in order to be always a bit exploratory, we perform a maximum operation to cut ε to a minimum threshold value ε_{min}.

[3] Whose *keys* are the state-action pairs encoded as strings with *values* the corresponding Q-values.

In Table 4.1 we summarize the default parameter values that will be used in the simulations of this chapter [4].

Table 4.1: Some constant parameters for the Q-learning simulations. The remaining ones can be found in table B.1 from the appendix.

Parameter	Value	Parameter	Value
n	5	d_i	4
k	2	m_i	1
ε_0	1.0	$\varepsilon_{\text{decay}}$	The inverse of some fraction of `n_episodes` (e.g. $(0.05 \cdot \texttt{n\_episodes})^{-1}$)
ε_{min}	0.05	γ	0.9
α_0	1	β	0
τ	30	`n_episodes`	500.000

Note that we restrict to the case where we have 5 shops and 2 trucks; each shop divides its stock capacity in 4 discrete levels, and each truck does it in only 1 level (the truck's capacity). In table B.1 in the appendix, we can find all the parameter values used to define our product delivery system for the case $n = 5$ and $k = 2$.

During the action selection via the policy π_ε we start being purely exploratory ($\varepsilon_0 = 1$) and end being a bit exploratory 5% of the times ($\varepsilon_{\text{min}} = 0.05$) (see Figure 4.1).

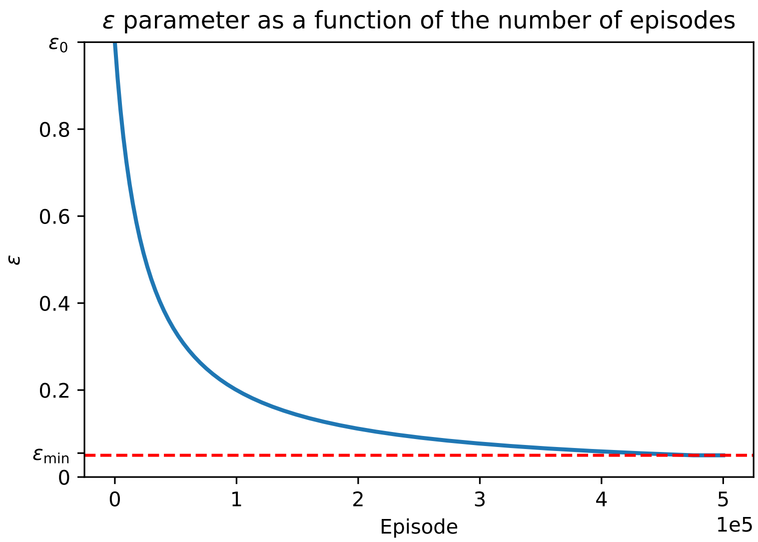


Figure 4.1: Representation of the ε parameter as a function of the number of episodes using the hyperparameter values from Table 4.1.

Although our implementation of Q-learning follows almost literally Algorithm1, there is a subtle detail that we must point out: just after applying the action returned by π_ε and before both "observing" the next state and computing the reward in that step, the current stocks of each shop is reduced in some amount equal to a "daily consumption" that may depend on the shop and also on the day if we consider stochastic (noisy) consumption rates.

[4]The reader may consider to refresh the notation from section 3.2.

When we consider a deterministic system such that all shops have a constant daily consumption rate, it will be given by the expected consumption for the next 24 hours. For us it is approximated by $C_i \frac{b+c}{2}$ (i.e. the mean between the expected consumption in the next 12 and 36 hours), where i denotes the shop.

If we want to add some noise to the consumption of stock in shops, we will consider consumption rates sampled via

$$C_i \frac{b+c}{2} + \delta \cdot \text{Uniform}(-1, 1),\ i \in \{0, ..., n-1\}, \tag{4.6}$$

where typically $\delta \in [0, 1]$. For $\delta = 0$ we are in the deterministic case and for $\delta > 0$ we say that we have a noise of the $\delta \cdot 100$ %.

According to table 4.1 we are taking a constant learning rate equal to 1 for all episodes. Theoretically, according to theorem 4.1, for cases where the system behaves with stochasticity, which would be the case if we add noise to the consumption rates; to ensure convergence to the optimal Q-values we should decrease the learning rate appropriately. In order not to overcomplicate the process of parameter tuning for our Q-learning simulations we are going to use always a constant learning rate equal to 1.

Basically we are going to consider four types of simulations: deterministic or stochastic, depending on if consumption rates have noise or not; and with or without transport costs, which means considering or not the term J_{costs} in the function of rewards.

In table 4.2 we summarize the simulations that we will analyse in the next section.

Table 4.2: Simulation parameters (that differ from the default ones in table B.1)

	Id	Simulation	μ_1	Stochastic consumption
1)	216	Deterministic without transport/unload costs	0	No
2.1)	217	Deterministic with transport/unload costs	10^{-6}	No
2.2)	218	Deterministic with transport/unload costs	10^{-4}	No
2.3)	222	Deterministic with transport/unload costs	10^{-3}	No
3)	224	Stochastic without transport/unload costs	0	Yes, 10%
4.1)	225	Stochastic with transport/unload costs	10^{-6}	Yes, 10%
4.2)	227	Stochastic with transport/unload costs	10^{-4}	Yes, 10%
4.3)	228	Stochastic with transport/unload costs	10^{-3}	Yes, 10%

For all simulations we have arbitrarily chosen an identical initial configuration of the system just to show a single episode to illustrate the results. Moreover, in Appendix C.1 we have one table for each simulation containing information of many parameters averaged over 100 episodes to support the explanations of the results of a single episode.

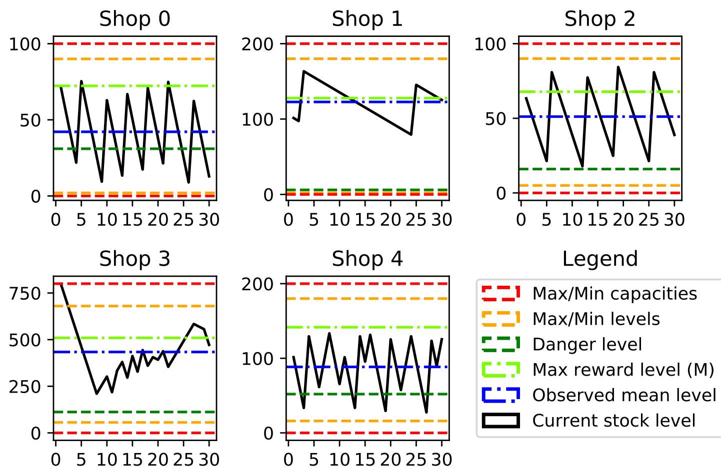
In the plots where we show the stocks of each shop during a simulated episode, we have included dashed lines to indicate the different levels of stock (minimum level and capacity, danger level and maximum level and capacity) for each shop. Moreover, we have added a dashed line of the average stock level (over the episode) of each shop and a dashed line to indicate the stock where the maximum reward M is achieved (remember Figure 3.3).

4.3 Results

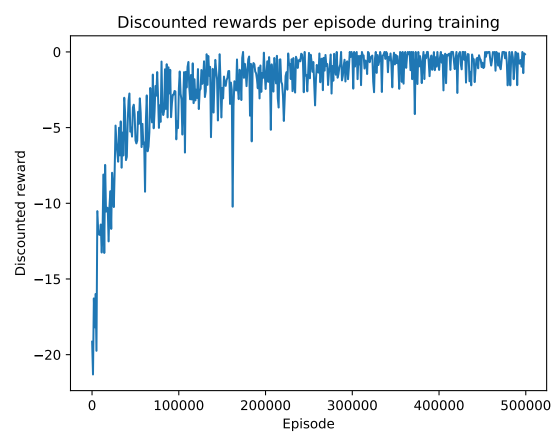
Deterministic simulations

Simulation 1 v.s. Simulation 2.1 (deterministic without and with transport costs)

In figures 4.2 and 4.3 we show the evolution of the levels of stock of each shop and the discounted rewards per episode during training for two deterministic simulations. The former without transport costs contribution ($\mu_{\text{transport}} = 0$) and the latter with some transport contribution ($\mu_{\text{transport}} = 10^{-6}$). Similarly, in tables 4.3 and 4.4 there is summarized some relevant information about the simulated episode of these two simulations.

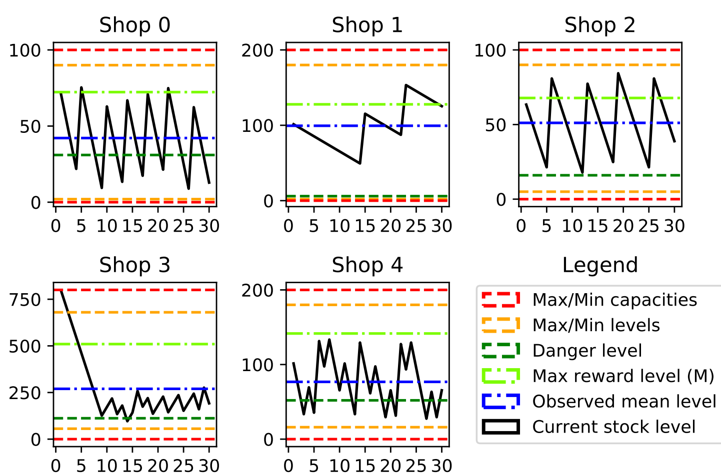


(a)

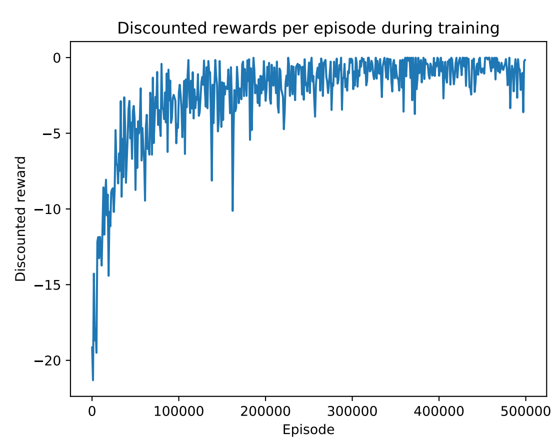


(b)

Figure 4.2: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic without transport costs, $\mu_{\text{transport}} = 0$). Transport costs: 29407. Trucks not delivering: 0. Total trucks sent: 43. (b) Discounted average rewards as a function of the number of training episodes.



(a)



(b)

Figure 4.3: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic with transport costs, $\mu_{\text{transport}} = 10^{-6}$). Transport costs: 27717. Trucks not delivering: 0. Total trucks sent: 40. (b) Discounted average rewards as a function of the number of training episodes.

The first observation we may note for comparing both simulations is that the behaviour of Q-learning in shops 0, 1 and 2 is the same in what refers to the number of trucks of each type sent there. However, note that the two small trucks that are sent to shop 1 are not sent in the same days for both simulations.

Without transport costs there are sent a total of 26 small trucks and 17 big trucks; whereas with transport costs there are only sent 23 small trucks and still 17 big. Since the distance from the depot to shops 3 and 4 is the same (see table B.1), it is clear in this case that the difference in transport costs for both simulations is given by the fact that in the second simulation there are sent 3 small trucks less. Hence, transport costs are reduced from 29407 to 27717. As expected, when considering some weight for the contribution of transport costs to the total reward function, the Q-learning algorithm learns to control the system in such a way that the final transport costs are reduced.

With regards to the *wellness* of shops, we see that in both cases most of the time shops are in the optimal zone. Note that when considering transport costs these costs are reduced from one simulation to the other, but the price to pay is a bit of wellness in shops 3 and 4. Concretely, in average, shops are the 88.67% of the time in the optimal zone without transport costs, and reduces to the 86% when considering transport costs.

Table 4.3: Simulation 1) Information for the simulated episode shown in Figure 4.2. Transport costs: 29407. Trucks not delivering: 0. Total trucks sent: 43.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	2	4	6	7	4	26	—
Big trucks sent	0	0	0	13	4	13	17	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	0	0	—	0	0%
Days in region $(b, c]$	11	0	0	0	4	—	15	10%
Days in region $(c, e]$	19	30	30	20	26	—	133	88.67%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

Table 4.4: Simulation 2.1) Information for the simulated episode shown in Figure 4.3. Transport costs: 27717. Trucks not delivering: 0. Total trucks sent: 40.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	2	4	2	8	7	23	—
Big trucks sent	0	0	0	14	3	13	17	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	0	0	—	0	0%
Days in region $(b, c]$	11	0	0	1	7	—	19	12.67%
Days in region $(c, e]$	19	30	30	27	23	—	129	86%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

We remark that in both simulations there is no truck going to a shop where it can not deliver all the product that is carrying. This would show that the “extra” cost contribution we added to the total reward function is avoiding that to happen (remember Section 3.2.4). We say that the *trucks not delivering* is zero.

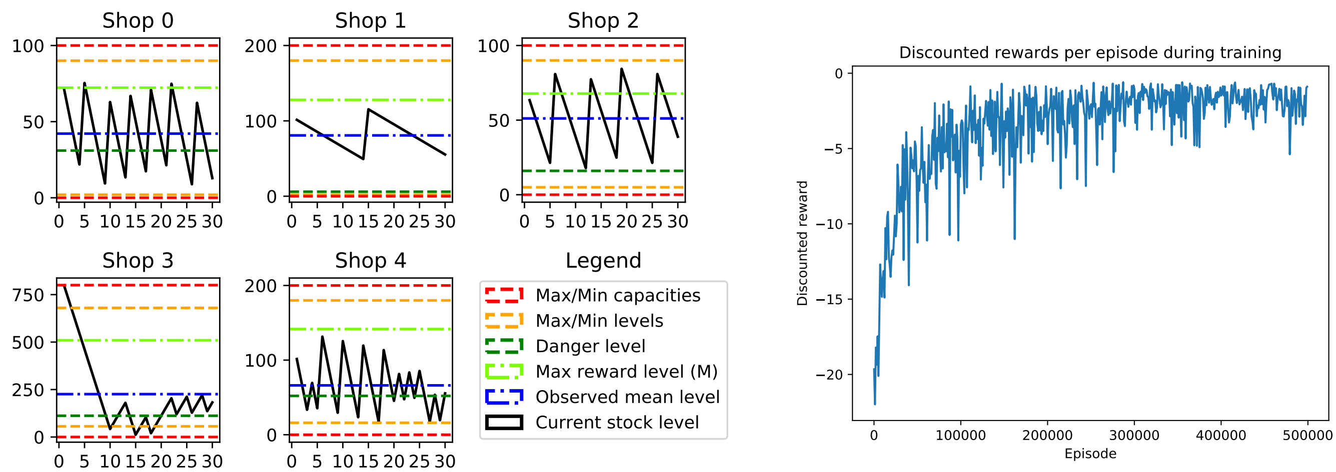
One may see that the average quantities over 100 episodes shown in the tables from the Appendix C.1 behave, in average, as we have just seen for a single simulated episode.

Finally we note that the discounted rewards per episode during training tends to converge and stabilize, hopefully, to the optimal solution but with quite high variance as it is typical in RL.

Simulation 2.2 (deterministic, with higher contribution from transport costs than 2.1)

This simulation is the same as 2.1 but now we increase the transport coefficient $\mu_{\text{transport}}$ from 10^{-6} to 10^{-4} to give a higher contribution of transport costs.

We see that there are sent 3 trucks less in total (from 40 to 37), and in consequence the transport costs over the episodes are reduced. However, the wellness of the shops is also reduced bit [5]



(a) (b)

Figure 4.4: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic with transport costs, $\mu_{\text{transport}} = 10^{-4}$). Transport costs: 26982. Trucks not delivering: 0. Total trucks sent: 37. (b) Discounted average rewards as a function of the number of training episodes.

Table 4.5: Simulation 2.2) Information for the simulated episode shown in Figure 4.4. Transport costs: 26982. Trucks not delivering: 0. Total trucks sent: 37.

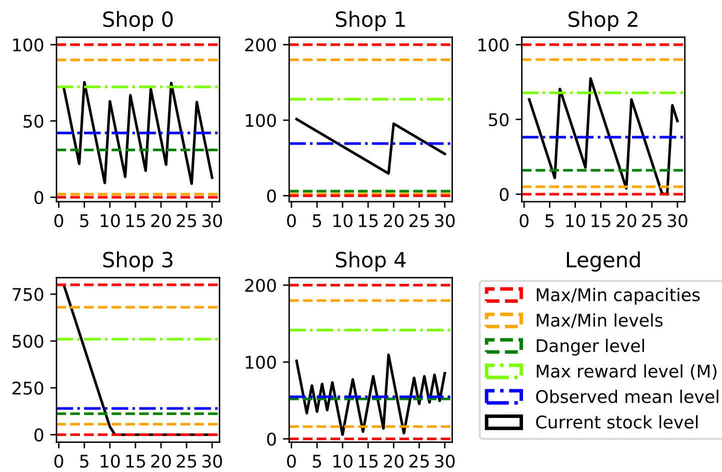
Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	1	4	0	6	12	18	—
Big trucks sent	0	0	0	15	4	11	19	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	3	0	—	3	2%
Days in region $(b, c]$	11	0	0	5	12	—	28	18.67%
Days in region $(c, e]$	19	30	30	20	18	—	117	78%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

[5] Compare the percentages from tables 4.4 and 4.5.

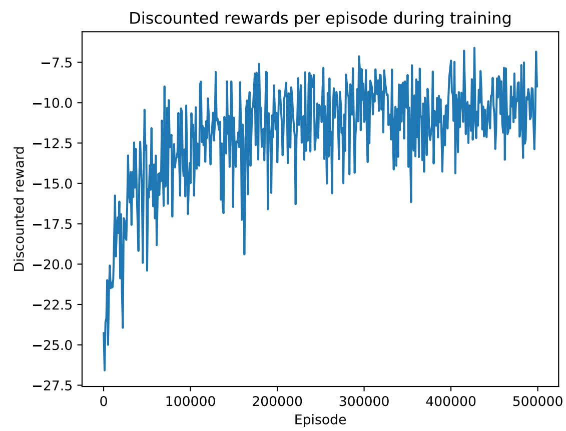
Simulation 2.3 (deterministic, with higher contribution from transport costs than 2.2)

If we now increase $\mu_{\text{transport}}$ in one order of magnitude more, going from $\mu_{\text{transport}} = 10^{-4}$ to $\mu_{\text{transport}} = 10^{-3}$, we see that the contribution of transport costs is now too high to still maintain all shops alive every day. For instance, shop 3 is with zero stock from the 11-th day on, since no truck has gone there during the episode.

Note that the discounted rewards during training are quite more noisy than in the previous simulations where both transport costs and wellness of shops were maintained quite well in all cases.



(a)



(b)

Figure 4.5: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic with transport costs, $\mu_{\text{transport}} = 10^{-3}$). Transport costs: 11530. Trucks not delivering: 0. Total trucks sent: 25. (b) Discounted average rewards as a function of the number of training episodes.

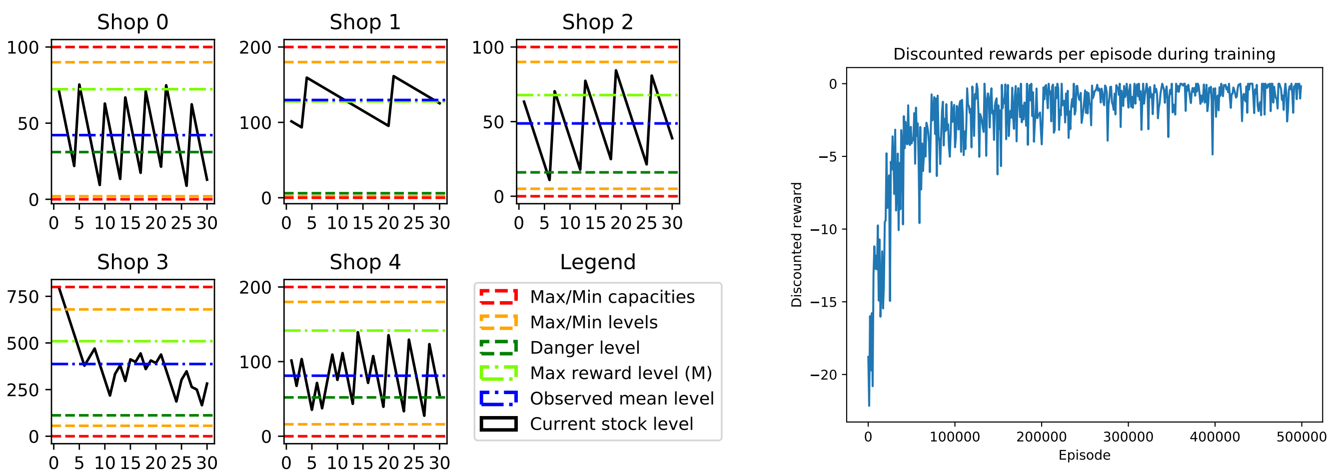
Table 4.6: Simulation 2.3) Information for the simulated episode shown in Figure 4.5. Transport costs: 11530. Trucks not delivering: 0. Total trucks sent: 25.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	1	4	0	12	6	24	—
Big trucks sent	0	0	0	0	1	29	1	—
Days empty	0	0	1	20	0	—	21	14%
Days in region $(0, b]$	0	0	2	1	4	—	7	4.67%
Days in region $(b, c]$	11	0	3	0	13	—	27	18%
Days in region $(c, e]$	19	30	24	7	13	—	93	62%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

Stochastic simulations

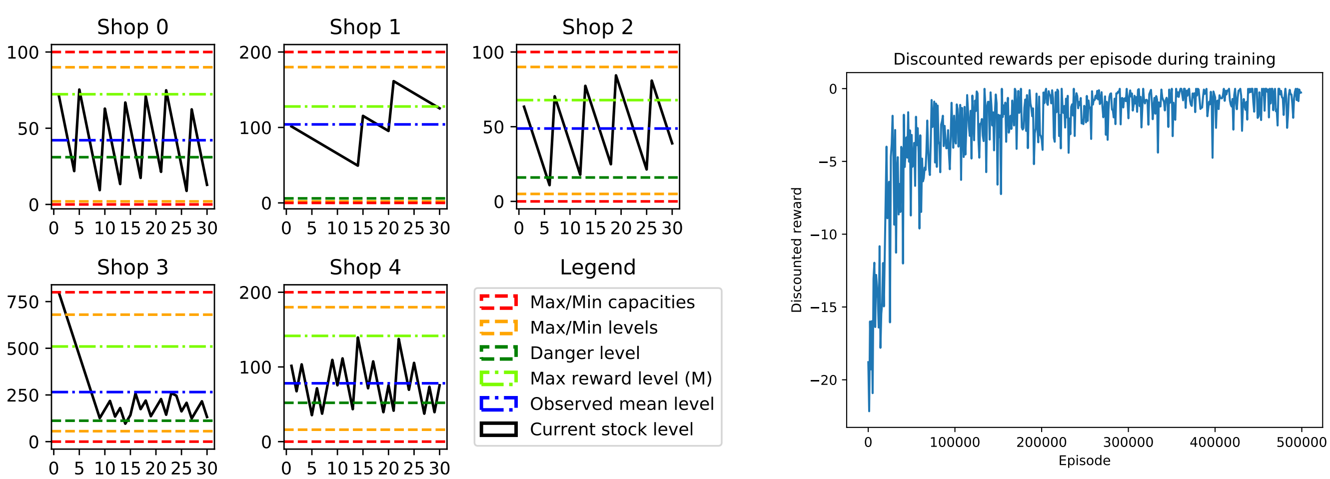
Simulation 3 v.s. Simulation 4.1 (stochastic, 10% noise, without and with transport costs)

In figures 4.6 and 4.7 we show the evolution of the levels of stock of each shop and the discounted rewards per episode during training for two stochastic simulations with a 10% of noise in the consumption rates. The former without transport costs contribution ($\mu_{\text{transport}} = 0$) and the latter with some transport contribution ($\mu_{\text{transport}} = 10^{-6}$). Similarly, in tables 4.7 and 4.8 there is summarized some relevant information about the simulated episode of these two simulations.



