Solving a Product Delivery Problem with Reinforcement Learning and Deep Neural Networks

Dani Salgado

Advisor: Toni Lozano

Master's thesis

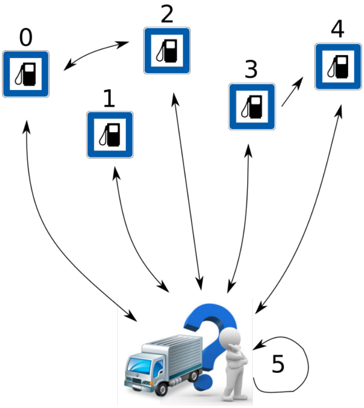
Universitat Autònoma de Barcelona

GRUPO AIA

Outline

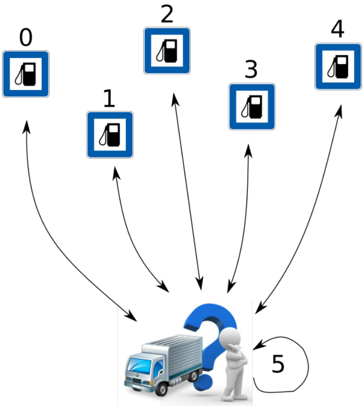
Problem statement

- Our client owns a chain of stores specialized in a particular product (e.g. Gas Natural), and he/she pays to a transport company to bring their product from a depot (e.g. Barcelona's dock) to the shops.
- The goal is to minimize the amount of money that our client has to pay to the transport company, but ensuring that there is always gas available for costumers (among other possible constraints).



Initial assumptions

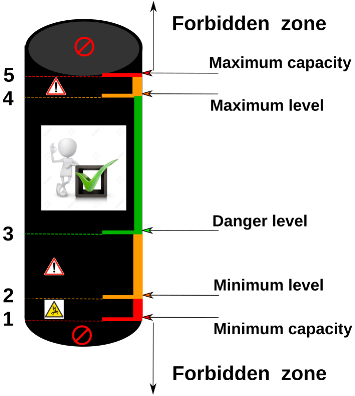
1. We have a system of n shops and k trucks.
2. We assume **single unloads** (a truck can only go to one shop everyday).
3. Trucks leave the depot fully loaded and return completely empty.
4. The price that our client has to pay to the transport company is given by some cost function J_{costs} which may depend on the distance travelled by the trucks, the amount of product delivered, holidays, etc.



Levels of stock

The company has a criteria to say if the current level of stock in a given shop is “good or bad”.

- **Maximum (resp. minimum) capacity**: it is physically impossible or very dangerous to have a stock **above (resp. below)** this level.
- **Maximum level**: desired maximum stock.
- **Danger level**: expected consumption in 36 hours.
- **Minimum level**: expected consumption in 12 hours.



Final goal

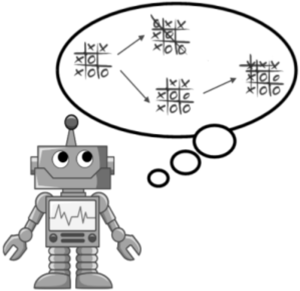
Our product delivery problem becomes an **optimization problem** that needs to balance transport company costs (J_{costs}) and the levels of stock of the shops (we need some "J_{levels}").

Thus, there will be a **trade-off between minimizing transport company costs and maximizing the *wellness* of the shops in terms of stock**.

Outline

Reinforcement learning: states and actions

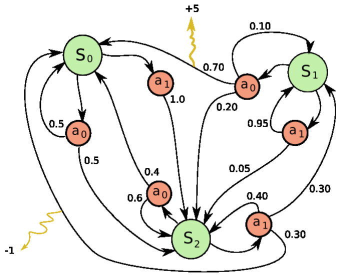
- **Reinforcement learning** (RL) comprise techniques used in **Artificial intelligence** (AI) and nowadays it is popular in applications such as training *bots* in video-games (e.g. chess, tictactoe, Alpha GO,....), robots, music personalization (YouTube), marketing,...
- We assume to have an **agent** (a robot, an algorithm) living in an **environment** (e.g. a tic tac toe board), a set of **states** $\mathcal{S}$ (e.g. the possible configurations of 'O' and 'X') and a set of **actions** $\mathcal{A}$ (e.g. move to one of the empty positions in the board).



Markov Decision Processes (MDP)

In RL the environment is modelled as a Markov Decision Process.

- A MDP is like a **Markov Chain** where there are transitions between states with some probabilities but now we have **actions**, which can be thought as intermediate states.
- Additionally, when transitioning from one state to another, there is the possibility that the agent receives **reward** depending on if the action taken has been "good or bad".