(a) (b)

Figure 4.6: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, without transport costs, $\mu_{\text{transport}} = 0$). Transport costs: 28361 Trucks not delivering: 0. Total trucks sent: 42. (b) Discounted average rewards as a function of the number of training episodes.



(a) (b)

Figure 4.7: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-6}$). Transport costs: 27315 Trucks not delivering: 0. Total trucks sent: 41. (b) Discounted average rewards as a function of the number of training episodes.

As it happened in the first two deterministic simulations, we see that the behaviour of Q-learning in shops 0, 1 and 2 is the same with regards to the number of trucks of each type sent to that shops (again, note that the two small trucks that are sent to shop 1 are not sent in the same days for both simulations).

Without transport costs there are sent a total of 26 small trucks and 16 big trucks to shops 3 and 4; whereas with transport costs there are still sent 26 small trucks but now 15 big trucks (one less). Since the distance from the depot to shops 3 and 4 is the same (see table B.1), it is clear in this case that the difference in transport costs for both simulations is given by the fact that in the second simulation there is sent a small truck less. Hence, the total transport costs is reduced from 28361 to 27315.

With these simulations we see that even including a 10% of noise in the consumption rates, when considering some weight for the contribution of transport costs to the total reward function, the Q-learning algorithm learns to control the system in such a way that the final transport costs are reduced while maintaining the wellness of shops properly. Concretely, in average, shops are the 86.67% of the time in the optimal zone without transport costs, and this percentage reduces to the 85.33% when considering transport costs.

Table 4.7: Simulation 3) Information for the simulated episode shown in Figure 4.6. Transport costs: 28361. Trucks not delivering: 0. Total trucks sent: 42.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	2	4	7	6	4	26	—
Big trucks sent	0	0	0	12	4	14	16	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	0	0	—	0	0%
Days in region $(b, c]$	11	0	1	0	6	—	18	12%
Days in region $(c, e]$	19	30	29	28	24	—	130	86.67%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

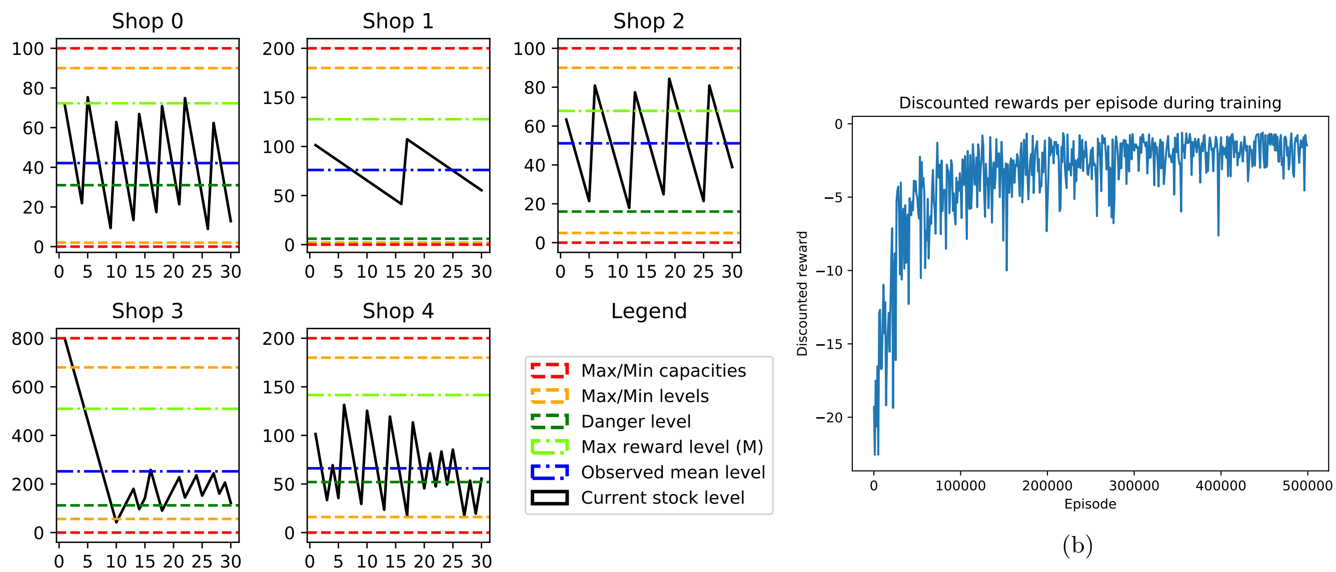
Table 4.8: Simulation 4.1) Information for the simulated episode shown in Figure 4.7. Transport costs: 27315. Trucks not delivering: 0. Total trucks sent: 41.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	2	4	3	10	4	26	—
Big trucks sent	0	0	0	13	2	15	15	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	0	0	—	0	0%
Days in region $(b, c]$	11	0	1	1	7	—	20	13.33%
Days in region $(c, e]$	19	30	29	27	23	—	128	85.33%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

Again, the number of trucks not delivering in both simulations is zero.

Simulation 4.2 (stochastic, 10% noise, with higher contribution from transport costs than 4.1)

Now, as we have done with the deterministic simulations, we can increase more the order of magnitude of the transport coefficient and see if the algorithm is still able to maintain the stocks of the shops properly. As it can be seen from Figure 4.8a and table 4.9, about the 80% of the time, shops are in the optimal level, and almost never in the (top and bottom) risky and forbidden zones. Hence, 10^{-4} is still a good order of magnitude for $\mu_{\text{transport}}$. With respect to the previous simulation where $\mu_{\text{transport}} = 10^{-6}$, transport costs have been reduced from 27315 to 26499.



(b)

(a)

Figure 4.8: (a) Stocks of each shop in a single simulated episode (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-4}$). Transport costs: 26499 Trucks not delivering: 0. Total trucks sent: 37. (b) Discounted average rewards as a function of the number of training episodes.

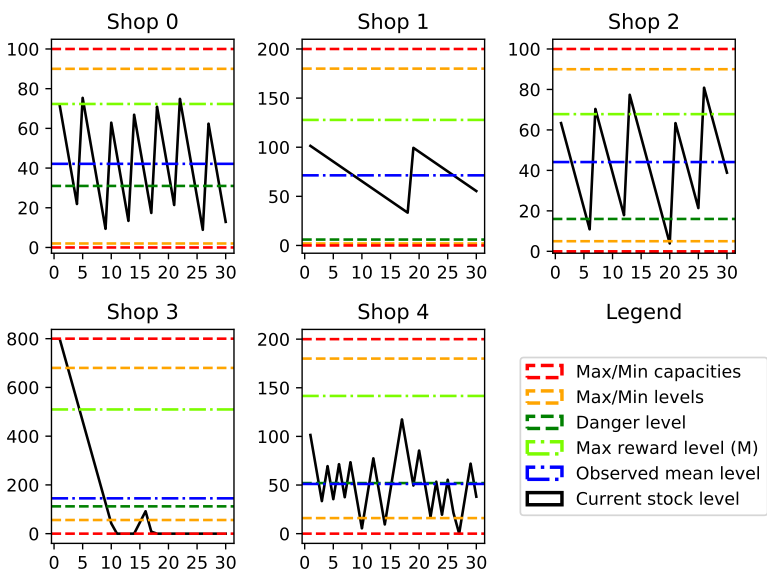
Table 4.9: Simulation 4.2) Information for the simulated episode shown in Figure 4.8. Transport costs: 26499. Trucks not delivering: 0. Total trucks sent: 37.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	1	4	1	6	11	19	—
Big trucks sent	0	0	0	14	4	12	18	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	1	0	—	1	0.67%
Days in region $(b, c]$	11	0	0	3	12	—	26	17.33%
Days in region $(c, e]$	19	30	30	24	18	—	121	80.67%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

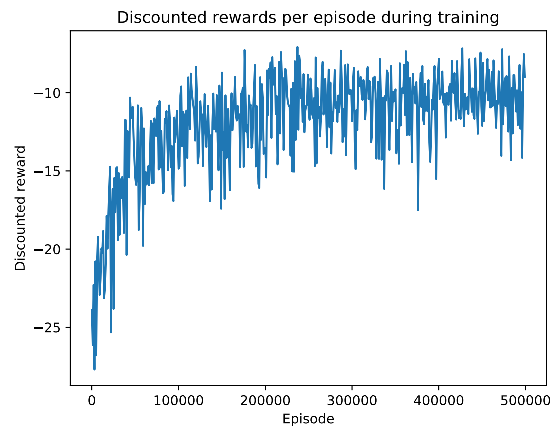
Simulation 4.3 (stochastic, 10% noise, with higher contribution from transport costs than 4.2)

If we now decrease $\mu_{\text{transport}}$ another order of magnitude, as it happened with the deterministic case, the agent is not able to maintain alive Shop 3 although it can still maintain the other four shops quite good.

With the results obtained for $\mu_{\text{transport}} = 10^{-4}$ and $\mu_{\text{transport}} = 10^{-3}$, we may conclude that a $\mu_{\text{transport}}$ of the order of magnitude 10^{-4} would be an optimal choice to have minimal transport costs while maintaining shops alive and within a good *wellness level*. Note that this conclusion would be the same for both the deterministic and the stochastic (with 10% noise) cases.



(a)



(b)

Figure 4.9: (a) Stocks of each shop in a single simulated episode (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-3}$). Transport costs: 13139 Trucks not delivering: 0. Total trucks sent: 27. (b) Discounted average rewards as a function of the number of training episodes.

Table 4.10: Simulation 4.3) Information for the simulated episode shown in Figure 4.9. Transport costs: 13139. Trucks not delivering: 0. Total trucks sent: 27.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	7	1	4	0	13	5	25	—
Big trucks sent	0	0	0	2	0	28	2	—
Days empty	0	0	0	17	1	—	18	12%
Days in region $(0, b]$	0	0	1	3	2	—	6	4%
Days in region $(b, c]$	11	0	2	1	14	—	28	18.67%
Days in region $(c, e]$	19	30	27	7	13	—	96	64%
Days in region $(e, 1]$	0	0	0	2	0	—	2	1.33%

Chapter 5

Neural Networks and Deep Learning

Artificial Neural Networks (ANN)[1] are computational structures inspired by the biological neural networks of the (human) brain with the aim of learning and adapting to a wide variety of tasks. Application areas for NN include system identification and control (vehicle control, trajectory prediction, process control), game-playing and decision making, patter recognition (radar systems, face identification, object recognition), sequence recognition (gesture, speech, hand written and printed text recognition), medical diagnosis, fincance (automated trading systems), data mining, visualization, social network filtering and e-mail spam filtering [46].

In this chapter we introduce the basic concepts about Artificial Neural Networks, more concretely Deep Neural Networks (DNN). We start with an introduction of what is a NN model, and then focus on how to train it. Finally we present an application of DNN to classification, for a toy example related to the product delivery problem we are working with in this thesis.

5.1 Introduction to Artificial Neural Networks

In 1943, Warren McCulloch and Walter Pitts [47] introduced the first mathematical and algorithmic models for NN that paved the way for NN research to split into two approaches: one focused on biological processes in the brain (Neuroscience), and another focused on the application of NN to Artificial Intelligence (e.g. Machine Learning). However, their real powerfulness had not been able to be exploited until the 1990s, thanks to the tremendous increase in computing power, the huge quantity of data available to train NN and the improvement of algorithms.

In this work we introduce NN with a functional point of view, which we think it is more intuitive and understandable. In this way, we consider a NN to be a parametrized function $F_\theta : \mathcal{X} \longrightarrow \mathcal{Y}$ whose inputs x are observations coming from the *train dataset* $X \subseteq \mathcal{X}$ that one wants to use to make the NN learn to predict some output $y \in \mathcal{Y}$. The final goal is to tune the $\theta \subseteq \mathbb{R}^m$ parameters so that F_θ is able to perform the desired task (e.g. a regression task).

Concretely, a NN can be represented as a weighted, directed and acyclic graph (DAG), whose nodes are called *neurons*. This neurons are organized in ordered *layers* in such a way that the only edges that arrive to neurons in the l-th layer are those coming from the $l-1$-th layer. In particular, we focus on fully-connected *feed-forward* NN, where the output from one layer is the input of every

[1]We will refer to them simply as Neural Networks (NN). The “Artificial” term is used to distinguish them from ‘biological” neural networks.

neuron in the next layer and there are no loops (see Figure 5.1). At least it makes sense to consider NN with $L \geq 3$ layers. The first one (left-most) is called *input layer*, the L-th layer (right-most) is the *output layer* and all other layers in the middle are called *hidden layers*. A NN with more than one hidden layer is referred to as Deep Neural Network (DNN).

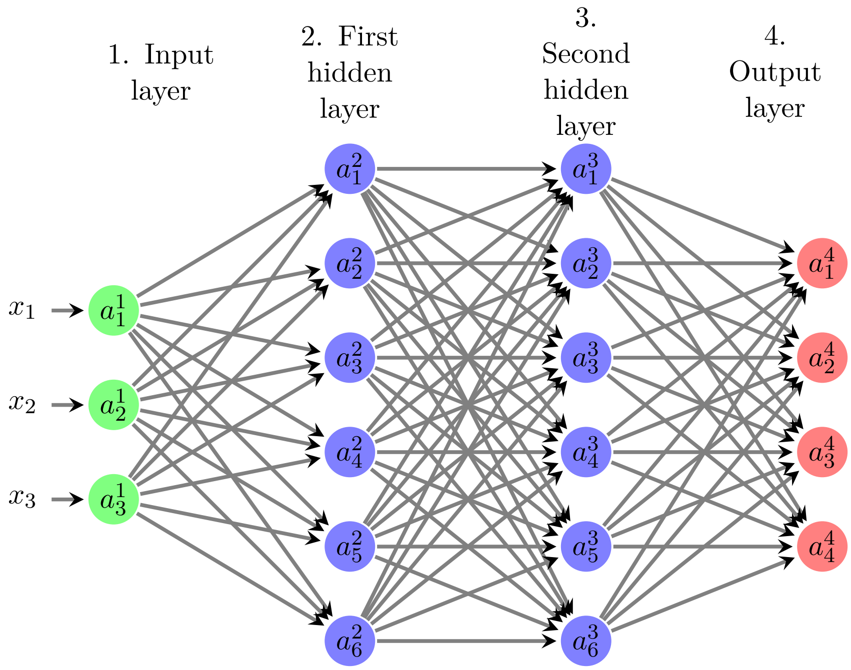


Figure 5.1: Representation of a feed-forward, fully connected Deep Neural Network with two hidden layers of six neurons each, an input layer of three neurons and an output layer with four neurons. With this idea one could generalize to a (feed-forward, fully connected) DNN with arbitrary number of hidden layers and neurons.

In order to define a NN function properly, we should focus on individual neurons first. For the sake of simplicity, lets consider that every layer has the same number of neurons n.

If we consider the j-th neuron in layer l, it receives some input values $a^{l-1} = (a_1^{l-1}, ..., a_n^{l-1})^T$, with associated weights $w_j^l = (w_{j1}^l, ..., w_{jn}^l)$ (w_{jk}^l being the weight of the edge that connects the k-th neuron from the $(l-1)$-th layer to the j-th neuron from the l-th layer), and a constant value b_j^l coming from an auxiliary *bias* node (see Figure 5.2).

As illustrated in Figure 5.2, after a neuron receives the inputs, it computes an intermediate value we call *weighted input*, which is "the weighted sum of inputs plus the bias":

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l = w_j^l \cdot a^{l-1} + b_j^l, \ l = 1, ..., L, \tag{5.1}$$

and then this value is passed to a so-called *activation function* σ [2] to produce a final output value $a_j^l = \sigma(z_j^l)$. For simplicity, we are assuming that each neuron uses the same function σ, but in general we could use a different activation function for each layer or even for each neuron.

In this way the j-th neuron in the l-th layer can be thought as a function f_j^l that takes a^{l-1}, w_{jk}^l (for each neuron k in the previous layer) and b_j^l as input, and outputs a value a_j^l, i.e. $f_j^l(a^{l-1}, w_{jk}^l, b_j^l) = \sigma(z_j^l) = a_j^l$. In this way, it is obtained an overall output value $a^l = (a_1^l, ..., a_n^l)^T$ from the l-th layer, one component per neuron.

[2] Usually the symbol σ is used for the so-called *sigmoid* function. Here it is an arbitrary function.

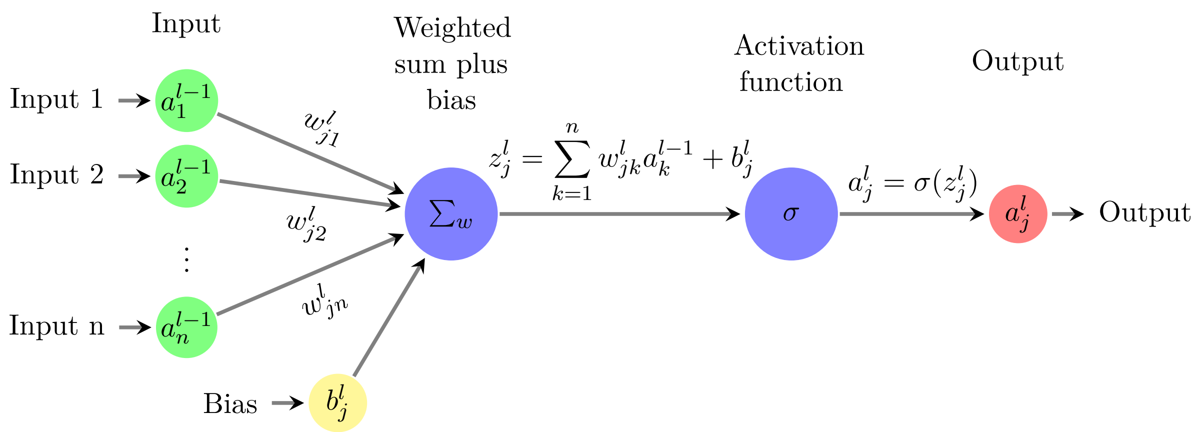


Figure 5.2: Schematic representation of how the inputs of a single neuron are converted to a final output which will go to the next layer of the network to be one of the new inputs in that layer.

Similarly, for each layer we can define a vector of weighted inputs $z^l = (z_1^l, ..., z_n^l)^T$, a vector of biases $b^l = (b_1^l, ..., b_n^l)^T$, and a matrix of weights $w^l = (w_{jk}^l)_{j,k}$, so that with the convention of thinking the activation σ as acting to a vector component by component (vectorized form of σ), we can write the following relation in a compact form:

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l), \qquad l \in \{1, ..., L\}. \tag{5.2}$$

$$a^1 = \sigma(z^1),\ z^1 := x. \tag{5.3}$$

With this notation in mind, we remark that for the input neurons ($l = 1$) there are neither input weights nor bias, so the "weighted input" for the first layer is $z^1 = x$ and the outputs $a^1 = (a_1^1, ..., a_n^1)^T$ are obtained by applying some activation function (usually the identity) to the input values $x = (x_1, ..., x_n)^T$.

To sum up, given a weighted DAG, it leads to a NN function F_θ which depends on linear combinations and compositions of the neural functions f_j^l, and the vector of parameters $\theta = (w_{jk}^l, b_j^l)$ for $l = 1, ..., L$ and j, k varying according to the number of neurons in each of the layers. In particular, F_θ takes $x \in \mathcal{X}$ as input, and outputs $a^L \in \mathcal{Y}$, i.e. $F_\theta(x) = a^L$.

5.2 Training of Deep Neural Networks

In the last section we introduced neural networks as parametrized functions F_θ but we do not know yet how to tune the parameters θ so that the neural network performs the task we want. The process to tune NN parameters is also known as *training*.

Assuming we have a train dataset X and some *loss* function $J_\theta(X) := J(\{F_\theta(x) \,|\, x \in X\})$ that quantifies how well (and how bad) a neural network F_θ is performing according to its learning goal, and also enough regularity on F_θ and J_θ; as we will see more concretely in the coming sections, then the way of tuning the parameters θ is by minimizing J_θ with respect to θ. Note that the J_θ is a function of the whole training set X, and in fact we are saying that it can be written as a function of the outputs $a^L = F_\theta(x)$.

Usual loss functions can be decomposed as a sum of the form,

$$J_\theta(X) = \frac{1}{n} \sum_{x \in X} \tilde{J}_\theta(x). \tag{5.4}$$

where $n = |X|$ is the number of training samples and $\tilde{J}_\theta$ a function that depends on θ and a single input observation x.

5.2.1 Gradient descent

The idea of Gradient Descent (GD) [3] algorithm in the context of optimization[4] problems is to tweak the parameters θ of the function to optimize iteratively, by following the opposite direction of the gradient. Once the gradient is zero, hopefully we would have reached a minimum. GD is a particular case of the so-called descent methods [48] [5].

In equation (5.6) we show the simple expression used to take a GD step for a given training set X,

$$\theta^{k+1} \leftarrow \theta^k - \alpha \nabla_\theta J_\theta(X)\big|_{\theta=\theta^k}, \tag{5.5}$$

where $\alpha > 0$ is the *learning rate* (or *stepsize*) hyperparameter that determines how "big" are the jumps when going from one parameter value to the next. In particular, α indirectly determines the speed of convergence (if any) of the algorithm.

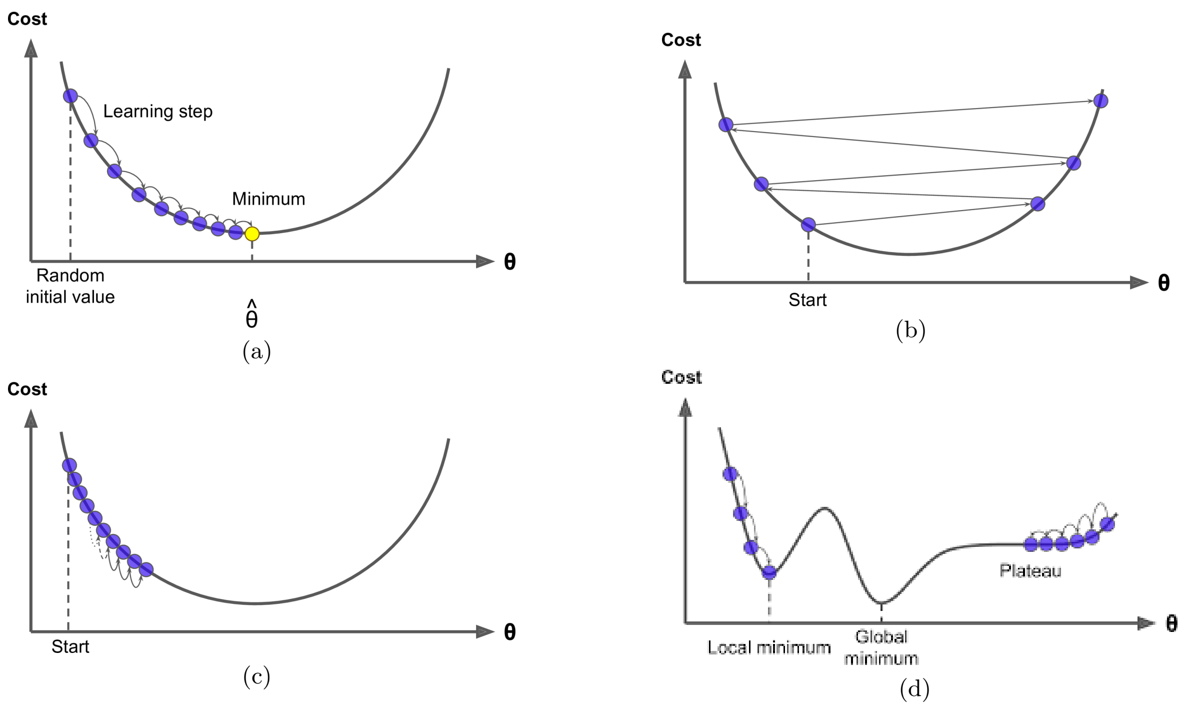


Figure 5.3: Gradient descent performance with (a) appropriate learning rate (b) big learning rate (c) small learning rate. (d) Illustration of possible problems for convergence to a global minimum (the presence of *plateaus* and local minima). Figures taken from [20].

In the pictures from Figure 5.5 we illustrate the role of α assuming that the computed gradients are very close the exact ones (since this would not always be the case for Stochastic GD). On the one hand, very big learning rates could make the GD algorithm diverge (Figure 5.3b). On the other hand, very small learning rates will probably make the algorithm converge, but the number of steps required to do so will probably be very large (Figure 5.3c). Hence, we see that one should try to find

[3] Also known as *steepest descent* method.
[4] With the goal of minimizing some function.
[5] In case we wanted to maximize, we would follow the direction of the gradient, and not the opposite.

some appropriate values for α so that both convergence and relatively fast convergence is achieved (Figure 5.3a).

We should be aware of the fact that GD algorithm for general optimization problems is usually the worst algorithm. As an example, consider the non-convex function depicted in Figure (5.3d) for the case of having a single parameter θ. If the initialized parameter is in the very left, then it will converge to a *local minimum* instead of the *global minimum.* If it starts on the very right, then it will take very long time to surpass the plateau so that if we stop the iterations too early we will never reach the global minimum [20].

However, in the context of NN, loss functions are usually convex, so that in these cases GD is arguably the best algorithm (in terms of speed and performance) to be used assuming we use good approximations for the gradients. Otherwise, we take the risk of finding local minima instead of global, stay in a *plateau* for a long time or even having divergence of the method if the learning rate is not carefully chosen at each step [49, 50].

Finally, it is worth commenting about *feature scaling*, a technique that is commonly used when gradient descent or its variants are used for minimization [6]. To understand why feature scaling usually improves convergence, we refer to Figure 5.4 and quote a very nice answer from the Q&A platform `quora` [7]:

Essentially, scaling the inputs gives the error surface a more spherical shape, where it would otherwise be a very high curvature ellipse. Since gradient descent is curvature-ignorant, having an error surface with high curvature will mean that we take many steps which aren't necessarily in the optimal direction. When we scale the inputs, we reduce the curvature, which makes methods that ignore curvature (like gradient descent) work much better. When the error surface is circular (spherical), the gradient points right at the minimum, so learning is easy.

where "error" refers to the loss function.

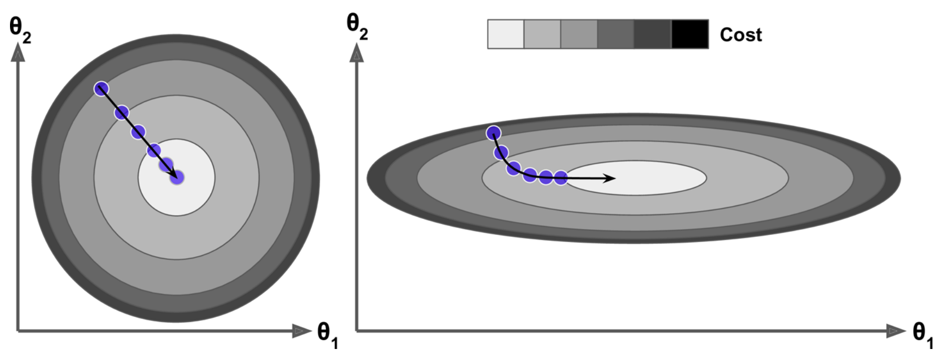


Figure 5.4: Representation of a loss function dependent on two parameters $\theta = (\theta_1, \theta_2)^T$ after feature scaling (on the left) and before feature scaling (on the right). Figures taken from [20].

In short, the idea is that since the learning rate is usually a constant value that multiplies all the components of the gradient vector in every GD step; if the loss function is more elongated in some direction than in others, a learning rate able to improve parameters faster in less elongated directions will not be that faster in more elongated directions.

However, we have to be aware of the fact that feature scaling will not always improve convergence, perhaps in some cases this technique may worsen it.

[6] More usually in the context of classification. A case example would be the one we present in the next section 5.4.
[7] See: https://www.quora.com/Why-does-mean-normalization-help-in-gradient-descent

Mini-batch Gradient Descent

Note that if the cost function decomposes as in (5.4), the GD step can be written as

$$\theta^{k+1} \leftarrow \theta^k - \frac{\alpha}{n} \sum_{x \in X} \nabla_\theta \tilde{J}_\theta(x)\big|_{\theta=\theta^k}. \tag{5.6}$$

In practice, to compute ∇J_θ, we need to compute the gradients $\nabla_\theta \tilde{J}_\theta(x)$ for each training input x and then average them. However, if the number of training inputs is very large, this computations can take a long time and the process of learning becomes very slow.

A way to overcome this computational problem is to consider a random sample of $m < n$ (usually $m \ll n$) observations $x_1, ..., x_m \in X$, we call it a *mini-batch*, and approximate the loss function gradient from (5.6) as

$$\nabla J_\theta = \frac{1}{n} \sum_{x \in X} \nabla_\theta \tilde{J}_\theta(x) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \tilde{J}_\theta(x_i). \tag{5.7}$$

The particular case with $m = 1$ is the so-called Stochastic Gradient Descent (SGD).

As one can intuitively deduce, the *batch-size* hyperparameter m leads to a trade-off between being very fast but perhaps less accurate ($m = 1$) and being very slow but probably very accurate ($m = n$) when computing the estimations of the gradient. Another point of view would be the exploration-exploitation trade-off for small or big values of m, respectively.

Adam optimizer

In this work, when training NN, we will be using one of the most recent and accepted optimization methods by the time of writing this thesis. This method is known as *Adam* (adaptive moment estimation) [51].

Although the majority of methods used in practice, such as *Adam*, technically differ from standard gradient descent; they have the common property of needing the gradients of the loss function with respect to the parameters and are inspired by equation (5.6). In particular, for *Adam* there are computed estimations of the first and second moments (mean and variance) of the gradients and combined together to play the same role as the gradient in equation (5.6).

Moreover, it is an adaptive learning rate algorithm, which means that starting from an initial learning rate α_0 "neither too big nor too small", the algorithm itself is able to tweak the learning rate in every iteration to achieve better convergence and performance.

5.2.2 The Backpropagation algorithm

So far we have seen how to update the parameters $\theta = (w^l_{jk}, b^l_j)_{j,k,l}$[8] of a NN using the gradients of a loss function, but we have not explained how to compute them yet. Note that the GD step from equation (5.5) can be written in terms of w^l_{jk} and b^l_j (instead of the whole parameter vector θ) in

[8] To simplify the notation in the coming explanations, we omit the lower and upper indices and just write w or b to refer to some weight or bias coefficient.

one equation for each parameter,

$$w_{jk}^l \leftarrow w_{jk}^l - \alpha \frac{\partial J_\theta}{\partial w_{jk}^l}, \tag{5.8}$$

$$b_j^l \leftarrow b_j^l - \alpha \frac{\partial J_\theta}{\partial b_j^l}. \tag{5.9}$$

Thus, compute the gradient of the loss function is equivalent to compute all those derivatives with respect to individual parameters.

The most famous and fast algorithm for computing gradients is known as *backpropagation* [52] and is "the workhorse of learning in Neural Networks" [53].

In section 5.2 we indirectly considered two assumptions about the loss J_θ that should be taken so that backpropagation works [53]:

1) First assumption is that J_θ can be decomposed as in equation (5.4). To simplify notation, we denote $\tilde{J}_\theta(x)$ simply by J_x, so that

$$J_\theta \equiv J_\theta(X) = \frac{1}{n} \sum_{x \in X} J_x \tag{5.10}$$

2) The second assumption is that given an input $x \in X$, the loss function J_θ can be written as a function of the outputs $a^L(x) := F_\theta(x)$, i.e.,

$$J_\theta(X) \equiv J(\{a^L(x) \,|\, x \in X\}). \tag{5.11}$$

   We are going to abbreviate $a^L(x)$ simply by a, so that $J_\theta(x) \equiv J(a)$.

The first assumption is made because what is computed in backpropagation are the partial derivatives $\frac{\partial J_x}{\partial w}, \frac{\partial J_x}{\partial b}$ for a single training input $x \in X$, and then the "total" derivatives $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}$ are computed by averaging over all x (due to equation (5.10) and the linearity of the derivatives). From now, to simplify notation we omit the subindex x for the cost function J_x of a single observation.

The second assumption is made so that one can consider the gradients $\nabla_a J$ of J with respect to the output "a"[9] of the neural network and be able to derive the backpropagation equations in a very simple way.

Before going on with backpropagation we need to introduce some definitions: the hadamard product of two vectors (or matrices) and what we will call here the *δ-error* measure.

Definition 1 (Hadamard (or Schur) product)**.** *Let K be a field and E a K-vector space of dimension n. Let $u, v \in E$ and $A, B \in \mathcal{M}_{r\times c}(K)$. The Hadamard product of u and v, and of A and B is just the element-wise product denoted $\circ$ and defined as:*

- $(u \circ v)_j = u_j \cdot v_j,\ j \in \{1, ..., n\}$,
- $(A \circ B)_{ij} = A_{ij} \cdot B_{ij},\ i \in \{1, ..., r\},\ j \in \{1, ..., c\}$,

where $\cdot$ denotes the product in K.

[9] Abbreviation for a^L.

Definition 2 (δ*-error*)**.** *The* δ*-error in layer* l*, we denote by* δ^l*, is the vector whose components are defined as*

$$\delta_j^l = \frac{\partial J}{\partial z_j^l}, \tag{5.12}$$

and we say that δ_j^l *is the* error *of neuron* j *in layer* l*.*

To understand the intuitive meaning of δ_j^l, imagine that for some reason the j-th neuron in layer l "mistakenly" outputs $\sigma(z_j^l + \Delta z_j^l)$ instead of $\sigma(z_j^l)$. Then, the error committed will be given by

$$J(z_j^l + \Delta z_j^l) - J(z_j^l) = \frac{\partial J}{\partial z_j^l}\Delta z_j^l + \mathcal{O}(\Delta z_j^l)^2 = \delta_j^l \Delta z_j^l + \mathcal{O}(\Delta z_j^l)^2 \approx \delta_j^l \Delta z_j^l, \tag{5.13}$$

for Δz_j^l sufficiently small.

Hence, δ_j^l is an amplification factor of how much varies the loss function J given an "error" Δz_j^l on the weighted input of the j-th neuron in layer l.

The backpropagation equations

We are now ready to derive the fundamental equations for the backpropagation algorithm. In Theorem 5.1 we formalize the four equations that are needed, and give a proof of them that only requires basic multi-variable calculus knowledge and the application of the chain rule.

The first equation (BP1) is an expression for the δ-error δ^L in the output layer, which is then used by the second equation (BP2) to recursively compute the δ-errors in the other layers. Finally, the last two equations (BP3) and (BP4) are the ones that allow us to compute the gradients with respect to the parameters of the neural network assuming that the δ-errors given by the other two equations have been computed and the activations (the outputs) in each neuron are available.

Theorem 5.1 (The backpropagation equations)**.** *Consider a loss function* J *satisfying the two assumptions presented at the beginning of this section, and an activation function* σ *that is applied by each neuron of a Neural Network, both differentiable in every point . Then, the following relations are satisfied:*

$$\delta^L = \nabla_a J \circ \sigma'(z^L), \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{BP4}$$

Proof.

(BP1) First of all note that since $\nabla_a J = \left(\frac{\partial J}{\partial a_1^L}, \ldots, \frac{\partial J}{\partial a_n^L}\right)$ by definition, the j-th component of (BP1) is

$$\delta_j^L = \frac{\partial J}{\partial a_j^L}\sigma'(z_j^L). \tag{5.14}$$

If we start from the definition 5.12 of δ_j^l for $l = L$ and apply the chain rule thinking J as a function of $a = a^L$ with components a_k^L that can depend on z_j^L (among other things), then

$$\delta_j^L = \frac{\partial J}{\partial z_j^L} = \sum_{k=1}^{n} \frac{\partial J}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial J}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}, \tag{5.15}$$

where in the last equality we have used the fact that only the activation a_j^L, i.e. the output of the j-th neuron in the L-th layer, depends on z_j^L (the other neurons depend on z_i^L for some $i \neq j$), so that $\frac{\partial a_k^L}{\partial z_j^L} = 0$ for $k \neq j$. Finally, remember that by definition $a_j^L = \sigma(z_j^L)$, so that $\frac{\partial a_j^L}{\partial z_j^L}$ can be written as $\sigma'(z_j^L)$. Thus, from (5.15) we have arrived to equation (5.14). This ends the proof for (BP1).

(BP2) Given that w^{l+1} is the matrix with components w_{jk}^{l+1}, its transpose will have components $(w^{l+1})_{jk}^T = w_{kj}^{l+1}$. Hence, the j-th component of $((w^{l+1})^T \delta^{l+1})$ will be given by the product of the j-th row of $(w^{l+1})^T$ and δ^{l+1}:

$$((w^{l+1})^T \delta^{l+1})_j = \sum_{k=1}^{n} (w^{l+1})_{jk}^T \delta_k^{l+1} = \sum_{k=1}^{n} w_{kj}^{l+1} \delta_k^{l+1}.$$

With this, the j-th component of the equation (BP1) we want to prove is

$$\delta_j^l = \left(\sum_{k=1}^{n} w_{kj}^{l+1} \delta_k^{l+1} \right) \sigma'(z_j^l). \tag{5.16}$$

So it is enough to prove (5.16) for an arbitrary j. With the same idea we used to prove equation (BP1), we use the definition (5.12) of δ_j^l and apply the chain rule but now with respect to z_k^{l+1}:

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} = \sum_{k=1}^{n} \frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_{k=1}^{n} \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{5.17}$$

In order to compute $\frac{\partial z_k^{l+1}}{\partial z_j^l}$, remember that by definition

$$z_k^{l+1} = \sum_i w_{ki}^{l+1} a_i^l + b_k^{l+1} = \sum_i w_{ki}^{l+1} \sigma(z_i^l) + b_k^{l+1}$$

Now if we take derivatives with respect to z_j^l, the bias term cancels out and, in the summation, the only term that survives is that with $i = j$. Thus,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \frac{\partial \sigma(z_j^l)}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_i^l) \tag{5.18}$$

Putting (5.16) in (5.17) we obtain

$$\delta_j^l = \sum_{k=1}^{n} \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l), \tag{5.19}$$

which is exactly (5.16) since $\sigma'(z_j^l)$ does not depend on k and it can be taken out of the summation.

(BP3) Starting from the definition of δ_j^l and applying the chain rule with respect to b_j^l we arrive to

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} = \frac{\partial J}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} \tag{5.20}$$

Now given that $z_j^l = \sum_{k=1}^{n} w_{jk}^l a_k^l + b_j^l$, we also have that $b_j^l = z_j^l - \sum_{k=1}^{n} w_{jk}^l a_k^l$, and taking the derivative with respect to z_j^l we trivially obtain that $\frac{\partial b_j^l}{\partial z_j^l} = 1$. And this proves (BP3).

(BP4) To find the expression for $\frac{\partial J}{\partial w_{jk}^l}$ we apply the chain rule with respect to z_j^l:

$$\frac{\partial J}{\partial w_{jk}^l} = \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l}.$$

Now since $z_j^l = \sum_s w_{js}^l a_s^{l-1} + b_j$, if we derivate with respect to w_{kj}^l, the only term of the sum that survives is that corresponding to $s = k$. Hence,

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1},$$

and this proves that

$$\frac{\partial J}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}.$$

□

To illustrate how the implementation of backpropagation looks like algorithmically, in Algorithm 2 we present a simplified pseudocode.

Algorithm 2 Backward-propagation algorithm [53]

procedure BACKPROPAGATION($x \in X$)
    $a^1 \leftarrow \sigma(x)$ ▷ Set the corresponding activation for the input layer.
    for $l \in \{2, ..., L\}$ **do**
        Compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$ ▷ Feedforward
    end for

    Compute δ^L using (BP1) ▷ Output δ-error

    for $l \in \{L-1, ..., 2\}$ **do**
        Compute δ^l using (BP2) ▷ Backpropagation of the δ-error
    end for

    Use the computed δ-errors and equations (BP3), (BP4) to obtain estimations of the derivatives of J with respect to the parameters of the neural network (weights and biases).

    return: $\frac{\partial J_x}{\partial w^l_{jk}}$, $\frac{\partial J_x}{\partial b^l_j}$ for all l, j, k.
end procedure

Once we know how to compute gradients, we can combine backpropagation with mini-batch gradient descent to train our neural network.

To end up with this section we need to introduce some standard notation in the NN framework. Imagine we have a training set X with n observations and consider mini-batches randomly sampled from X of size $m_B < n$. Assume for simplicity that n is some multiple s of n, so that we can split X in s mini-batches of size m_B. Then, we call a training epoch the fact of applying backpropagation plus mini-batch gradient descent (or any other optimization algorithm, such as *Adam*) to every mini-batch one time. Thus, if the number of epochs in our training algorithm is M, it means that we have used the training set X exactly M times to update the parameters of the NN.

In Algorithm (3) we present pseudocode for training a NN by updating parameters with backpropagation and mini-batch gradient descent. Note that we could use another algorithm such as *Adam* instead of mini-batch gradient descent, which also uses mini-batches to estimate gradients, but the parameter's update rules are slightly different and more sophisticated.

It is very important to notice that the loop over minibatches (Que loop es la clave para paralelizar? ahora no lo veo claro)

Que loop es la clav alelizar? ahora no

Algorithm 3 NN training.

procedure TRAIN(A training dataset X, M, m_B, s)
  for epoch $\in \{1, ..., M\}$ **do** ▷ Begin loop over epochs
    Split X in s mini-batches $X_1, ..., X_s$ of size m_B randomly.
    for $i \in \{1, ..., s\}$ **do** ▷ Begin loop over mini-batches X_i
      for $x \in X_i$ **do**
        $\frac{\partial J_x}{\partial w^l_{jk}}, \frac{\partial J_x}{\partial b^l_j} \leftarrow$ BACKPROPAGATION(x) ▷ Backpropagation
      end for

      for $l \in \{L, L-1, ..., 2\}$ **do**
        Update the NN parameters as ▷ Mini-batch Gradient Descent step

$$w^l_{jk} \leftarrow w^l_{jk} - \frac{\alpha}{m_B} \sum_{x \in X_i} \frac{\partial J_x}{\partial w^l_{jk}}, \tag{5.21}$$

$$b^l_j \leftarrow b^l_j - \frac{\alpha}{m_B} \sum_{x \in X_i} \frac{\partial J_x}{\partial b^l_j}. \tag{5.22}$$

      end for

    end for ▷ End loop over mini-batches X_i
  end for ▷ End loop over epochs
end procedure

5.3 Improving the way neural networks learn

In the context of training NN there are two main problems that one has to deal with in order to obtain a NN model with good performance (e.g. the accuracy of the predictions or desired outputs):

- The first is known as the vanishing or exploding gradients problem.
- The second, which is more general in the context of Machine Learning, is known as *overfitting*.

Vanishing gradients problem appears when in some layers the derivatives with respect to some parameters become very small so that the updates of the parameters given by the equations (5.21), (5.22) in algorithm 3 - for instance - become insignificant (we say that the parameters *learn slowly*). Similarly, *exploding gradients* appear when the derivatives become very big in value so that the training of the parameters usually becomes unstable. This processes of the gradients becoming small or big, is propagated and accentuated more and more as backpropagation goes from the output layer to lower layers.

On the other hand, we can think of *overfitting* as the fact that the NN have memorized all the train dataset, in the sense that it outputs correctly the desired values for every train observation in that dataset. However, generally it would imply that the weights and biases have been adapted to the train dataset so much that when trying to use the NN with new observations[10], the performance would be reduced considerably. Thus, it is preferred to avoid overfitting in the sense of training the

[10] Observations/dataset points not used during training.

NN in a way that it is capable of learning from the train dataset but also generalize properly to data that has never seen before.

In this section we present some techniques and options that can help to avoid the vanishing (or exploiting) gradients and overfitting, as well as improve the speed of convergence during training. We follow mainly [20].

5.3.1 Activation functions

One of the main reasons for vanishing gradient problems can be attributed to activation functions.

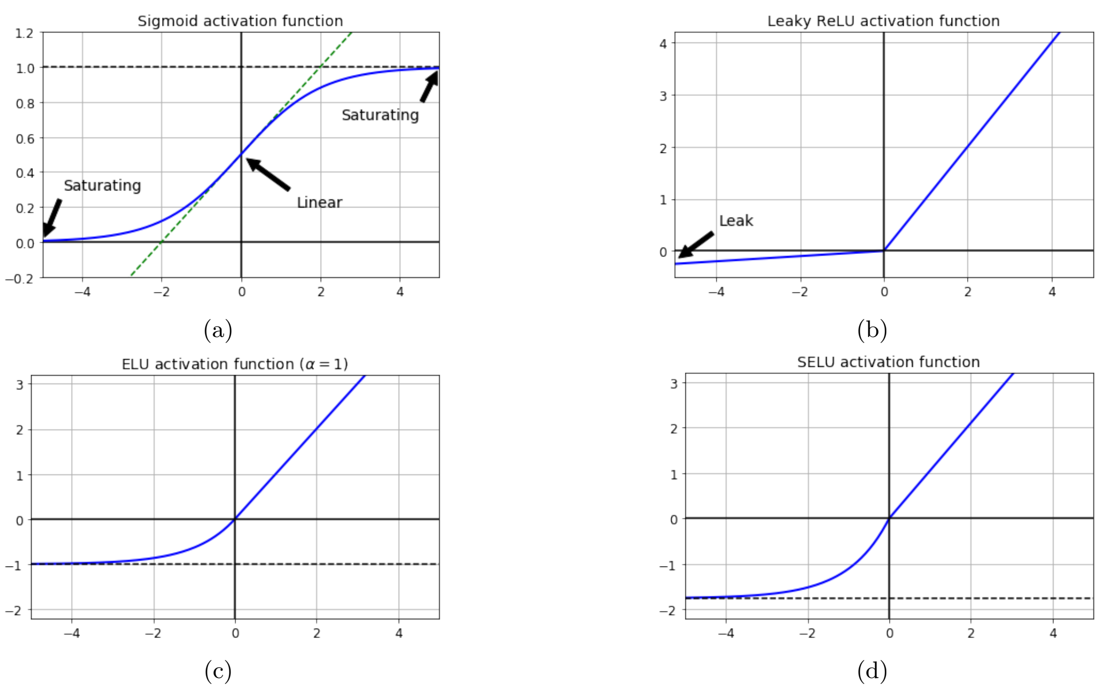
Traditionally, due to the activation functions used in models for describe biological neurons, there have been used "S"-shaped functions such as the sigmoid (or *logistic*, see Figure 5.5a) and the hyperbolic tangent functions as activation functions to define neural networks. We say that these type of activation functions "saturate" (to either 0 or 1 in the case of the sigmoid) in the extremes if the input values are quite small or quite big. In these cases the derivatives σ' are nearly zero, and if we remember the backpropagation equations from Theorem (5.1), the derivatives of the loss with respect to the parameters are proportional to σ' so that they will become small if the neurons "saturate".

Another activation that was popular and also inspired on biological NN was the so-called `ReLU` function defined as

$$\texttt{ReLU}(z) = \max(0, z). \tag{5.23}$$

Then, variants of this function have been proposed by several authors and are reported to be better for training in terms of avoiding vanishing (or exploiting) gradient problems due to "saturation".

In Figures 5.5b, 5.5c and 5.5d we show the currently most used variants of the `ReLU` activation function.



(a) (b) (c) (d)

Figure 5.5: (a) Sigmoid (b) Leaky-ReLU (c) ELU with $\alpha = 1$ [54] (d) SELU with $\beta \approx 1.05$, $\alpha \approx 1.67$, [55]. Figures taken from [20].

Mathematically, these functions are defined as follows:

$$\texttt{Leaky-ReLU}(z) = \max(0, \alpha z),\ \ \alpha > 0 \tag{5.24}$$

$$\texttt{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{5.25}$$

$$\texttt{SELU}_\alpha(z) = \beta\texttt{ELU}_\alpha(z),\ \ \beta > 1 \tag{5.26}$$

Note that for the activation functions we have considered, apart from the sigmoid, all are not differentiable in any neighbourhood of zero, so technically they do strictly not satisfy the conditions of the backpropagation theorem 5.1. However, in practice we will not have values that small............

5.3.2 Weight and biases initialization

When we explained the backpropagation algorithm to compute the gradients of the loss function with respect to the parameters of a NN, we talked about how to update them, but we did not talk about how to initialize them.

es el verbo ade-

It has been seen that if the weights (and biases) are initialized randomly (for example with a normal distribution with mean 0 and variance 1) with the same distribution, the variance of the outputs a^l to a given layer are much greater than the variance of the inputs to this layer. Thus, going forward in the network, the variance keeps increasing after passing each layer, until at some point it is very probable that from some layer on the activation functions saturate (if we use sigmoid or hyperbolic tangent, for instance).

In 2010 Xavier Glorot and Yoshua Bengio proposed a way to initialize weights to avoid these phenomena [56] for activation functions that can saturate. The idea was to initialize the weights of each layer based on the number of input neurons in the previous layer (n_{in}) and the number of neurons in the current layer (n_{out}). Inspired by these ideas, some authors [57] have proposed similar initialization criteria for different activation functions. In table 5.1 we summarize these criteria for initializing with uniform and normal random distributions.

Table 5.1: Initialization parameters for each type of activation function [20]. The normal distributions always have zero mean.

Activation function	Uniform distribution $[-r, r]$	Normal distribution
Logistic (sigmoid)	$r = \sqrt{\dfrac{6}{n_{\text{in}} + n_{\text{out}}}}$	$\sigma = \sqrt{\dfrac{2}{n_{\text{in}} + n_{\text{out}}}}$
Hyperbolic tangent	$r = 4\sqrt{\dfrac{6}{n_{\text{in}} + n_{\text{out}}}}$	$\sigma = 4\sqrt{\dfrac{2}{n_{\text{in}} + n_{\text{out}}}}$
`ReLU` (and its variants)	$r = \sqrt{2}\sqrt{\dfrac{6}{n_{\text{in}} + n_{\text{out}}}}$	$\sigma = \sqrt{2}\sqrt{\dfrac{2}{n_{\text{in}} + n_{\text{out}}}}$

These criteria to initialize weights is also known as *variance scaling initialization* and it is the one we will use for initializing NN.

5.3.3 Batch normalization

Although by combining novel activation functions with variance scaling initialization can significantly reduce vanishing/exploding gradients problems at the beginning of training [20], it does not ensure that these problems will not come back at some point during training.

In 2015 Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) [58], and it has become very popular for being able to improve the majority of DNN. This technique consists of adding an operation `BN` just before applying the activation function of each layer.

Imagine we wanted to apply the `BN` operator in the l-th layer. Then, instead of having the output a^l of this layer defined as in equation (5.2), it would now be defined as

$$a^l = \sigma(\mathtt{BN}(z^l)). \tag{5.27}$$

Basically, `BN` zero-centres and normalizes the weighted inputs z^l, and then scales and shifts the result using two additional parameters (per layer) ν^l (scale), η^l (offset) that can be learnt during training.

In Algorithm 4 it is described how the `BN` operator works. In order to zero-centring and normalize the inputs, the algorithm estimates the inputs' mean and standard deviation using the empirical mean and standard deviation over the observations of the current mini-batch. Note that when working with a mini-batch, equation (5.27) has to be thought as a component by component equality (one for each element of the mini-batch). Moreover, it is considered a tiny number ϵ to avoid division by zero when normalizing inputs (*smoothing term*).

Algorithm 4 Batch Normalization operator [58] for a given input mini-batch X_j of size m_B.

procedure $\mathtt{BN}(z_x = z(x),\ \forall x \in X_j,\ \nu,\ \eta,\ \epsilon)$
  $\mu_B \leftarrow \frac{1}{m_B} \sum_{x \in X_j} z_x$ ▷ Estimation of the mean
  $\sigma_B^2 \leftarrow \frac{1}{m_B} \sum_{x \in X_j} (z_x - \mu_B)^2$ ▷ Estimation of the variance
  for $x \in X_j$ **do**
    $\hat{z}_x \leftarrow \frac{z_x - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}}$ ▷ Zero-centred and normalized input
    $\hat{z}_x \leftarrow \nu\, \hat{z}_x + \eta$ ▷ Scaled and shifted input
  end for

  return: $\hat{z}_x$ for all $x \in X_j$.
end procedure

A drawback of BN is the slowdown of training, so it is worth checking first if the DNN works well enough without applying BN.

5.3.4 Regularization techniques

"*Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set*" [20].

In this section we present very briefly three common techniques that are used to avoid overfitting when using DNN. We consider early stopping, ℓ_1 and ℓ_2 regularization, and dropout, since these techniques could be applied in the context of Reinforcement Learning using DNN, as we will see in chapter 6.

Early stopping

Early stopping consists in interrupting training when the performance starts dropping. In a classification framework, performance would be measured for example with validation accuracy, and in a Reinforcement Learning framework the metric could be the rewards.

ℓ_1, ℓ_2 - regularization

The ℓ_1 and ℓ_2 regularization techniques, are shrinkage methods commonly used for generalized linear models and also known by Lasso and Ridge regularization, respectively.

These methods consist in adding an extra term Ω_θ to the cost function J_θ, i.e.,

$$J_\theta \longrightarrow J_\theta + \Omega_\theta.$$

In the context of deep neural networks regularization is usually only applied to the weights and not the biases. In this case the expressions for Ω_θ are the following:

For ℓ_1-regularization,

$$\Omega_\theta^{\ell_1} = \sum_i |\theta_i| = \sum_{j,k,l} |w_{jk}^l|. \tag{5.28}$$

For ℓ_2-regularization,

$$\Omega_\theta^{\ell_2} = \sum_i \theta_i^2 = \sum_{j,k,l} (w_{jk}^l)^2. \tag{5.29}$$

The term Ω_θ imposes a penalty on parameters (weights) that are very big, so that by adding this extra term to the usual cost function, one can avoid the parameters of the network becoming very large. In addition, a consequence of adding these penalties is that some “irrelevant” weights tend to become very small (and with ℓ_1 even become exactly zero). Hence, ℓ_i-regularization is a way of reducing the complexity of a DNN.

Dropout

Dropout is one of the most popular regularization techniques and it was proposed by G.E. Hinton in 2012 [59]. It has been proven to be very successful in improving the performance of many DNN structures.

In order to apply dropout one introduces a hyperparameter $p_d \in (0, 1)$, called *dropout rate*, and in every training step, with probability p_d every neuron of the network except those from the output layer are “dropped out” (as it were not in the network, hence not connected to any other neuron. See Figure 5.6). After all training, neurons do not get dropped nevermore.

In order to understand intuitively why dropout works we quote [20]:

“*Neurons trained with dropout cannot co-adapt with their neighbouring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of they input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.* ”

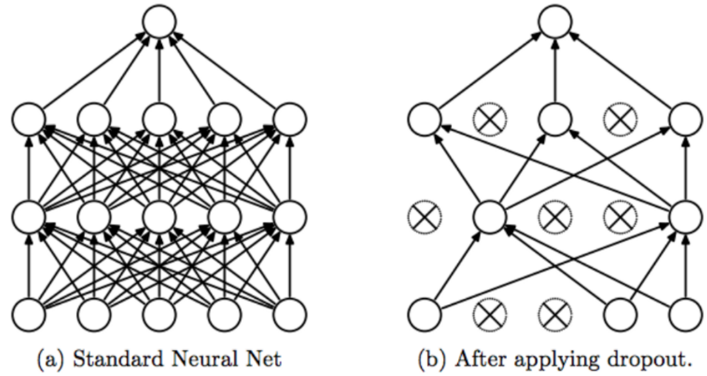


Figure 5.6: Dropout representation. (a) A standard feedforward fully connected DNN. (b) The DNN in (a) after before applying dropout, so that some neurons become inactive (the ones with a cross inside) and do not appear in the current training step. Picture taken from the authors of dropout [59].

5.4 Learning a simple policy with a Deep Neural Network

In order to put in practice what we have learnt about Deep Neural Networks, in this section we consider one of the classical applications of Supervised Learning: classification.

First in section 5.4.1 we explain briefly how to use DNN for classification problems; then in 5.4.2 we set up a classification problem related to our product delivery problem, and finally in section 5.4.3 we show the results of solving that classification problem with DNN.

5.4.1 Deep Neural Networks and Classification

Imagine that we had data about the volume, the diameter and the weight of a sample of fruits, each of them being either an apple, an orange, a banana or a kiwi. In other words, we would have four classes of fruits and data about three features (volume, diameter and weight) that can be easily measured from representatives to these classes.

Our goal is to have a model such that given a set of values for the volume, the diameter and the weight, it is able to predict from which class of fruit (either an apple, an orange, a banana or a kiwi) these values come from. To train the model we need a train set X such that each $x \in X$ is a triple of feature values, and a target set Y containing the labels $y(x)$ for each observation in X (e.g., the name of the fruit that corresponds to each observation in the train set).

The easiest approach would be to consider a DNN with one output neuron per class, and consider the output value a_i^L as a measure or score for the i-th class. Then, for instance, given a training instance $x \in X$, corresponding to class $y(x) = i$, we would expect a higher score in the i-th output value; i.e., $a_i^L > a_j^L$, for all $i \neq j$. In this framework a^L are called the *logits*.

Remember that in order to train a DNN (for classification, in this case), we need some loss function to minimize. As it is shown in Figure 5.7, what is done in practise is to apply a *softmax* activation function to the usual output layer which we now call "logits layer". In this way, fixed $x \in X$, the final outputs p_i of the DNN are normalized to sum 1, so that they can be considered as a probability distribution $p(x)$ (see eq. (5.30)) over classes.

$$p_i = \frac{\exp(a_i^L)}{\sum_j \exp(a_j^L)} \tag{5.30}$$

Then we can define a "true probability distribution" $q = q(x)$ as follows:

$$q_i = \begin{cases} 1 & \text{if } y(x) = i \\ 0 & \text{if } y(x) \neq i \end{cases} \tag{5.31}$$

With these probability distributions p and q defined, one can consider the loss function to be a distance between them. For instance, the usual distance chosen is the *cross entropy*, which can be written as in equation (5.32).

$$H(p,q) = \sum_i q_i \log p_i \tag{5.32}$$

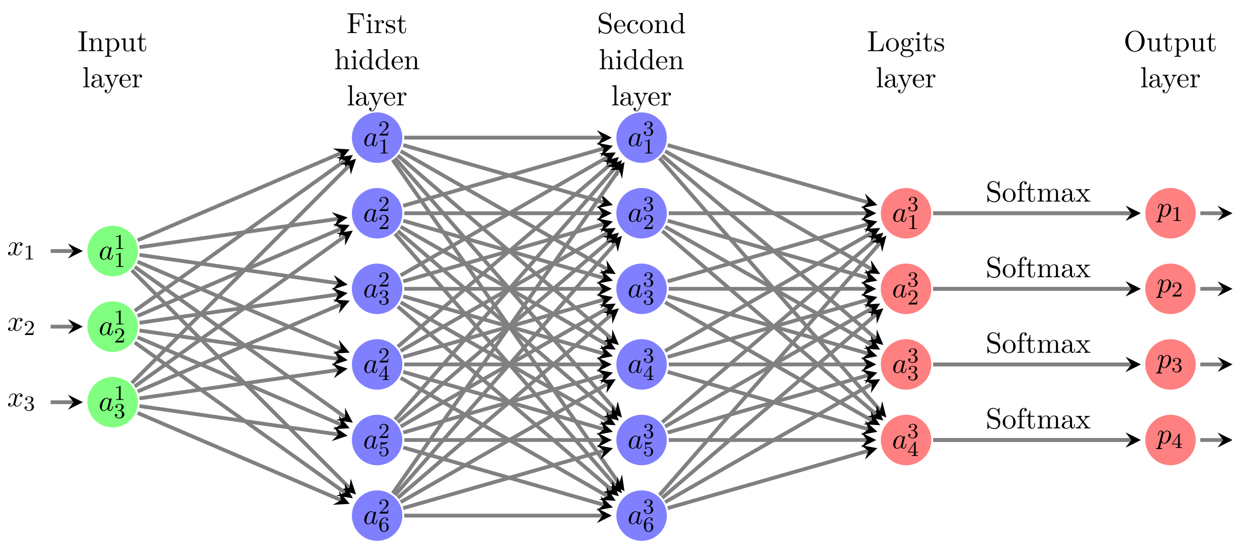


Figure 5.7: Representation of a feed-forward fully connected DNN for a classification problem where data has three features and there are four classes. We consider a simplified case with two hidden layers with six neuron each. The logits layer behaves in the same way as a hidden layer but then to the outputs of this layer, the *logits*, we apply a softmax activation function to normalize outputs so that they sum 1.

Since one usually uses mini-batches, the final loss function used for training a DNN such as the one in Figure 5.7 becomes

$$J_\theta = -\frac{1}{m_B} \sum_{x \in X_j} H(p(x), q(x)), \tag{5.33}$$

where X_j is a mini-batch of size m_B.

5.4.2 Simulation framework for learning a hard-coded policy

In order to get started with DNN and the product delivery problem we modelled with the point of view of Reinforcement Learning in chapter 3, we study the capability of a DNN of learning a simple hard-coded policy.

First, remember that we are restricting to the toy model we explained in section 3.3, and we consider a system with $n = 3$ shops and a single truck ($k = 1$). In this case, the state of the system will be given by the current stock of each shop, i.e., $s_t = (c_0, c_1, c_2)$, and the $n + 1 = 4$ possible actions for the truck would be going to either one of the shops or staying at the depot, i.e. $a_t = i$ for $i \in \{0, 1, 2, 3\}$ [11].

Next, lets consider the policy π_m that sends the truck to the shop with minimum stock, if it is possible (i.e., if the load L of the truck fits all in the shop without surpassing the maximum level of stock), or makes it stay at the depot otherwise. Mathematically, speaking, we define π_m as follows:

$$\pi_m(s) = \begin{cases} \arg\min(s) & \text{if } \min(s) + L \leq C_{\arg\min(s)} \\ n & \text{otherwise.} \end{cases} \tag{5.34}$$

Finally, note that if we now generate a train dataset X containing possible states of the system using some distribution $I(s)$ (which is equivalent to generate a table with a row for each state, and columns contain the level of stock of each shop), and a target set Y given by $\pi_m(X)$ (where here it means to apply π_m to each row of X); then we are in the framework of a classification problem like the one we exposed in the previous section. In short, our train dataset consists of observations with three features (the stock of each shop) and an associated target value equal to the action to take according to π_m.

With these ideas, if we consider a DNN F_θ with the same structure as in Figure 5.7, that is, n input neurons, some hidden layers and $n + 1$ output neurons with a final softmax activation, if it learns properly after training, $\hat{\pi}_m := \arg\max F_\theta$ will act almost or exactly equally as the policy π_m, i.e., $\hat{\pi}_m(s) = \pi_m(s)$ for most of the states s we may use as input to de DNN.

Habría que coment
argmax se hace nu
dices des de 0. Tam
ser conveniente hac
icy sea un poco est
pleando con multin
de argmax

5.4.3 Simulations and Results

In this section we present results for different DNN simulations prepared in the classification framework we introduced in the previous section.

To define DNN in PYTHON code and train them, we use the popular library Tensorflow from Google [60]. Moreover, to work with a product delivery environment adapted to our definitions and model of states and actions, we have developed an Open AI gym environment [14, 61]. All the code to reproduce the results we show in this section can be found in a Jupyter notebook from the *Imitation-Learning* folder in [13].

First of all, we need to generate train, test and validation datasets containing state and action pairs of the form $(s, \pi_m(s))$. To choose states, we use a uniform random distribution for each of the components of an state $s = (c_0, c_1, c_2)$ ranging from 0 to the maximum capacities C_i. Hence, we sample $s \sim I(s)$ where I is such that $c_i \sim \text{Uniform}([0, C_i])$, $i = 0, 1, 2$.

For all simulations we have generated a train data set with 10^4 observations and test and validation sets of 10^3 observations each.

[11] Remember that $a_t = n$ is the action of staying at the depot.

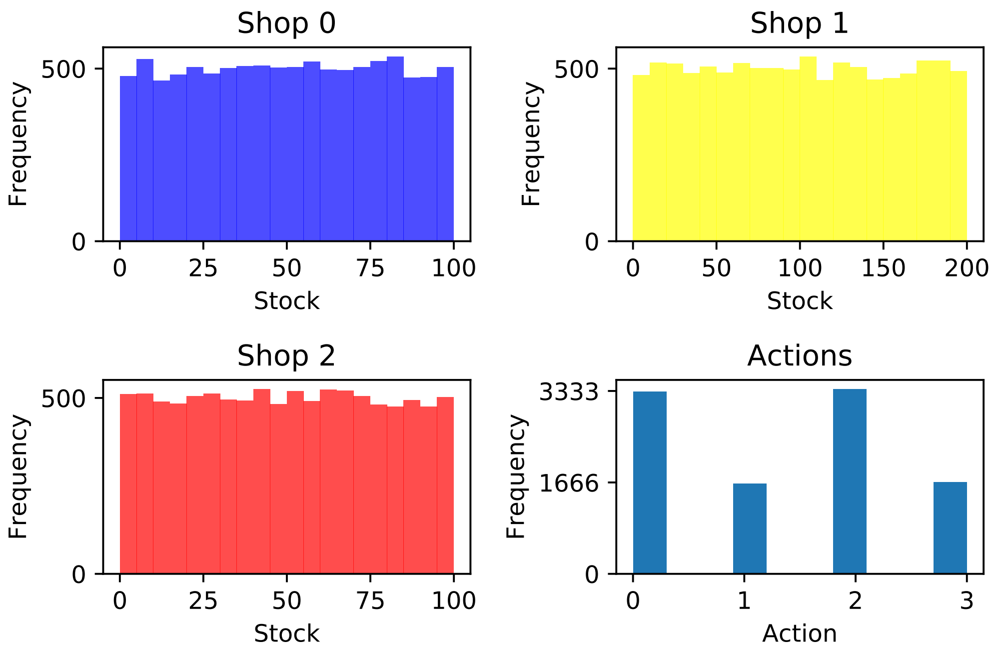


Figure 5.8: Histograms for the stocks of each and the corresponding actions for a train dataset of 10^4 observations. The bin size for the stocks is 20.

In Figure 5.8 we show the histograms of the train dataset, and in Table 5.2 the main features of the system used to generate the data.

Table 5.2: Maximum capacities and daily consumption rates for shops and the truck.

	Maximum capacity	Daily consumption rate
Shop 1	100	16.5
Shop 2	200	4.0
Shop 3	100	10.5
Truck	50	either 0 or 50

We can see that the empirical distributions for the stock of each shop approximates very well a uniform between 0 and the corresponding maximum capacity. About the distribution of actions, we can see that the times that the action of sending the truck to shop 1 is about half the times that a truck is sent to shops 0 and 2. This makes sense since shop 1 has the double maximum capacity when compared to shops 0 and 2, so it is half likely that shop 1 has the minimum current load when initializing stocks uniformly. As a consequence, this leads the frequency of staying at the depot to be the same as sending a truck to shop 1.

Every simulation we present in this section consists on training a DNN with n_1 neurons in the first hidden layer and n_2 neurons in the second. For weight initialization we have used normally distributed variance scaling (see 5.3.2) and biases have been initialized to zero by default. With regards to activation functions, there are considered ELU activation functions in the two hidden layers and no activation function is used in the input layer. In the output layer, a softmax activation is used to normalize outputs so that they sum 1. Finally, the optimization algorithm used to minimize cross entropy is the Adam optimizer [51].

In table 5.3 we show the hyperparameters used for training the DNN.

Table 5.3: Hyperparameter values

Hyperparameter	Value
Learning rate	0.01
`batch_size`	50
τ (episode length)	30
epochs	500

After training, at the end of a simulation, it is obtained a DNN model that one can use to make predictions on a test dataset, for instance. In section 5.3 we introduced some techniques that can improve the performance of a DDN. However, for the example we are working with now to learn about DNN, such techniques are not needed since the accuracies obtained without them are good enough.

There is only one remarkable thing we have observed when testing simulations: we said that scaling the input data usually improves accuracy when using a DNN for classification. Nevertheless, for the particular data we are using, we have seen that the DNN models perform better without scaling the inputs.

In table 5.4 we summarize the results of training DNN varying the number of neurons n_1, n_2 in the hidden layers. The accuracies are obtained by averaging over 5 runs for each simulation and the standard deviation is used as a measure of the error.

Table 5.4: Cross validation train and test accuracies for different simulations and 5 runs for each one. The results are given in the form $\bar{x} \pm \sigma$, where $\bar{x}$ is the average of quantity x over runs and σ the corresponding standard deviation.

Simulation id	Classifier	Train accuracy	Test Accuracy	CPU time per run
1	DNN ($n_1 = 100, n_2 = 50$)	0.993 ± 0.003	0.996 ± 0.003	0.81 min
3	DNN ($n_1 = 10, n_2 = 5$)	0.993 ± 0.001	0.992 ± 0.004	0.59 min
4	DNN ($n_1 = 4, n_2 = 4$)	0.991 ± 0.002	0.990 ± 0.003	0.57 min
7	DNN ($n_1 = 500, n_2 = 200$)	0.994 ± 0.003	0.993 ± 0.002	2.0 min

In figure 5.9 we show the validation accuracies, the best validation accuracy and the train accuracy (of the last batch of each epoch, not the whole train dataset) as a function of the number of epochs. The simulations shown correspond to the first simulation run done to obtain the averaged values in table 5.4.

At first sight, the four simulations from some epoch on seem to behave similarly, and the validation accuracies tend to oscillate around some mean value above 0.95. For instance, for simulation 1, the best validation accuracy seems to stabilize in about 100 epochs, whereas for simulation 3, where we have reduced the number of neurons of each hidden layer by a factor of 10, the best validation accuracy does not stabilize until the 250th epoch (approximately).

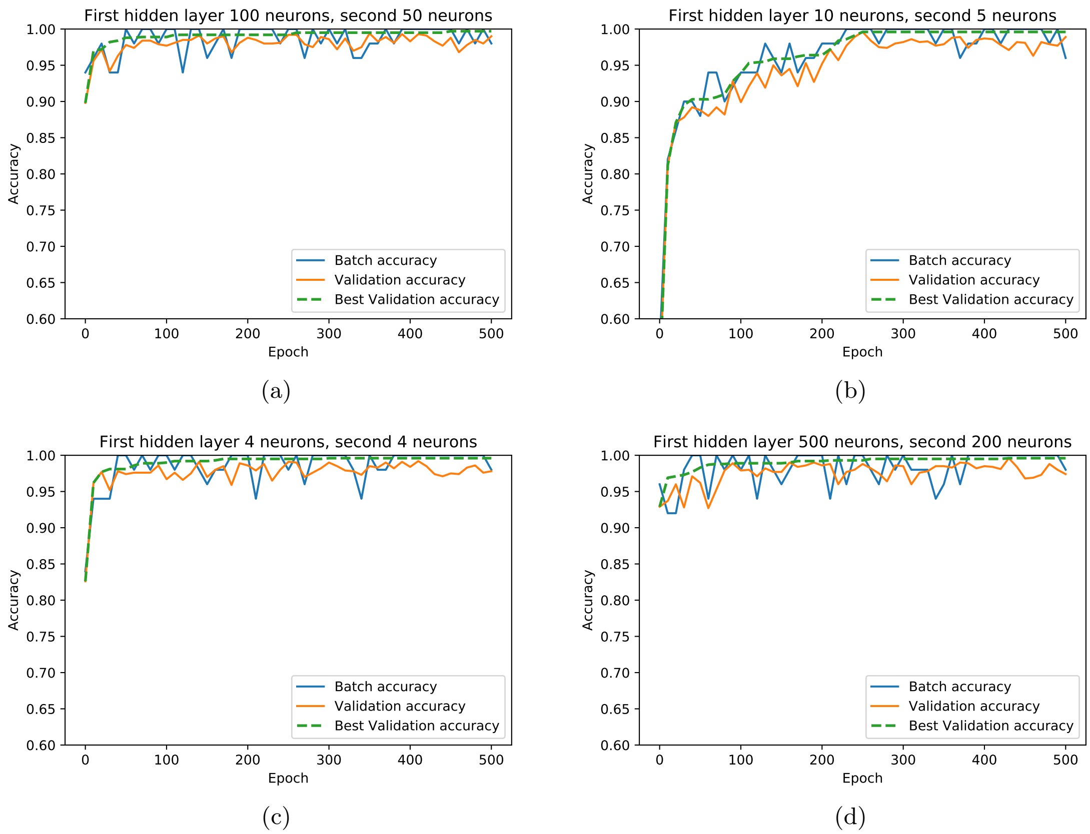


Figure 5.9: Each subplot displays the batch train accuracy, validation accuracy and best validation accuracy as a function of the training epoch for the first run of each simulation detailed in 5.4.

To obtain more rigorous conclusions about each simulation and see which would be the best classifier model among the four we have considered; we compare the cross validated accuracies from table 5.4. To decide which model is *better* we are going use the test accuracy as a metric.

If we consider the confidence intervals of simulation 1 and 4 for test accuracy, since they do not overlap and the first one has a higher mean test accuracy, we can say that classifier 1 is better than classifier 4. If we now look at the intervals for classifiers 3 and 7, we see that they overlap with the interval for classifier 1 so now we could not rigorously say that classifier 1 is better than 3 and 7 given one standard deviation σ. However, classifier 7 needs a bit more than the double of the time that classifier 1 needs to be trained and moreover classifier 1 has a larger mean test accuracy, so we would choose 1 against 7. Similarly, although the intervals for classifiers 1 and 3 overlap and although classifier 3 need a bit less time to be trained, we chose again classifier 1 as the best classifier among those four since it leads to larger average test accuracies.

Finally, it is worth to remark a fact that has been observed many times in the literature when working with DNN and other machine learning models in general: as the learning model becomes more and more complex, it tends to overfit the train data more and more. For DNN, increase complexity means to increase the number of neurons in each layer and/or the number of hidden layers. In our case, we have maintained the number of hidden layers constant and we have varied the number of neurons on each layer. If we look at the train accuracies from table 5.4, we can observe that as the number of neurons in each layer increases, the train accuracy tends to increase (in average).

Once we have chosen a classifier, in order to see how the learnt DNN policies perform when using them to control the product delivery system of three shops and one truck, we have simulated 9 episodes of 30 days each and plotted the stock level of each shop in the same plot and as a function of the time step (see Figure 5.10).

To check if at a given time step the truck is performing the correct action [12] we should take into account the following observations:

- If the minimum stock is at shop 1, the truck must always go there. The reason is that in the worst of the cases, the stock of shop 1 would be almost 100 (equal to the maximum capacity of the other two shops), and since the truck's capacity is 50 and shop 1 has a capacity of 200, it would have enough space to fit all the product in the truck.
- If the minimum stock is at shop i, for an $i \in \{0, 2\}$, the truck must go to that shop if and only if the current stock there is less or equal than 50. Otherwise the truck must stay at the depot since there would not be enough space in the shop to store all the product carried by the truck.

Figure 5.10: Each subplot represents an episode of 30 time steps where we represent the stocks of each shop during the corresponding episode. The policy used to take actions is the DNN policy learnt in the first run of the simulation with $id = 1$ (see table 5.4).

With these considerations we can analyse the simulated episodes that follow a learned policy to control our product delivery system. Just to fix ideas, lets focus on episode 4 from Figure 5.10:

- We can see that shop 1 has a stock always greater than the other two shops. In particular, as expected, this stock is never the minimum so that the truck never goes to shop 1. Regards

[12] Remember that we are training a DNN to learn the policy defined mathematically in equation (5.34). Hence, by *correct action* we mean the action that would be taken using π_m.

to shops 0 and 2, at day $t = 0$ shop 2 has a stock a bit smaller than that in shop 0, and thus the truck goes to that shop and as a consequence the stock in shop 2 increases. Then at day $t = 1$ the stock of shop 0 is very close to zero and in particular below the stock of shop 2; therefore, the truck goes to shop 2 and its stock increases. If we continue this reasoning for each time step t until the end of the episode, we can see that the truck seems to be performing the correct action.

- However, in the plots of each episode it is not possible to appreciate if the learned policy is performing always the correct action. If there is a case where the learned policy sends a truck to a shop that is not empty enough to store the contents of the truck, the stock of that shop remains the same so that, apparently, it is like no truck has gone there [13]. In fact, in episode 3 there are two days, $t = 4$ and $t = 25$, where the learnt policy is not performing the correct action.

During the simulations of these episodes, we have recorded the episode number, the time step and the state of the system before applying an action in that time step,

a) `Incorrect action in episode 1, step 12 . Correct action: 3 , Chosen action:  0`
 `State before action:  [50.4 78.8 52.7]`

b) `Incorrect action in episode 1, step 17 . Correct action: 3 , Chosen action:  2`
 `State before action:  [67.9 58.8 50.2]`

c) `Incorrect action in episode 2, step 23 . Correct action: 1 , Chosen action:  3`
 `State before action:  [82.2 71.7 72.1]`

d) `Incorrect action in episode 4, step 4 . Correct action: 3 , Chosen action:  0`
 `State before action:  [ 50.5 163.4  70.8]`

e) `Incorrect action in episode 4, step 25 . Correct action: 3 , Chosen action:  2`
 `State before action:  [54.0  79.4 50.3]`

We can distinguish two basic errors that the learnt policy is committing:

- The most common, observed in a), b), d) and e), the chosen action corresponds to go to a shop where the stock is very near 50 but it is still greater so that the correct action would be staying at the depot. It is reasonable that the network hardly distinguishes such cases.
- Finally in case c), a slightly different confusion appears. The stocks of shops 1 and 2 before taking the action are very similar (71.7 and 72.1, respectively), but shop 1 has the minimum stock so that the truck should go there. However, the learnt policy has decided to make the truck stay at the depot. An intuitive explanation of why this has happened would be that the learnt policy "has seen that the shop with minimum stock is shop 2, and since the stock is above 50, it does not send the truck there".

To sum up, we have that for the 8 episodes of 30 days we have simulated, only 5 actions have been taken wrongly. This means that out of 240 days, in 5 the learnt policy has performed a wrong action and this is equivalent to have a $1 - \frac{5}{240} \cdot 100 \approx 98\%$ accuracy, a bit but not much, less than what we observed for validation and test datasets.

[13] We do not fill the shop until reaching the maximum capacity, since one of our restrictions was that a truck leaves the depot fully loaded and returns completely empty.

Chapter 6

Deep Reinforcement Learning

In chapter 5 we introduced Deep Neural Networks treated as parametrized functions and understood how their parameters can be trained to improve their outputs according to a predefined criteria or goal, such as classification.

In this chapter we are going to use DNN to play the role of a parametrized policy π_θ, and introduce a particular type of algorithms, Policy Gradient (PG), that allow us to train the network to improve the policy by means of simulated episodes. The field where there are used Deep Neural Networks to solve Reinforcement Learning problems is called Deep Reinforcement Learning (DRL).

6.1 Monte Carlo Policy Gradient algorithm

The Monte Carlo Policy Gradient method consists on updating the parameters of a parametrized policy $\pi_\theta(a|s)$ by means of the gradient of a loss function estimated averaging over simulated episodes. According to our interests, as it is typically chosen for episodic tasks, the cost function to consider will be the expected total discounted reward obtained in every episode,

$$J_\theta = \mathbb{E}_{\pi_\theta}\left[R_0^\gamma\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\tau-1} \gamma^k R_k\right]. \tag{6.1}$$

In this framework, our goal is to maximize the total discounted rewards obtained in every episode, in average. Thus, the problem has become a maximization problem where the parameters θ of the policy π_θ should be updated following the direction of the gradient by means of some algorithm such as Gradient Ascent (GA). For GA, an update step may look like

$$\theta^{k+1} \leftarrow \theta^k + \alpha \nabla_\theta J_\theta\big|_{\theta=\theta^k}, \tag{6.2}$$

where $\alpha > 0$ is the learning rate. Note that an implicit assumption about π_θ is differentiability with respect to the parameters.

In order to update parameters, we need a way to compute the gradient of the loss function J_θ. In practice, this gradient will be estimated by sampling episodes, estimating the gradient for each episode and then averaging over the sampled episodes. The Policy Gradient Theorem 6.1 [28], gives an expression for the gradient of many loss functions, in particular the one we have defined in equation (6.1).

Theorem 6.1 (Policy Gradient). *Assuming an episodic MDP, for any differentiable policy* $\pi_\theta(a|s)$ *and the policy loss function* J_θ *defined in equation* (6.1)*, the policy gradient is*

$$\nabla_\theta J_\theta = \mathbb{E}_{\pi_\theta}\left[R_0^\gamma \sum_{t=0}^{\tau-1} \nabla_\theta \log \pi_\theta(A_t|S_t) \middle| S_0 = s, A_0 = a \right] \tag{6.3}$$

$$= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=0}^{\tau-1} \gamma^t R_t\right) \left(\sum_{t=0}^{\tau-1} \nabla_\theta \log \pi_\theta(A_t|S_t)\right) \middle| S_0 = s, A_0 = a \right]. \tag{6.4}$$

Proof. To understand how policy gradient equations are derived see the references [62], chapter 7 (pgs. 230-233) from [30] and [63]. □

Equation (6.4) gives us a way to estimate the gradient of the loss function by sampling some number N of episodes $E_j = \{s_0^j, a_0^j, r_0^j, s_1^j, a_1^j, r_1^j, ..., s_{\tau-1}^j, a_{\tau-1}^j, r_{\tau-1}^j, s_\tau^j\}$ - for $j = 1, ..., N$ - as follows:

$$\nabla_\theta J_\theta \approx \frac{1}{N} \sum_{j=1}^{N} \left[\left(\sum_{t=0}^{\tau-1} \gamma^t r_t^j\right) \left(\sum_{t=0}^{\tau-1} \nabla_\theta \log \pi_\theta(a_t^j|s_t^j)\right) \right] \tag{6.5}$$

Using equation (6.5) we consider Algorithm 5 with pseudocode for the Monte Carlo Policy Gradient (MCPG) in order to train a policy π_θ.

Algorithm 5 Monte Carlo Policy Gradient algorithm to train a policy π_θ via gradient ascent.

procedure MCPG(π_θ, γ, α, τ , `n_episodes` $= N$, `n_iterations`)
    $\theta \leftarrow \theta_0$ ▷ Initialise θ randomly.
    for $i \in \{1, ..., \texttt{n\_iterations}\}$ **do**
        Sample N episodes $E_j = \{s_0^j, a_0^j, r_0^j, s_1^j, a_1^j, r_1^j, ..., s_{\tau-1}^j, a_{\tau-1}^j, r_{\tau-1}^j, s_\tau^j\} \sim \pi_\theta$
        for $j \in \{1, ..., \texttt{n\_episodes}\}$ **do**
            for $t \in \{0, ..., \tau - 1\}$ **do**
                Compute and store $\nabla_\theta \log \pi_\theta(a_t^j|s_t^j)$
            end for ▷ End for over time steps
            Compute and store

$$\nabla_\theta J_\theta^j = \left(\sum_{t=0}^{\tau-1} \gamma^t r_t^j\right) \left(\sum_{t=0}^{\tau-1} \nabla_\theta \log \pi_\theta(a_t^j|s_t^j)\right)$$

        end for ▷ End for over episodes
        Obtain an averaged estimation of the gradient over episodes

$$\nabla_\theta J_\theta = \frac{1}{N} \sum_{j=1}^{N} \nabla_\theta J_\theta^j$$

        Update parameters via gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_\theta J_\theta$$

    end for ▷ End for over iterations
end procedure

We have considered Gradient Ascent updates, but as we commented in chapter 5 for Gradient Descent, better and more sophisticated methods are used in practise.

6.2 Deep Policy Gradient for product delivery

In this section we use deep neural networks that play the role of a parametrized policy π_θ, which take as inputs a representation in $\mathbb{R}^n$ - for some n - of the states, and outputs the probability of taking each action $a \in \mathcal{A}$ given the input state. Remembering that we denoted Λ the cardinality of $\mathcal{A}$, then the DNN policy would consist of an input layer of n neurons and an output layer with Λ neurons, one for each possible action. In Figure 6.1 we show such a DNN architecture with two hidden layers. Note that after the output layer (whose outputs are the *logits*), a softmax activation function is applied to normalize outputs and convert them to a probability distribution over actions "$p_\theta = (p_1, ..., p_\Lambda)$". If $\mathcal{A} = \{a_1, ..., a_\Lambda\}$, then $p_i = \pi_\theta(a_i|\hat{s})$, where $\hat{s} = (I_1, ..., I_n)$ is a representation in $\mathbb{R}^n$ of state $s \in \mathcal{S}$.

If we return to the product delivery problem with states and actions defined in section 3.3, states are exactly vectors $(c_1, ..., c_n) \in \prod_{i=0}^{n-1} [0, C_i] \subset \mathbb{R}^n$, where c_i are the "current" stocks of each shop and C_i the respective maximum capacities. With the previous notation, we would have $I_j = c_j \ \forall j$ and $s = \hat{s}$ in this case.

About the actions, if we have k trucks and each one can either go to one of the n shops or stay at the depot, then the number of possible actions combining all trucks results to be $\Lambda = (n+1)^k$, and this gives us the number of neurons needed in the output layer.

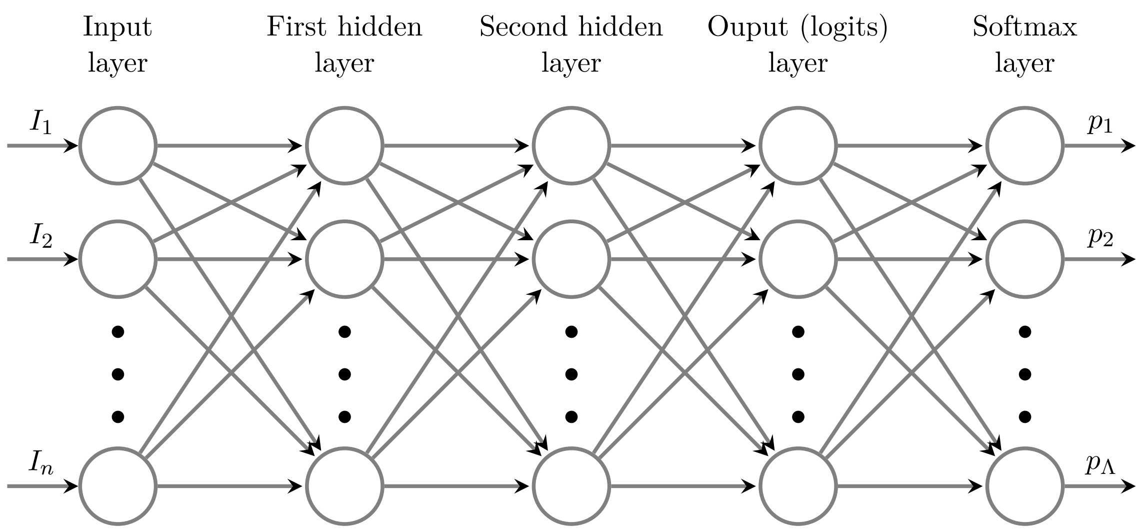


Figure 6.1: Deep Neural Network structure for a first policy gradient approach: one output neuron per possible action. In general we may consider an arbitrary number of hidden layers.

In [13] we have a Jupyter notebook with code to implement Algorithm 5 to train a DNN with two hidden layers.

There is only one subtle detail we have to notice. Policy Gradient theorem gives us a way to compute the gradient of the loss function J_θ but we still need to know how to compute the quantities $\nabla_\theta \log \pi_\theta(a_t^j|s_t^j)$. To understand this it is worth to read again section 5.4.2 where we defined cross entropy (eq. (5.32)), a distance between probability distributions that serves as a cost function for classification problems.

Imagine that the vectorial representation of state s_t^j is the current input of the DNN that we are training, and $\hat{a}_t^j$ is a one-hot encoded vector such that $(\hat{a}_t^j)_i = 1$ if $\hat{a}_t^j = a_i$ and $(\hat{a}_t^j)_i = 0$ otherwise [1]. The vector $\hat{a}_t^j$ could be obtained, for example, by sampling a number r between 1 and n with a multinomial distribution with the probabilities $p_1, ..., p_n$, and then setting $(\hat{a}_t^j)_i = 1$ if $i = r$ and $(\hat{a}_t^j)_i = 0$ otherwise. This would be a way of deciding which action $a_r \in \mathcal{A}$ to take according to our DNN policy given an input representation of state s_t^j. An alternative would be to consider $r = \arg\max_i p_\theta$, and then choose action a_r; the only difference is that in this case we would have a deterministic policy and in the previous case a stochastic one.

Then, with the notation introduced so far in this section, we can define a cross entropy function defined as follows:

$$H(\hat{a}_t^j, p_\theta) = \sum_{i=1}^{\Lambda} (\hat{a}_t^j)_i \log p_i = \sum_{i=1}^{\Lambda} (\hat{a}_t^j)_i \log \pi_\theta(a_i | s_t^j), \tag{6.6}$$

where p_i is the output of the i-th neuron in the softmax layer (see Figure 6.1) when using s_t^j as input, and corresponds to a probability associated to action a_i.

Now if we take gradients with respect to θ in equation (6.6) we obtain

$$\nabla_\theta H(\hat{a}_t^j, p_\theta) = \sum_{i=1}^{\Lambda} (\hat{a}_t^j)_i \nabla_\theta \log \pi_\theta(a_i | s_t^j). \tag{6.7}$$

Since only one of the $(a_t^j)_i$ is different from zero (lets say for $i = r$), we have that the gradient of the cross entropy defined this way is the term $\nabla_\theta \log \pi_\theta(a_t^j | s_t^j)$ (with $a_t^j = a_r$) we needed to compute.

6.2.1 Simulations and Results

In this section we present the results of some Monte Carlo Policy Gradient simulations that we have done by training DNNs as in Algorithm 5. The code has been implemented in Tensorflow [60] and all Jupyter notebooks of the simulations are available in the Policy Gradient folder from [13].

We have considered almost the same simulations as in Q-learning (see Table 6.1): deterministic and stochastic, and with or without transport costs varying the transport cost coefficient. Moreover, apart from the number of shops and trucks, we consider the values in table 4.1 for parameters that are common in Q-learning and in Policy Gradient algorithms.

We start considering a system of 5 shops and 2 trucks, as we did for Q-learning simulations, and finally show the results obtained when simulating a system of 12 shops and 3 trucks to prove the better scalability of DNNs with the structure shown in Figure C.11, when compared with Q-learning[2]. We note that we have not considered a higher number of trucks due to the fact that for $k = 4$ and - for instance - $n = 12$, the number of neurons in the output layer would be $\Lambda = (12+1)^4 = 28561$, which is a quite big number of neurons to train in a reasonable amount of time. On the contrary, for $k = 3$ trucks we have $\Lambda = (12+1)^3 = 2197$ which is not that much.

The way we introduce noise to the consumption rates when doing stochastic simulations is again the same as in Q-learning (see Section 4.2). For exploration, during training, the way actions are

[1] It is the q probability distribution we defined in equation (5.31).
[2] In Q-learning $n = 12$ and $k = 3$ would....

chosen is given by multinomial sampling of the output probabilities p_i of the DNN. In addition, the noise of the consumption rates in stochastic simulations adds additional exploration, which could improve (or not) the performance of the DNN. During testing, in order to be deterministic, instead of using multinomial sampling for choosing actions, we choose actions greedily by applying an $\arg\max$ function to the p_is.

During training, in each iteration in which the average discounted rewards over simulated episodes improves, the model is saved (i.e., the weights and biases in that iteration are saved and substitute the previous "best model saved"). During testing, the weights and biases from the last best model saved are used [3].

Table 6.1: Simulation parameters (that differ from the default ones in table B.2)

	Id	Simulation	n, k	μ_1	Stochastic consumption
1)	316	Deterministic without transport/unload costs	5,2	0	No
2.1)	317	Deterministic with transport/unload costs	5,2	10^{-6}	No
2.2)	319	Deterministic with transport/unload costs	5,2	10^{-4}	No
3)	324	Stochastic without transport/unload costs	5,2	0	Yes, 10%
4)	325	Stochastic with transport/unload costs	5,2	10^{-6}	Yes, 10%
4.1)	328	Stochastic with transport/unload costs	5,2	10^{-3}	Yes, 10%
5)	424	Stochastic without transport/unload costs	12,3	0	Yes, 10%
6)	425	Stochastic with transport/unload costs	12,3	10^{-6}	Yes, 10%

For all simulations we have arbitrarily chosen an identical initial configuration of the system just to show a single episode to illustrate the results. Moreover, in Appendix C.2 we have one table for each simulation containing information of many parameters averaged over 100 episodes to support the explanations of the results.

As we did for Q-learning simulations, in the plots where we show the stocks of each shop during a simulated episode, we have included dashed lines to indicate the different levels of stock (minimum level and capacity, danger level and maximum level and capacity) for each shop. Moreover, we have added a dashed line of the average stock level (over the episode) of each shop and a dashed line to indicate the stock where the maximum reward M is achieved (remember Figure 3.3).

Our DNNs are very basic, in the sense that we have not used any extra regularization technique such as the ones we commented in section 5.3. We show the results for this "simple" case, and we expect that experimenting a bit by combining some regularization techniques and different NN architectures, the learnt policies would be much better. However, due to the extra time this experimentation would require, it has been left as future work.

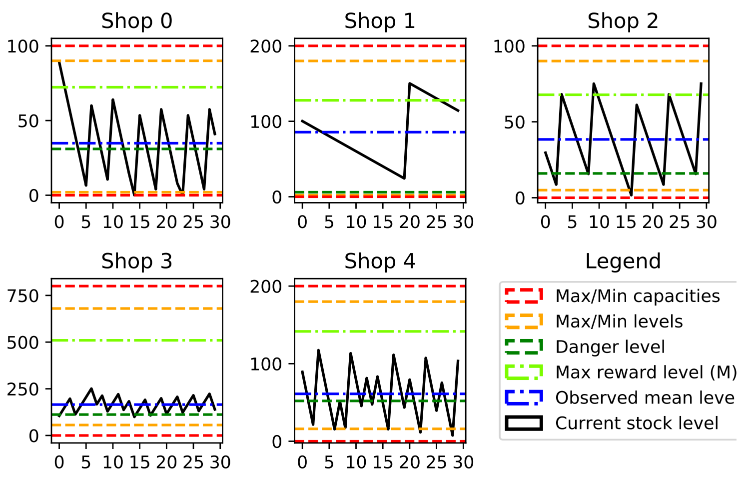
Finally we note that the discounted average rewards that we will display in the coming figures are computed by averaging the discounted reward of each of the `n_episodes` episodes simulated in every PG iteration (remember Algorithm 5). Thus, although in the horizontal label it says "episode", strictly speaking there should be said "iteration".

[3] Since in all cases the last best model is before the 75.000th iteration, the training for some simulations is variable but always with more than 75.000 iterations.

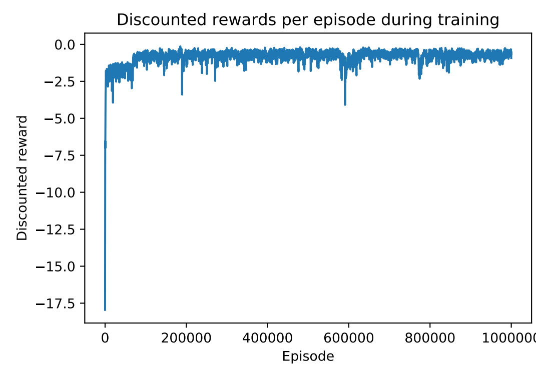
Deterministic simulations

Simulation 1 v.s. Simulation 2.1 (deterministic without and with transport costs)

In figures 6.2 and 6.3 we show the evolution of the levels of stock of each shop and the discounted rewards per PG iteration during training for two deterministic simulations. The former without transport costs contribution ($\mu_{\text{transport}} = 0$) and the latter with some transport contribution ($\mu_{\text{transport}} = 10^{-6}$). Similarly, in tables 6.2 and 6.3 there is summarized some relevant information about the simulated episode of these two simulations.

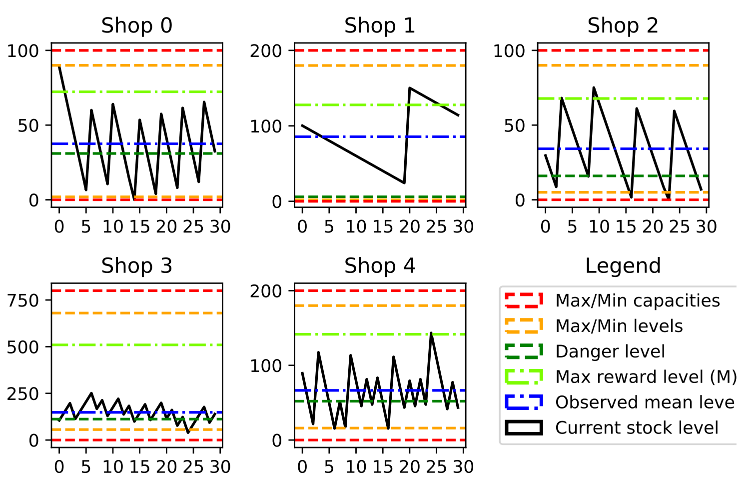


(a)

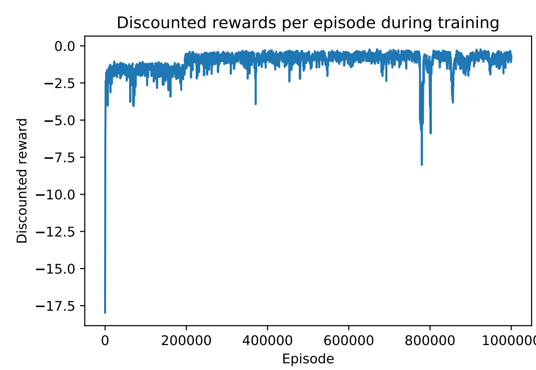


(b)

Figure 6.2: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic without transport costs, $\mu_{\text{transport}} = 0$). Transport costs: 33658. Trucks not delivering: 0. Total trucks sent: 42. (b) Discounted average rewards as a function of the number of training episodes.



(a)



(b)

Figure 6.3: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic with transport costs, $\mu_{\text{transport}} = 10^{-6}$). Transport costs: 33175. Trucks not delivering: 0. Total trucks sent: 42. (b) Discounted average rewards as a function of the number of training episodes.

Similarly as it happened for the two Q-learning simulations from chapter 4 that used the same parameter values than the current two, the number of trucks of each type sent to the first three shops is the same. The relevant differences are found in shops 3 and 4.

Without transport costs there are sent a total of 26 big trucks and 16 small trucks; whereas with some transport costs contribution, 25 big trucks and 17 small trucks are sent in total. Since the distance from the depot to shops 3 and 4 is the same, it is clear in this case that the difference in transport costs for both simulations is given by the fact that in the second simulation there is sent 1 big truck less than in the first, but one more small truck. Since it is cheaper to unload a small truck (remember expression (3.10)), transport costs are reduced from 33658 to 33175. As expected, when considering some weight for the contribution of transport costs to the total reward function, the DNN trained with the PG algorithm learns to control the system in such a way that the final transport costs are reduced. This successful results are similar to the ones we obtained with Q-learning.

With regards to the *wellness* of shops, we see that in both cases most of the time shops are in the optimal zone. Note that when considering transport costs these costs are reduced from one simulation to the other, but the price to pay is a bit of wellness in the shops. Concretely, in average and for the simulated episode, shops are the 75.33% of the time in the optimal zone without transport costs, and this percentage reduces to the 73.33% when considering transport costs.

Table 6.2: Simulation 1) Information for the simulated episode shown in Figure 6.2. Transport costs: 33658. Trucks not delivering: 0. Total trucks sent: 42.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	6	0	5	0	5	14	16	—
Big trucks sent	0	1	0	20	5	4	26	—
Days empty	2	0	0	0	0	—	2	1.33%
Days in region $(0, b]$	0	0	1	0	4	—	5	3.33%
Days in region $(b, c]$	11	0	5	3	11	—	30	20%
Days in region $(c, e]$	17	30	24	27	15	—	113	75.33%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

Table 6.3: Simulation 2.1) Information for the simulated episode shown in Figure 6.3. Transport costs: 33175. Trucks not delivering: 0. Total trucks sent: 42.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	6	0	5	0	6	13	17	—
Big trucks sent	0	1	0	19	5	5	25	—
Days empty	1	0	1	0	0	—	2	1.33%
Days in region $(0, b]$	0	0	1	1	2	—	4	2.67%
Days in region $(b, c]$	11	0	5	6	12	—	34	22.67%
Days in region $(c, e]$	18	30	23	23	16	—	110	73.33%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

As it happened in Q-learning, the *trucks not delivering* is zero.

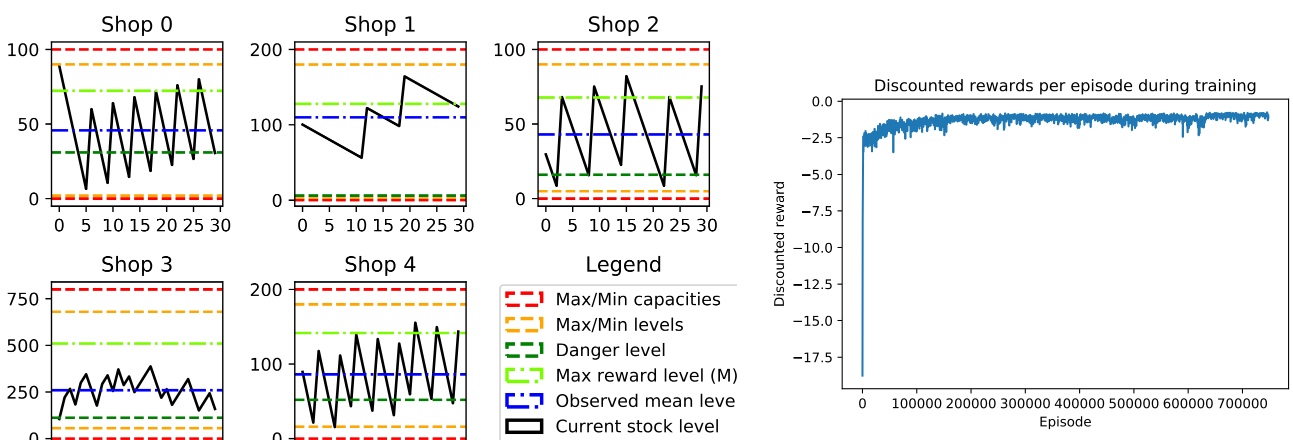
One may see that the average quantities over 100 episodes shown in the tables from the Appendix C.1 behave, in average, as we have just seen for a single simulated episode.

Finally we note that the discounted rewards per iteration (of 10 episodes each) during training, tends to converge and stabilize, hopefully, to a (maybe local) optimal solution but with quite high variance as it is typical in RL. Whereas in Q-learning we had a theorem that ensures convergence to the optimal Q values and thus to an optimal policy; now in PG we are dealing with an optimization problem, and therefore we can not assure, in principle, if the DNN trained has reached a global maxima (in terms of the discounted rewards).

Simulation 2.2 (deterministic, with higher contribution from transport costs than 2.1)

This simulation is the same as 2.1 but now we increase the transport coefficient $\mu_{\text{transport}}$ from 10^{-6} to 10^{-4} to give a higher contribution of transport costs.

Contrary to what we observed in the simulated episode in Q-learning when using $\mu_{\text{transport}} = 10^{-4}$, now transport costs have increased with respect to the previous two simulations (one extra truck is sent). One may expect them to decrease since now the role of transport costs is more important. However, in this particular episode, the DNN has found a solution where a higher *wellness* of shops compensates the increase of transport costs. Note that now the 86% of the time, shops are in the optimal zone (before it was about 75%); moreover, in the simulated episodes from the two previous simulations, some shops became empty in two days, whereas now, no shop becomes empty anymore.



(a) (b)

Figure 6.4: (a) Stocks of each shop in a single simulated episode of 30 time steps (deterministic with transport costs, $\mu_{\text{transport}} = 10^{-4}$). Transport costs: 34223. Trucks not delivering: 0. Total trucks sent: 43. (b) Discounted average rewards as a function of the number of training episodes.

If we look at the tables from Appendix C.2, we can see that the transport costs averaged over 100 episodes decrease when increasing the coefficient $\mu_{\text{transports}}$. When changing $\mu_{\text{transports}}$ from 0 to 10^{-6} the wellness decreases a bit in compensation to the reduction in transport costs. However, surprisingly, for $\mu_{\text{transports}} = 10^{-4}$, in average, not only transport costs are reduced, but also the wellness of shops improves. With these results we see that the process of parameter tuning is very important to achieve a good trade-off between costs and wellness of shops.

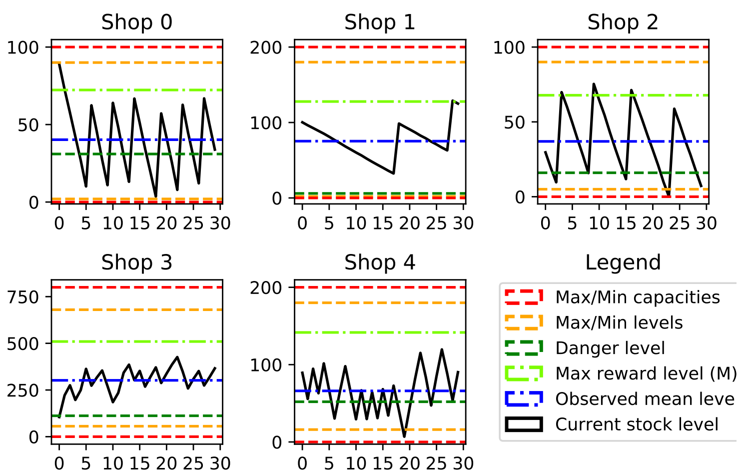
Table 6.4: Simulation 2.2) Information for the simulated episode shown in Figure 6.4. Transport costs: 34223. Trucks not delivering: 0. Total trucks sent: 43.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	
Small trucks sent	6	2	5	4	0	13	17	—
Big trucks sent	0	0	0	18	8	4	26	—
Days empty	0	0	0	0	0	—	0	0%
Days in region $(0, b]$	0	0	0	0	1	—	1	0.67%
Days in region $(b, c]$	9	0	4	1	6	—	20	13.33%
Days in region $(c, e]$	21	30	26	29	23	—	129	86%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

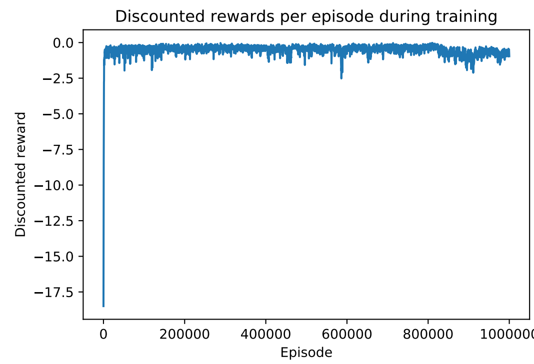
Stochastic simulations

Simulation 3 v.s. Simulation 4.1 (stochastic, 10% noise, without and with transport costs)

In figures 6.5 and 6.6 we show the evolution of the levels of stock of each shop and the discounted rewards per PG iteration during training for two stochastic simulations with a 10% of noise in the consumption rates. The former without transport costs contribution ($\mu_{\text{transport}} = 0$) and the latter with some transport contribution ($\mu_{\text{transport}} = 10^{-6}$). Similarly, in tables 6.5 and 6.6 there is summarized some relevant information about the simulated episode of these two simulations.

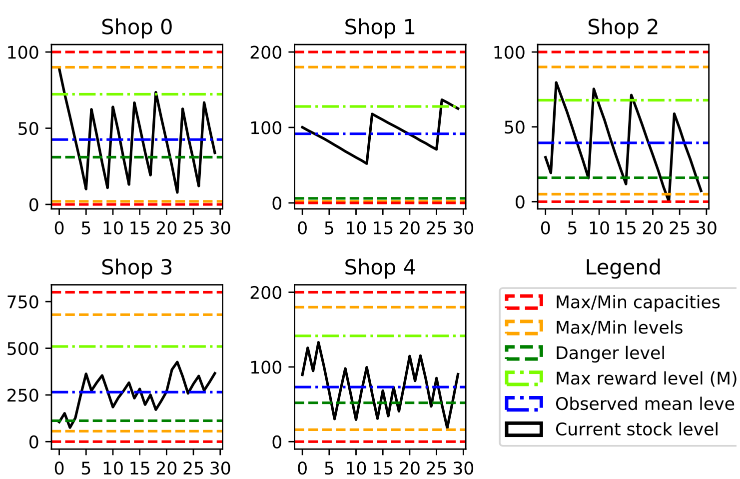


(a)

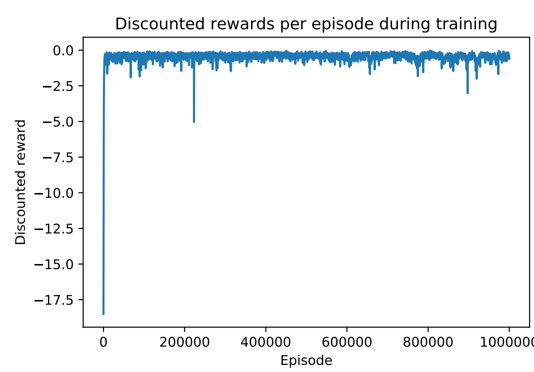


(b)

Figure 6.5: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, without transport costs, $\mu_{\text{transport}} = 0$). Transport costs: 34212 Trucks not delivering: 0. Total trucks sent: 49. (b) Discounted average rewards as a function of the number of training episodes.



(a)



(b)

Figure 6.6: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-6}$). Transport costs: 34223 Trucks not delivering: 0. Total trucks sent: 49. (b) Discounted average rewards as a function of the number of training episodes.

For this two first simulations, one without costs and the other with costs, we see a very little difference between the observed transport costs in a single episode. With $\mu_{\text{transport}} = 10^{-6}$, transport costs are 11 units greater than for $\mu_{\text{transport}} = 0$. This extra cost comes from the fact that one of the small trucks that in the first simulation goes to shop 3, in the second it goes to shop 1 instead. Since shop 1 is at a greater distance from the depot, then it is more expensive to travel there. However, in the particular episode we show in the figures above, this small increase in transport costs is compensated by a small improvement in the wellness of shops (the percentage of time that shops are in the optimal zone increases from about 80% to 82%).

If we look at tables C.12 and C.13, where we have some parameter values averaged over 100 simulated episodes, we see that transport costs decrease (in average) from 33419 to 33091 and the wellness of shops is more or less the same in both cases. Hence, we can say that our DNN trained with PG is able to control the shops in a proper way and reducing costs when adding some contribution of transport costs via $\mu_{\text{transport}} \neq 0$, even in a stochastic framework.

Table 6.5: Simulation 3) Information for the simulated episode shown in Figure 6.5. Transport costs: 34212. Trucks not delivering: 0. Total trucks sent: 49.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	6	1	5	4	14	0	30	—
Big trucks sent	0	0	0	19	0	11	19	—
Days empty	0	0	1	0	0	—	1	0.67%
Days in region $(0, b]$	0	0	0	0	1	—	1	0.67%
Days in region $(b, c]$	12	0	5	1	9	—	27	18%
Days in region $(c, e]$	18	30	24	29	20	—	121	80.67%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

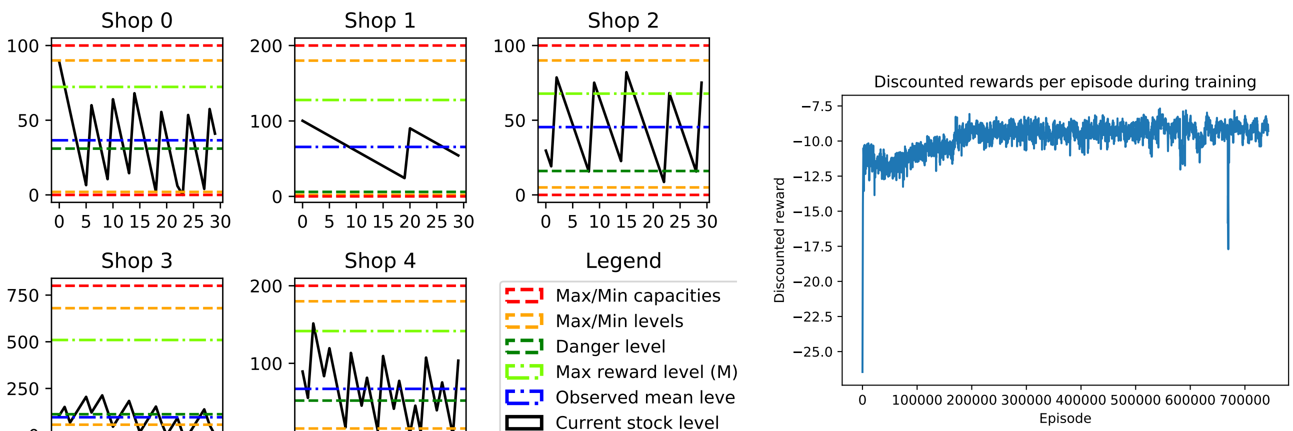
Table 6.6: Simulation 4.1) Information for the simulated episode shown in Figure 6.6. Transport costs: 34223. Trucks not delivering: 0. Total trucks sent: 49.

Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	6	2	5	3	14	0	30	—
Big trucks sent	0	0	0	19	0	11	19	—
Days empty	0	0	0	1	0	—	1	0.67%
Days in region $(0, b]$	0	0	0	0	1	—	1	0.67%
Days in region $(b, c]$	11	0	4	2	8	—	25	16.67%
Days in region $(c, e]$	19	30	25	28	21	—	123	82%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

Simulation 4.2 (stochastic, 10% noise, with higher contribution from transport costs than 4.1)

If we now increase the $\mu_{\text{transport}}$ coefficient in two orders of magnitude with respect to simulation 4.1), then in the simulated episode shown in Figure 6.7a costs are reduced from 34223 to 31073, and in turn the wellness is slightly reduced (for instance, the percentage of time in the optimal zone reduces from the 82% to the 70%).

Similarly as we did for the previous two simulations, if we now compare tables C.13 and C.14, we can see that transport costs are reduced considerably in average from 33091 to 29291 and the wellness of shops is also reduced, but not so much. For instance, the percentage in the optimal zone decreases from 86% to the 73%.



(a) (b)

Figure 6.7: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-4}$). Transport costs: 31073 Trucks not delivering: 0. Total trucks sent: 40. (b) Discounted average rewards as a function of the number of training episodes.

Table 6.7: Simulation 4.2) Information for the simulated episode shown in Figure 6.7. Transport costs: 31073. Trucks not delivering: 0. Total trucks sent: 40.

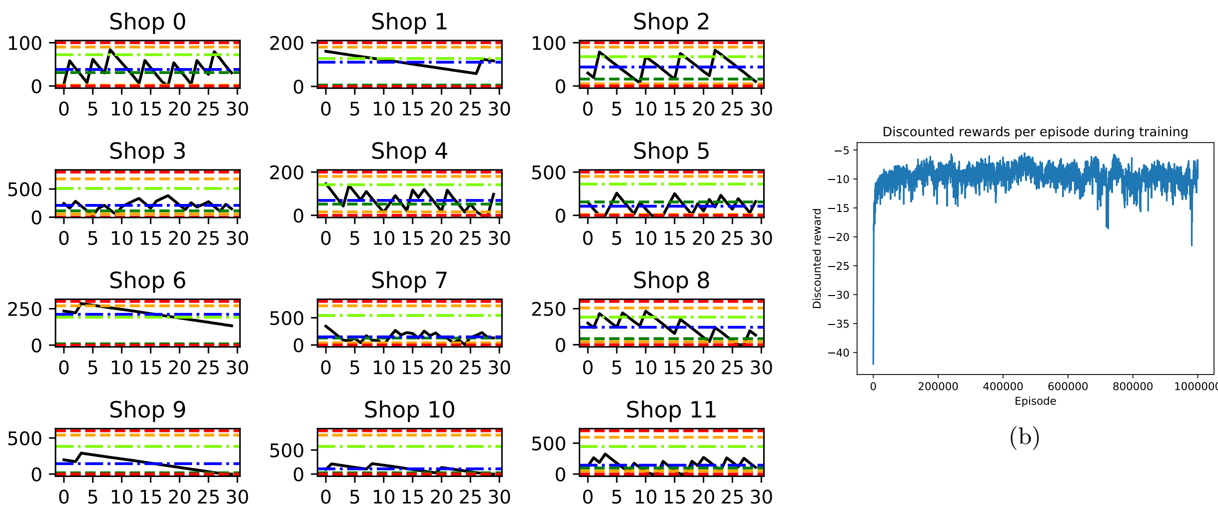
Parameter	Shop 0	Shop 1	Shop 2	Shop 3	Shop 4	Depot	Total	%
Small trucks sent	6	1	5	0	5	0	17	—
Big trucks sent	0	0	0	18	5	11	23	—
Days empty	1	0	0	3	0	—	4	2.67%
Days in region $(0, b]$	0	0	0	6	4	—	10	6.67%
Days in region $(b, c]$	11	0	3	9	9	—	32	21.33%
Days in region $(c, e]$	18	30	27	12	17	—	104	69.33%
Days in region $(e, 1]$	0	0	0	0	0	—	0	0%

In conclusion, we see that the parameters μ_i that appear in front of each term of the reward function (3.15) (in our case we have fixed all except for $\mu_{\text{transport}}$), allow us to modify/control the trade-off between wellness of shops and the transport costs; similarly as what we saw for Q-learning.

Simulation 5 v.s. simulation 6

In figures 6.8 and 6.9 we show the evolution of the levels of stock of each of the 12 shops and the discounted rewards per PG iteration during training for two stochastic simulations with a 10% of noise in the consumption rates. The former without transport costs contribution ($\mu_{\text{transport}} = 0$) and the latter with some transport contribution ($\mu_{\text{transport}} = 10^{-6}$). Similarly, in tables 6.8 and 6.9 there is summarized some relevant information about the simulated episode of these two simulations.

Before we go to the results, it is important to remark that the ratio of trucks available per shop, i.e. the ratio $\frac{n}{k}$, was 2.5 for the system of 5 shops and 2 trucks considered so far, but it is 4.0 for the new system of 12 shops and 3 trucks that we consider now. Thus, it might be more harder to the learnt policy to manage trucks in such a way all shops are maintained alive and with a great wellness. However, we are going to see that the results are fairly good.



(b)

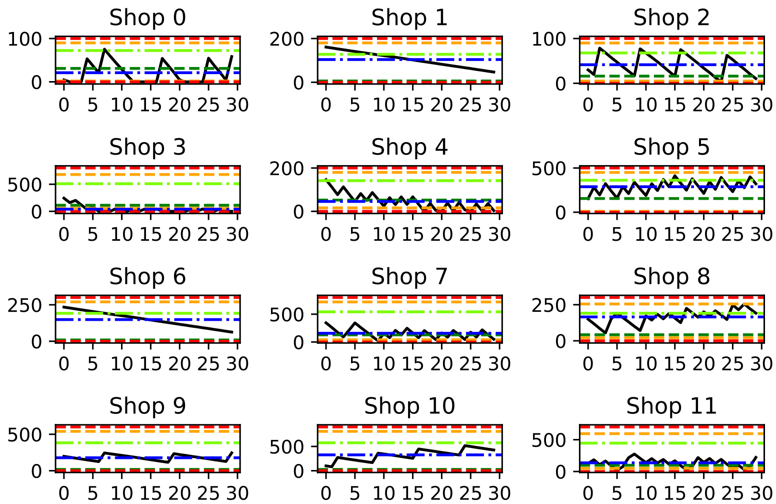
(a)

Figure 6.8: $n = 12, k = 3$: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, without transport costs, $\mu_{\text{transport}} = 0$). (b) Discounted average rewards as a function of the number of training episodes.

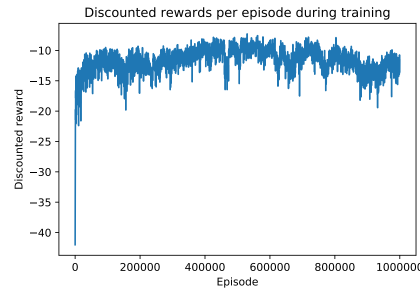
Lets compare now the two simulated episodes. The effects of considering the contribution of transport costs are a slight reduction of the overall wellness of shops - e.g. shops are more days empty and the time in the optimal zone is a bit smaller - but this is compensated by a reduction of the total transport costs (from 81503 to 79480).

At first sight one may think that the reduction of costs is due to the fact that with transport costs contribution, there are sent 2 trucks less than without. However, note that if we subtract the trucks not delivering from the total number of trucks sent in both simulations, we see that the total number of trucks that are in fact delivering product when going to a shop is the same (75 trucks in both cases).

Thus, in this case the reduction of transport costs is not due to a reduction in the number of trucks sent (as it usually happened for the simulations with 5 shops and 2 trucks), but it is due to the fact that the DNN when trained with transport costs contribution has learnt to sent the same number of (effective) trucks but in such a way that the total transport costs are reduced.



(a)



(b)

Figure 6.9: $n = 12, k = 3$: (a) Stocks of each shop in a single simulated episode of 30 time steps (stochastic, 10% noise, with transport costs, $\mu_{\text{transport}} = 10^{-6}$). (b) Discounted average rewards as a function of the number of training episodes.

Table 6.8: Simulation 5) Information for the simulated episode shown in Figure 6.8. Transport costs: 81503. Trucks not delivering: 5. Total trucks sent: 80.

Parameter	Total	%
Small trucks sent	30	—
Medium trucks sent	30	—
Big trucks sent	20	—
Days empty	18	5%
Days in region $(0, b]$	10	2.78%
Days in region $(b, c]$	54	15%
Days in region $(c, e]$	275	76.39%
Days in region $(e, 1]$	3	0.83%

Table 6.9: Simulation 6) Information for the simulated episode shown in Figure 6.9. Transport costs: 79480. Trucks not delivering: 3. Total trucks sent: 78.

Parameter	Total	%
Small trucks sent	30	—
Medium trucks sent	30	—
Big trucks sent	18	—
Days empty	31	8.61%
Days in region $(0, b]$	20	5.56%
Days in region $(b, c]$	40	11.11%
Days in region $(c, e]$	269	74.72%
Days in region $(e, 1]$	0	0%

Similar conclusions can be obtained if one analyses the parameter values from tables C.15 and C.16 in the Appendix, where we have averaged the results over 100 simulated episodes. As expected, transport costs are reduced, in average, and in compensation the wellness of shops decreases a bit.

Chapter 7

Conclusions and Future Work

In this chapter we sum up the main conclusions of the work done in this thesis, comparing classical and deep reinforcement learning results, as well as proposing immediate extensions of this work.

7.1 Classical v.s. Deep Reinforcement Learning

At the end of chapters 4 and 6 we showed simulations coming from applying Q-learning and Policy Gradient algorithms, respectively, to solve an optimization problem in the framework of Product Delivery. In the former case, we were dealing with dictionaries (or equivalently, tabular functions), whereas in the latter we used Deep Neural Networks.

If we refresh the results we presented in the respective sections 4.3 and 6.2.1, we may think that the results obtained in Q-learning for the system of 5 shops and 2 trucks are better than for PG in terms of wellness and else. However, we have to be aware of the fact that the initial states of the system in the simulations from each chapter were different. Thus, we cannot do a direct comparative, but what we can say for sure is that with both algorithms we obtained very satisfactory results.

However, the scalability of Q-learning to a large product delivery system in terms of the number of shops n and trucks k is very bad. If we remember equations (3.13) and (3.14), the number of possible states is d^n, where d is the number of parts in which we discretize the level of stock of shops; and the number of actions can be up to $(n+1)^k$. Thus, the dictionary where we store the Q-values for each state-action pair $(s,a) \in \mathcal{S} \times \mathcal{A}$, needs to have about $d^n \times (n+1)^k$ keys, a value that grows exponentially with the number of trucks and shops.

On the contrary, in the PG approach, the dimensionality of $\mathcal{S}$ is not important, since there is no need to discretize the levels of stocks of each shop. We just use the exact level of stock of each shop as input to a DNN. In this case what limits scalability is the total number of possible actions, since it is equal to the number of neurons in the output layer. The problem is that an order of 10^3 output neurons or more, would be impractical to train in a plausible amount of time. That is the reason for which we did not consider more than 12 shops and 3 trucks - in that case the total number of possible actions is up to $13^3 = 2197$, and increasing the number of trucks in one unit, the order of magnitude of output neurons also increases in one unit.

Since the scalability is now limit by the number of output neurons $\Lambda = (n+1)^k$, in the next section we propose a possible alternative of the current PG approach that would alleviate the scalability from $\Lambda = (n+1)^k$ to $\Lambda = (n+1) \cdot k$, i.e., from exponential to linear scalability on n and k.

7.2 Furtherwork

More scalable “Deep Policy Gradient” for product delivery

In order to have a better scalable approach in a DRL context, we propose a new DNN architecture different from the one that was shown in Figure 6.1. The new DNN structure we propose is the one shown in Figure 7.1. The idea now is to have a softmax layer for each truck so that we would have kind of a marginal probability distribution for each truck (one coming from each softmax layer), and from them we would be able to decide what to do with each truck (either send it to a shop or make it stay at the depot).

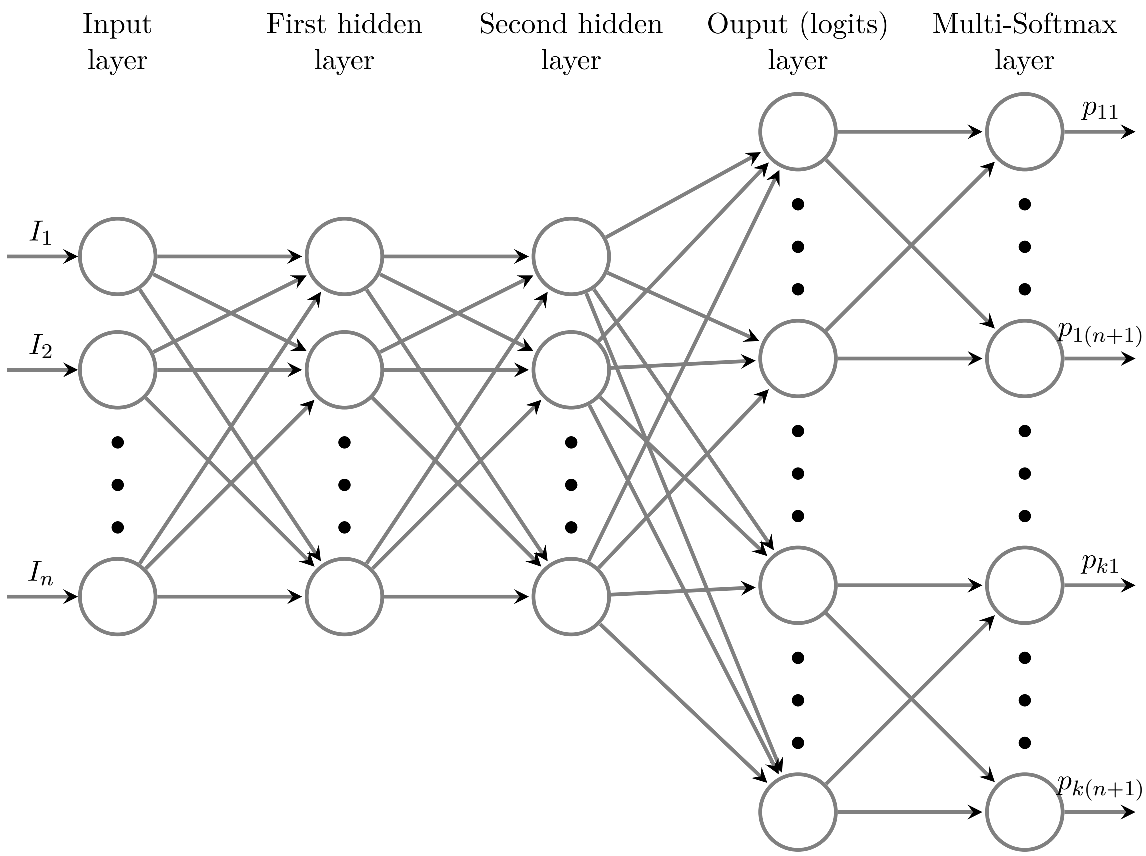


Figure 7.1: Deep Neural Network structure for an alternative “Policy Gradient” approach: one softmax layer per truck with one neuron per shop each. There are shown two hidden layers, but in general we may consider many more.

The main problem now would be how to train this proposed DNN. If we were able to obtain an algorithm that made this DNN learn a good policy, and to obtain similar good results as we obtained with Q-learning and Policy Gradient, now the number of output neurons would be $\Lambda = (n+1) \cdot k$, so that it would scale linearly, and not exponentially, with the number of trucks and shops.

The first approach that may come to our minds would be to use Policy Gradient as we did in chapter 6. However, in order to apply it we need our DNN to be a stochastic policy for actions given states , i.e., some $\pi_\theta(a|s)$, which is a probability distribution over actions given states. For the proposed approach, if we write an action as $a = (a_1, ..., a_k)$, where a_i is the action that the i-th truck makes, then the DNN from Figure 7.1 can be thought as a function whose outputs are a tuple of the form

$(\pi_{\theta_1}(a_1|s), ..., \pi_{\theta_i}(a_k|s))$, where π_{θ_i} is in fact a probability distribution over "single truck actions" given states (each softmax layer leads to a probability distribution, by definition).

We now show that it is feasible to train such a DNN with PG algorithm if there is independence between the actions performed by each truck. In order to formalize this, consider an action (random variable) decomposing as $A_t = (A_{t1}, ..., A_{tk})$ and which take values of the form $a = (a_1, ..., a_k)$. If all the A_{ti} are independent from each other, then

$$\begin{aligned}\pi_\theta(a|s) &= P\left((A_{t1}, ..., A_{tk}) = (a_1, ..., a_k)\middle| S_t = s\right)\\ &= \prod_{i=1}^{k} P\left(A_{ti} = a_i \middle| S_t = s\right)\\ &= \prod_{i=1}^{k} \pi_{\theta_i}(a_i|s).\end{aligned}$$

In particular, we would have

$$\log \pi_\theta(a|s) = \sum_{i=1}^{k} \log \pi_{\theta_i}(a_i|s). \tag{7.1}$$

Then, inserting equation (7.1) into equation (6.5) we have that the expression for estimating the gradient of the cost function according to PG Algorithm 5 would be

$$\nabla_\theta J_\theta \approx \sum_{i=1}^{k} \left\{ \frac{1}{N} \sum_{j=1}^{N} \left[\left(\sum_{t=0}^{\tau-1} \gamma^t r_t^j \right) \left(\sum_{t=0}^{\tau-1} \nabla_\theta \log \pi_{\theta_i}(a_{it}^j | s_t^j) \right) \right] \right\} := \sum_{i=1}^{k} \nabla J_{\theta,i}^j, \tag{7.2}$$

where each term $\nabla J_{\theta,i}^j$ would be a gradient of the cost function coming from each of the softmax layers (one for each truck $i \in \{1, ..., k\}$).

In section 6.2 we obtained the equation (6.7) that related the crossentropy with a term of the form $\log \pi_\theta(a|s)$ needed to compute the estimation of the gradient ∇J_θ. Similarly, we could follow the same reasoning and arrive to the following expression for each $\nabla J_{\theta,l}^j$,

$$\nabla_\theta H(\hat{a}_{lt}^j, p_\theta) = \sum_{i=1}^{n+1} (\hat{a}_{lt}^j)_i \nabla_\theta \log \pi_{\theta_i}((a_l)_i | s_t^j). \tag{7.3}$$

for $l \in \{1, ..., k\}$. Mejorar notación y 1 o 2 lineas

Hence, we would be approximating the whole gradient as the sum of gradients, which in turn is equivalent to a sum of crossentropies, one coming from each of the softmax layers. Intuitively, this would update the weights of the DNN policy in the direction .... no se como estruct aquí

Shared unloads and connections between shops

So far we have been working with a system where trucks can go to at most one shop every day and that when doing so they must deliver all its product to the visited shop. The latter assumption restricts ourselves to cases where all shops have a stock capacity greater (or equal) than the capacity of every truck (remember that one of our hard restrictions was that a truck that leaves the depot fully loaded must return empty).

The next step would be to generalize to systems of shops and trucks where there were possibly some shops smaller - in terms of product capacity - than some of the trucks. To do so, we would need to include connections between shops (and not only connections between every shop and the depot) to allow a truck go from one shop to another in the same day. Similarly, we would need to reconsider the way we simulate our PD system in order to allow *shared unloads* and the different possible quantities of product that a truck delivers in each shop.

Note that this generalization is impractical for Q-learning, since it will make explode even more the number of possible states and actions in the new system. Thus, the new approach should be thought to be implemented in a framework of Deep Reinforcement Learning.

Alternative definition of states (complex approach)

For the simulations done so far we followed the “simple” approach we presented in section 3.2.1. A further final step would be to work with the more complex approach we considered in section 3.2.2. The new approach would be useful if we had a model capable of predicting the consumptions of stock each day and for each shop, and then take profit of this information in the simulations that are used to train the model (in our case, a DNN).

The first alternative would be to use as inputs of a DNN not only the current level of stock of each shop, but also the predictions for consumption rates for some future days (the next day, two days to the future, or more). However, although we were able to solve the scalability problem encountered for the number of neurons in the output layer when using PG, now a new problem of scalability in the input layer would appear. However, we could engine more sophisticate DNN architectures, by combining several Deep Neural Networks while maintaining one of them whose inputs were only the stocks of each shop.

Then, hopefully, this extension would approach our model to the reality and could probably be industrialized and used by companies interested in a Product Delivery framework.

Bibliography

[1] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, September 2013.

[2] Michael J. Frank, Lauren C. Seeberger, and Randall C. O'Reilly. By carrot or by stick: Cognitive reinforcement learning in parkinsonism. *Science*, 306(5703):1940–1943, 2004.

[3] Zhao Yufan, Kosorok Michael R., and Zeng Donglin. Reinforcement learning design for cancer clinical trials. *Statistics in Medicine*, 28(26):3294–3315.

[4] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, Siddhartha Sen, and Alex Slivkins. A multiworld testing decision service. *CoRR*, abs/1606.03966, 2016.

[5] Han Cai, Kan Ren, Weinan Zhang, Kleanthis Malialis, Jun Wang, Yong Yu, and Defeng Guo. Real-time bidding by reinforcement learning in display advertising. *CoRR*, abs/1701.02490, 2017.

[6] Naoki Abe, Naval Verma, Chid Apte, and Robert Schroko. Cross channel optimized marketing by reinforcement learning. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 767–772, New York, NY, USA, 2004. ACM.

[7] Francesco Bertoluzzo and Marco Corazza. Reinforcement learning for automated financial trading: Basics and applications. In Simone Bassis, Anna Esposito, and Francesco Carlo Morabito, editors, *Recent Advances of Neural Network Models and Applications*, pages 197–213, Cham, 2014. Springer International Publishing.

[8] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization. *CoRR*, abs/1705.04304, 2017.

[9] Bhuwan Dhingra, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng. Towards end-to-end reinforcement learning of dialogue agents for information access. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. ACL – Association for Computational Linguistics, July 2017.

[10] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[11] W. B. Powell, A. George, B. Bouzaiene-Ayari, and H. P. Simao. Approximate dynamic programming for high dimensional resource allocation problems. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 5, pages 2989–2994 vol. 5, July 2005.

[12] Scott Proper and Prasad Tadepalli. Scaling model-based average-reward reinforcement learning for product delivery. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 735–742, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] Daniel Salgado Rojo. Master's thesis. `https://github.com/dsalgador/master-thesis`, 2018.

[14] Daniel Salgado Rojo. Product delivery gym environment. `https://github.com/dsalgador/gym-pdsystem`, 2018.

[15] Wikipedia contributors. Arthur samuel — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Arthur_Samuel&oldid=845052366`, 2018. [Online; accessed 11-June-2018].

[16] Pinterest. Venn diagram for machine learning subfields. `https://i.pinimg.com/564x/4a/e7/de/4ae7de2f2157e8e36f2b563060694a06.jpg`, 2017.

[17] Wikipedia contributors. Timeline of artificial intelligence — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Timeline_of_artificial_intelligence&oldid=843741084`, 2018. [Online; accessed 11-June-2018].

[18] Wikipedia contributors. Timeline of machine learning — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Timeline_of_machine_learning&oldid=843292382`, 2018. [Online; accessed 11-June-2018].

[19] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[20] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Sebastopol, CA, 6 edition, 2017.

[21] Deep Mind. The story of alpha go, 2015.

[22] Wikipedia contributors. History of go — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=History_of_Go&oldid=833729414`, 2018. [Online; accessed 15-June-2018].

[23] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[24] Pinterest. Truck image. `https://www.pinterest.es/pin/521925044288845297/`, 2017.

[25] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[26] Craig Boutilier, Thomas L. Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *CoRR*, abs/1105.5460, 2011.

[27] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[28] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[29] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[30] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[31] Wikipedia contributors. Markov decision process — wikipedia, the free encyclopedia, 2018. [Online; accessed 18-March-2018].

[32] Wikipedia contributors. Reinforcement learning — wikipedia, the free encyclopedia, 2018. [Online; accessed 18-March-2018].

[33] Mostafa Samir. Building a tic-tac-toe ai with javascript. `https://mostafa-samir.github.io/Tic-Tac-Toe-AI/`, 2015.

[34] Wikipedia contributors. Q-learning — wikipedia, the free encyclopedia, 2018. [Online; accessed 20-March-2018].

[35] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[36] David Silver. Ucl course on rl: Lecture 2. University Lecture, 2015.

[37] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[38] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[39] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[40] R. S. Sutton and A. G. Barto. Time-derivative models of Pavlovian reinforcement. In J. W. Moore and M. Gabriel, editors, *Learning and Computational Neuroscience*, pages 497–537. MIT Press, Cambridge, MA, 1990.

[41] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[42] Tokic M and Palm G. Value-difference based exploration: Adaptive control between epsilon-greedy and softmax. *Lecture Notes in Computer Science*, 7006, Springer, Berlin, Heidelberg, KI 2011.

[43] Eyal Even-Dar and Yishay Mansour. Convergence of optimistic and incremental q-learning. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 1499–1506, Cambridge, MA, USA, 2001. MIT Press.

[44] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[45] Francisco S. Melo. Convergence of q-learning: a simple proof. http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[46] Wikipedia contributors. Artificial neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=840077868, 2018. [Online; accessed 9-May-2018].

[47] Warren S. McCulloch and Walter Pitts. Neurocomputing: Foundations of research. chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988.

[48] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Texts in applied mathematics. Springer, 2000.

[49] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014.

[50] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points - online stochastic gradient for tensor decomposition. *CoRR*, abs/1503.02101, 2015.

[51] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[52] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[53] Michael A. Nielsen. Neural networks and deep learning, 2018.

[54] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[55] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017.

[56] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[59] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[60] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[61] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[62] David Silver. Ucl course on rl: Lecture 7. University Lecture, 2015.

[63] Sergey Levine. Deep reinforcement learning: Policy gradients lecture. University Lecture, 2017.

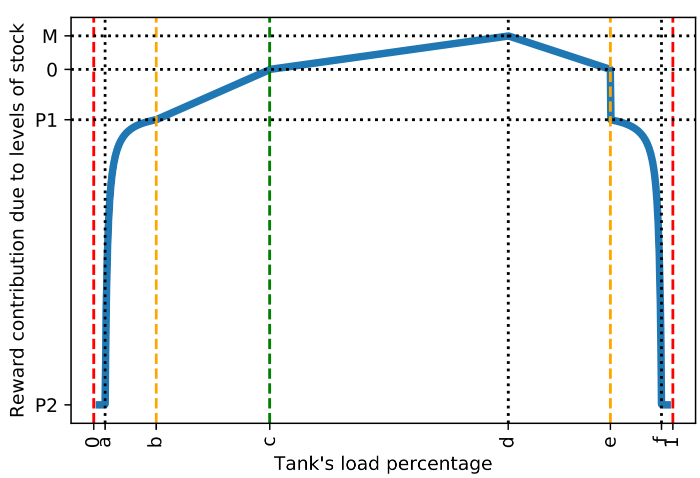
Appendix A

Derivation of $J^{(i)}_{\text{levels}}$

In section 3.2.4 we presented a contribution of the levels of stock to the total function of rewards, for each shop i which was a term denoted $J^{(i)}_{\text{levels}}$ and defined by

$$J^{(i)}_{\text{levels}}(x) = \begin{cases} P_2 & \text{if } x < a \text{ or } x > f \\ C_{ab} \exp\left(\frac{\alpha_{ab}}{x}\right) & \text{if } a \leq x \leq b \\ m_{bc}x + n_{bc} & \text{if } b < x \leq c \\ m_{cd}x + n_{cd} & \text{if } c < x \leq d \\ m_{de}x + n_{de} & \text{if } d < x \leq e \\ C_{ef} \exp\left(\frac{\alpha_{ef}}{1-x}\right) & \text{if } e < x \leq f. \end{cases}$$

In this appendix we derive the expressions for the constants C_{ij}, m_{ij}, n_{ij} and α_{ij} and understand the meaning of the percentages a, b, c, d, e and f, as well as the constants P_1, P_2 and M from Figure 3.3 which was the following plot:



b, c and d are the percentages corresponding to the minimum, the danger and the maximum levels of stock of a given shop, respectively.

Between the minimum capacity of the shop and the minimum level (i.e., from 0 to b), we want to impose a negative reward that decreases exponentially from some threshold P_1 as the level of stock tends to 0, in order to penalize more and more. Due to numerical reasons, we set a negative threshold P2 so that from 0 to a small percentage $a < b$ the $J^{(i)}_{\text{levels}}$ value is P_2. We choose a to be $a = b/10$. The reason of taking this threshold is that a function of the form $-\exp(x^{-1})$ goes to $-\infty$ as x tends to 0.

Similarly, between the maximum level of stock and the maximum capacity (i.e, from e to 1), we want impose a negative reward that decreases exponentially from the threshold P_1 as the level of stock tends to 1. Again, due to numerical reasons, we set a negative threshold P_2 so that from 1 to some $f < 1$ the $J^{(i)}_{\text{levels}}$ value is P_2. We choose f to be $f = 1 - \frac{1-e}{10}$. The reason of taking this threshold is that a function of the form $-\exp\left(\frac{1}{1-x}\right)$ goes to $-\infty$ as x tends to 1 [1].

From the minimum level of stock to the danger level (i.e., from b to c) we consider a negative reward in b that starts from P_1 and increases linearly until it reaches 0.

From the danger level to the maximum level of stock we should consider positive reward, since it is the desired stock zone. At this point we define an intermediate level given by some percentage d that we choose as $d = p0 * e + (1 - p0) * c$ for some $p_0 \in (0.5, 1)$. The idea is that this intermediate level would be above the danger level and below the maximum level of stock, since we neither want a shop to be very close to the danger level nor to the maximum level; we prefer it to be at some intermediate level. Then from c to d, the reward is considered to increase linearly until it reaches a threshold value $M > 0$ at $x = d$. From d to e the reward is considered to decrease linearly until it reaches 0.

Now we are ready to find the mathematical expressions for C_{ij}, m_{ij}, n_{ij} and α_{ij} as a function of all the other parameters.

$\underline{C_{ab}, \alpha_{ab}}$

The boundary conditions in $x = a$ and $x = b$ are the following:

$$P_2 = C_{ab} \exp\left(\frac{\alpha_{ab}}{a}\right) \tag{A.1}$$

$$P_1 = C_{ab} \exp\left(\frac{\alpha_{ab}}{b}\right) \tag{A.2}$$

Dividing (A.1) by (A.2),

$$\frac{P_2}{P_1} = \exp\left(\alpha_{ab}\left(\frac{1}{a} - \frac{1}{b}\right)\right), \tag{A.3}$$

which leads to

$$\boxed{\alpha_{ab} = \frac{1}{\frac{1}{a} - \frac{1}{b}} \log\frac{P_2}{P_1}}. \tag{A.4}$$

Now from (A.2),

$$P_1 = C_{ab}\exp\left(\left(\frac{\frac{1}{b}}{\frac{1}{a} - \frac{1}{b}}\right)\log\frac{P_2}{P_1}\right) = C_{ab}\left(\frac{P_2}{P_1}\right)^{\frac{1/b}{1/a-1/b}} = C_{ab}\left(\frac{P_2}{P_1}\right)^{\frac{1}{\frac{b}{a}-1}} \tag{A.5}$$

$$\boxed{C_{ab} = P_1\left(\frac{P_2}{P_1}\right)^{\frac{1}{1-\frac{b}{a}}}}. \tag{A.6}$$

[1]Note that we may choose different thresholds P_1 and P_2, for every exponential, but we use the same in order to have less parameters in the model.

$\underline{m_{bc}, n_{bc}}$

The boundary conditions in b and c are

$$m_{bc}b + n_{bc} = P_1 \tag{A.7}$$
$$m_{bc}c + n_{bc} = 0 \tag{A.8}$$

Considering (A.8)-(A.7) we have

$$m_{bc}(c-b) = -P_1, \tag{A.9}$$

$$\boxed{m_{bc} = \frac{-P_1}{c-b}}. \tag{A.10}$$

Now from (A.8) and (A.10),

$$\boxed{n_{bc} = P_1\frac{c}{c-b}} \tag{A.11}$$

$\underline{m_{cd}, n_{cd}}$

The boundary conditions in c and d are

$$m_{cd}c + n_{cd} = 0 \tag{A.12}$$
$$m_{cd}d + n_{cd} = M \tag{A.13}$$

Considering (A.13)-(A.12) we have

$$m_{cd}(d-c) = M, \tag{A.14}$$

$$\boxed{m_{cd} = \frac{M}{d-c}}. \tag{A.15}$$

Now from (A.8) and (A.15),

$$\boxed{n_{cd} = -M\frac{c}{d-c}} \tag{A.16}$$

$\underline{m_{de}, n_{de}}$

The boundary conditions in c and d are

$$m_{de}d + n_{de} = M \tag{A.17}$$
$$m_{de}e + n_{de} = 0 \tag{A.18}$$

Considering (A.18)-(A.17) we have

$$m_{de}(e-d) = -M, \tag{A.19}$$

$$\boxed{m_{de} = -\frac{M}{e-d}}. \tag{A.20}$$

Now from (A.18) and (A.20),

$$\boxed{n_{de} = M\frac{e}{e-d}} \tag{A.21}$$

$\underline{C_{ef}, \alpha_{ef}}$

The boundary conditions in $x = e$ and $x = f$ are the following:

$$P_1 = C_{ef} \exp\left(\frac{\alpha_{ef}}{e}\right) \tag{A.22}$$

$$P_2 = C_{ef} \exp\left(\frac{\alpha_{ef}}{f}\right) \tag{A.23}$$

Dividing (A.23) by (A.22),

$$\frac{P_2}{P_1} = \exp\left(\alpha_{ef}\left(\frac{1}{1-f} - \frac{1}{1-e}\right)\right), \tag{A.24}$$

which leads to

$$\boxed{\alpha_{ef} = \frac{1}{\frac{1}{1-f} - \frac{1}{1-e}} \log \frac{P_2}{P_1}}. \tag{A.25}$$

Now from (A.22),

$$P_1 = C_{ef}\exp\left(\left(\frac{\frac{1}{1-e}}{\frac{1}{1-f} - \frac{1}{1-e}}\right) \log \frac{P_2}{P_1}\right) = C_{ef}\left(\frac{P_2}{P_1}\right)^{\frac{1/(1-e)}{1/(1-f)-1/(1-e)}} = C_{ef}\left(\frac{P_2}{P_1}\right)^{\frac{1}{\frac{1-e}{1-f}-1}} \tag{A.26}$$

$$\boxed{C_{ef} = P_1\left(\frac{P_2}{P_1}\right)^{\frac{1}{1-\frac{1-e}{1-f}}}}. \tag{A.27}$$

Appendix B

Default simulation parameter values

B.1 Model system parameters for Q-learning

In this appendix we present Table B.1 with the default parameters used to define the system of $n = 5$ shops and $k = 2$ trucks.

Table B.1: Default parameters for Q-learning simulations from Chapter 4.

Parameter(s)	Description	Value(s)
C_i	Maximum capacity of the shops	$[100, 200, 100, 800, 200]$
L_i	Maximum capacity of the trucks	$[70, 130]$
b	Minimum level (fraction of C_i)	$[0.02, 0.01, 0.05, 0.07, 0.08]$
c	Danger level (fraction of C_i)	$[0.31, 0.03, 0.16, 0.14, 0.26]$
e	Maximum level (fraction of C_i)	$[0.9, 0.9, 0.9, 0.85, 0.9]$
a	(see Appendix A)	$b/10$
f	(see Appendix A)	$1 - (1 - e)/10$
d	(see Appendix A)	$p_0 e + (1 - p_0)c$, $p_0 = 0.7$
w_{5i}	Weights from the depot to shop i (or to depot if $i = 5$)	$[32, 159, 162, 156, 156, 0]$
M	(see Appendix A)	10
P_1	(see Appendix A)	-10^3
P_2	(see Appendix A)	-10^6
μ_1	Reward coefficient for the costs contribution	0
C_{costs}	(see equation (3.10))	0.051591
$\mu_{\text{extra},1}$	Reward coefficient a “a truck not delivering” contribution	10^{-6}
μ_2	Reward coefficient for the levels of stock contribution	10^{-6}

B.2 Model system parameters for Policy Gradient

For PG simulations where the system is made of 5 shops and 2 trucks, the parameter values are the same than those from Table B.1 for the Q-learning simulations.

For simulations where there are considered $n = 12$ shops and $k = 3$ trucks the parameter values that differ from those of the case $n = 5$, $k = 2$ are shown in Table B.2.

Table B.2: Default parameters for Monte Carlo Policy Gradient simulations from Chapter 6.

Parameter(s)	Description	Value(s)
C_i	Maximum capacity of the shops	$[100, 200, 100, 800, 200, 500,$ $300, 800, 300, 600, 900, 700]$
L_i	Maximum capacity of the trucks	$[70, 130, 210]$
b	Minimum level (fraction of C_i)	$[0.02, 0.01, 0.05, 0.07, 0.08, 0.02,$ $0.01, 0.05, 0.07, 0.01, 0.01, 0.07]$
c	Danger level (fraction of C_i)	$[0.31, 0.03, 0.16, 0.14, 0.26, 0.31,$ $0.03, 0.16, 0.14, 0.03, 0.03, 0.14]$
e	Maximum level (fraction of C_i)	$[0.9, 0.9, 0.9, 0.85, 0.9, 0.9,$ $0.9, 0.9, 0.85, 0.9, 0.9, 0.85]$
w_{12i}	Weights from the depot to shop i (or to depot if $i = 12$)	$[32, 159, 162, 156, 156, 32,$ $159, 162, 156, 150, 150, 150, 0]$

Appendix C

Simulation tables

C.1 Q-learning simulations

Deterministic simulations

Table C.1: Average over 100 episodes for some simulation parameters. Simulation with id 216. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-78623	—
"Levels average" rewards	-234031	—
"Costs" average rewards	31048	—
Average total trucks sent	44.38	—
Average small trucks sent	25.84	—
Average big trucks sent	18.54	—
Average total trucks not delivering	0.0	—
Average n$^{\text{o}}$ of days a shop is empty	0.22	0.15%
Average n$^{\text{o}}$ of days a shop is in region $(0, b]$	0.15	0.1%
Average n$^{\text{o}}$ of days a shop is in region $(b, c]$	14.22	9.48%
Average n$^{\text{o}}$ of days a shop is in region $(c, e]$	134.32	89.55%
Average n$^{\text{o}}$ of days a shop is in region $(e, 1]$	1.09	0.73%

Table C.2: Average over 100 episodes for some simulation parameters. Simulation with id 217. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-156145	—
"Levels average" rewards	-804347	—
"Costs" average rewards	29285	—
Average total trucks sent	42.47	—
Average small trucks sent	25.18	—
Average big trucks sent	17.29	—
Average total trucks not delivering	0.09	—
Average n$^{\text{o}}$ of days a shop is empty	0.76	0.51%
Average n$^{\text{o}}$ of days a shop is in region $(0, b]$	0.7	0.47%
Average n$^{\text{o}}$ of days a shop is in region $(b, c]$	19.18	12.79%
Average n$^{\text{o}}$ of days a shop is in region $(c, e]$	128.3	85.53%
Average n$^{\text{o}}$ of days a shop is in region $(e, 1]$	1.06	0.71%

Table C.3: Average over 100 episodes for some simulation parameters. Simulation with id 218. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-210438	—
"Levels average" rewards	-933838	—
"Costs" average rewards	29013	—
Average total trucks sent	38.84	—
Average small trucks sent	18.03	—
Average big trucks sent	20.81	—
Average total trucks not delivering	0.07	—
Average n$^{\text{o}}$ of days a shop is empty	0.85	0.57%
Average n$^{\text{o}}$ of days a shop is in region $(0, b]$	1.41	0.94%
Average n$^{\text{o}}$ of days a shop is in region $(b, c]$	24.27	16.18%
Average n$^{\text{o}}$ of days a shop is in region $(c, e]$	122.42	81.61%
Average n$^{\text{o}}$ of days a shop is in region $(e, 1]$	1.05	0.7%

Table C.4: Average over 100 episodes for some simulation parameters. Simulation with id 222. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-6051478	—
"Levels average" rewards	-27700997	—
"Costs" average rewards	11328	—
Average total trucks sent	24.45	—
Average small trucks sent	23.46	—
Average big trucks sent	0.99	—
Average total trucks not delivering	0.01	—
Average n$^{\text{o}}$ of days a shop is empty	26.3	17.53%
Average n$^{\text{o}}$ of days a shop is in region $(0, b]$	4.06	2.71%
Average n$^{\text{o}}$ of days a shop is in region $(b, c]$	26.32	17.55%
Average n$^{\text{o}}$ of days a shop is in region $(c, e]$	92.26	61.51%
Average n$^{\text{o}}$ of days a shop is in region $(e, 1]$	1.06	0.71%

Stochastic simulations with 10% noise

Table C.5: Average over 100 episodes for some simulation parameters. Simulation with id 224. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-77146	—
"Levels average" rewards	-196451	—
"Costs" average rewards	30938	—
Average total trucks sent	44.14	—
Average small trucks sent	25.55	—
Average big trucks sent	18.6	—
Average total trucks not delivering	0.02	—
Average n° of days a shop is empty	0.19	0.13%
Average n° of days a shop is in region $(0, b]$	0.13	0.087%
Average n° of days a shop is in region $(b, c]$	14.16	9.44%
Average n° of days a shop is in region $(c, e]$	134.46	89.64%
Average n° of days a shop is in region $(e, 1]$	1.06	0.71%

Table C.6: Average over 100 episodes for some simulation parameters. Simulation with id 225. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-79457	—
"Levels average" rewards	-374889	—
"Costs" average rewards	29229	—
Average total trucks sent	42.37	—
Average small trucks sent	25.24	—
Average big trucks sent	17.13	—
Average total trucks not delivering	0.0	—
Average n° of days a shop is empty	0.33	0.22%
Average n° of days a shop is in region $(0, b]$	0.78	0.52%
Average n° of days a shop is in region $(b, c]$	19.24	12.83%
Average n° of days a shop is in region $(c, e]$	128.6	85.73%
Average n° of days a shop is in region $(e, 1]$	1.05	0.7%

Table C.7: Average over 100 episodes for some simulation parameters. Simulation with id 227. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-231898	—
"Levels average" rewards	-1033656	—
"Costs" average rewards	29009	—
Average total trucks sent	39.16	—
Average small trucks sent	18.75	—
Average big trucks sent	20.41	—
Average total trucks not delivering	0.03	—
Average nº of days a shop is empty	0.93	0.62%
Average nº of days a shop is in region $(0, b]$	1.42	0.95%
Average nº of days a shop is in region $(b, c]$	25	16.67%
Average nº of days a shop is in region $(c, e]$	121.6	81.07%
Average nº of days a shop is in region $(e, 1]$	1.05	0.7%

Table C.8: Average over 100 episodes for some simulation parameters. Simulation with id 228. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-6267245	—
"Levels average" rewards	-29690599	—
"Costs" average rewards	11669	—
Average total trucks sent	25.49	—
Average small trucks sent	25.02	—
Average big trucks sent	0.47	—
Average total trucks not delivering	0.01	—
Average n$^{\text{o}}$ of days a shop is empty	28.13	18.75%
Average n$^{\text{o}}$ of days a shop is in region $(0, b]$	5.2	3.47%
Average n$^{\text{o}}$ of days a shop is in region $(b, c]$	27.02	18.01%
Average n$^{\text{o}}$ of days a shop is in region $(c, e]$	88.6	59.07%
Average n$^{\text{o}}$ of days a shop is in region $(e, 1]$	1.05	0.7%

C.2 Policy Gradient simulations

Deterministic simulations

Table C.9: Average over 100 episodes for some simulation parameters. Simulation with id 316. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-844643	—
"Levels average" rewards	-1351352	—
"Costs" average rewards	32647	—
Average total trucks sent	42.43	—
Average small trucks sent	18.19	—
Average big trucks sent	24.24	—
Average total trucks not delivering	0.84	—
Average n^o of days a shop is empty	1.17	0.78%
Average n^o of days a shop is in region $(0, b]$	1.96	1.31%
Average n^o of days a shop is in region $(b, c]$	21.32	14.21%
Average n^o of days a shop is in region $(c, e]$	125.21	83.47%
Average n^o of days a shop is in region $(e, 1]$	0.34	0.23%

Table C.10: Average over 100 episodes for some simulation parameters. Simulation with id 317. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-966292	—
"Levels" average rewards	-1637648	—
"Costs" average rewards	32288	—
Average total trucks sent	42.56	—
Average small trucks sent	18.85	—
Average big trucks sent	23.71	—
Average total trucks not delivering	1.05	—
Average n^o of days a shop is empty	1.4	0.93%
Average n^o of days a shop is in region $(0, b]$	2.3	1.53%
Average n^o of days a shop is in region $(b, c]$	27	14.18%
Average n^o of days a shop is in region $(c, e]$	124.77	83.18%
Average n^o of days a shop is in region $(e, 1]$	0.26	0.17%

Table C.11: Average over 100 episodes for some simulation parameters. Simulation with id 319. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-224097	—
"Levels" average rewards	-361205	—
"Costs" average rewards	31815	—
Average total trucks sent	40.68	—
Average small trucks sent	16.17	—
Average big trucks sent	24.51	—
Average total trucks not delivering	0.09	—
Average n^{o} of days a shop is empty	0.27	0.18%
Average n^{o} of days a shop is in region $(0, b]$	1.49	0.99%
Average n^{o} of days a shop is in region $(b, c]$	21.7	14.47%
Average n^{o} of days a shop is in region $(c, e]$	126.45	84.3%
Average n^{o} of days a shop is in region $(e, 1]$	0.09	0.06%

Stochastic simulations with 10% noise

Table C.12: Average over 100 episodes for some simulation parameters. Simulation with id 324. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-532375	—
"Levels" average rewards	-609721	—
"Costs" average rewards	33419	—
Average total trucks sent	48.62	—
Average small trucks sent	30.0	—
Average big trucks sent	18.62	—
Average total trucks not delivering	0.57	—
Average nº of days a shop is empty	0.51	0.34%
Average nº of days a shop is in region $(0, b]$	1.16	0.77%
Average nº of days a shop is in region $(b, c]$	18.01	12%
Average nº of days a shop is in region $(c, e]$	130.18	86.79%
Average nº of days a shop is in region $(e, 1]$	0.14	0.09%

Table C.13: Average over 100 episodes for some simulation parameters. Simulation with id 325. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-404402	—
"Levels" average rewards	-577390	—
"Costs" average rewards	33091	—
Average total trucks sent	48.43	—
Average small trucks sent	30	—
Average big trucks sent	18.43	—
Average total trucks not delivering	0.46	—
Average nº of days a shop is empty	0.46	0.31%
Average nº of days a shop is in region $(0, b]$	1.38	0.92%
Average nº of days a shop is in region $(b, c]$	17.59	11.73%
Average nº of days a shop is in region $(c, e]$	130.13	86.75%
Average nº of days a shop is in region $(e, 1]$	0.44	0.29%

Table C.14: Average over 100 episodes for some simulation parameters. Simulation with id 328. Reward values using $\mu_i = 1$ for all i.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-1442292	—
“Levels” average rewards	-4783519	—
“Costs” average rewards	29291	—
Average total trucks sent	39.55	—
Average small trucks sent	18.68	—
Average big trucks sent	20.87	—
Average total trucks not delivering	0.7	—
Average n^{o} of days a shop is empty	4.13	2.75%
Average n^{o} of days a shop is in region $(0, b]$	7.68	5.12%
Average n^{o} of days a shop is in region $(b, c]$	28.82	19.21%
Average n^{o} of days a shop is in region $(c, e]$	109.36	72.91%
Average n^{o} of days a shop is in region $(e, 1]$	0.01	0.007%

Table C.15: Simulation 5) Information for the simulated episode shown in Figure 6.8. $n = 12$ shops, $k = 3$ trucks.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-8449398	—
"Levels" average rewards	-26506695	—
"Costs" average rewards	81680	—
Average total trucks sent	79.37	—
Average small trucks sent	30	—
Average medium trucks sent	30	—
Average big trucks sent	19.37	—
Average total trucks not delivering	4.0	—
Average n° of days a shop is empty	24.11	6.7%
Average n° of days a shop is in region $(0, b]$	9.72	2.7%
Average n° of days a shop is in region $(b, c]$	48.28	13.41%
Average n° of days a shop is in region $(c, e]$	277.34	77.04%
Average n° of days a shop is in region $(e, 1]$	0.55	0.15%

Table C.16: Simulation 6) Information for the simulated episode shown in Figure 6.9. $n = 12$ shops, $k = 3$ trucks.

Parameter	Average over 100 episodes	Percentage (%)
Total average discounted rewards	-9700768	—
"Levels" average rewards	-31959941	—
"Costs" average rewards	79351	—
Average total trucks sent	77.53	—
Average small trucks sent	30	—
Average medium trucks sent	30	—
Average big trucks sent	17.53	—
Average total trucks not delivering	3.64	—
Average n° of days a shop is empty	28.74	7.98%
Average n° of days a shop is in region $(0, b]$	19.21	5.34%
Average n° of days a shop is in region $(b, c]$	44.85	12.46%
Average n° of days a shop is in region $(c, e]$	265.64	73.79%
Average n° of days a shop is in region $(e, 1]$	1.56	0.43%