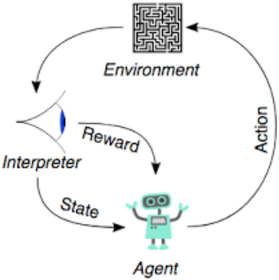


MDP formalism (I) Transition operator

- We define the **transition function** T as

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

so that $T(s, a, s')$ is the probability of a system in a state "s" to make a transition to a new state "s'" after taking action "



- The system being controlled is *Markovian*:

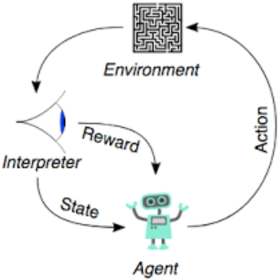
$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1}).$$

MDP formalism (II) Reward function

- 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.



MDP definition

Definition

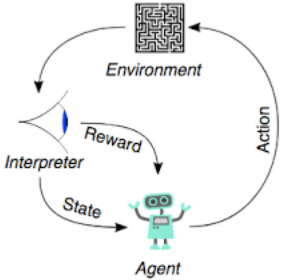
A **Markov decision process** (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, T, R)$ in which $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ a finite set of actions, T a transition probability function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ and R a reward function, $R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

Control of a MDP: agent's policy

- We define a **deterministic policy** π as

$$\pi : \mathcal{S} \longrightarrow \mathcal{A}$$
$$s \longmapsto \pi(s) = a$$

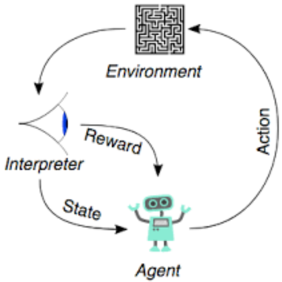
- The goal is to **find an optimal policy** π^* that takes the actions that maximize rewards.



Controlling the environment using a policy π

A policy π can be used to make evolve 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 a obtained reward $r_0 = R(s_0, a_0, s_1)$.
- By iterating this process, one obtains a sequence $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$ of state-action-reward triples.



If the task is episodic, the sequence of state-action-reward triples ends in a finite number of iterations τ.

When to use reinforcement learning?

We need a Markov Decision Process $(\mathcal{S}, \mathcal{A}, T, R)$ that can model our system.

We said that we want an "optimal policy" that maximizes rewards. But which rewards?

In RL we want to maximize: the **discounted sum of rewards** received by the agent starting from state s_t:

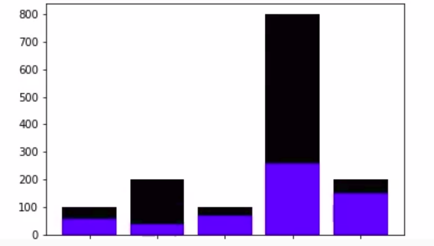
$$R_t = \sum_{k=0}^{\tau} \gamma^k r_{t+k} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^\tau r_{t+\tau} \approx r_t + \gamma R_{t+1}$$

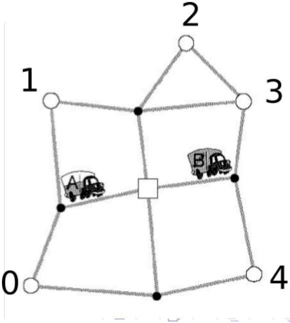
where $r_t = R(s_t, a_t, s_{t+1})$ and $\gamma \in [0, 1]$ is a **discount factor** that determines the importance of future rewards.

Outline

States and Actions

- n shops, k trucks, (remind assumptions), $\tau = 30$ (each step t of an episode will correspond to one day)
- **States**: $s = (c_0, \ldots, c_{n-1})$, c_i the stock of shop i.
- **Actions**: $a = (p'_0, \ldots, p'_{k-1})$, p'_i the position of truck i.





States and Actions

- n shops, k trucks,
- Total number of possible actions:

$$(n+1)^k$$

  - n=5, k = 2 —> 36
  - n=12, k = 4 —> $28 \cdot 10^3$
  - n=30, k = 5 —> $28 \cdot 10^6$
- Infinite number of states (stock is a real number): we will need to discretize states in order to apply classical reinforcement learning.

Function of rewards

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

Three contributions:

- J_{costs}: **economical costs** such as transport distances, amount of product unloaded, holidays,...,
- J_{levels}: **levels of stock** of each shop,
- Additional terms such as $J_{\text{extra},1}$, a cost for **penalizing a truck that goes to some shop but can't deliver** its product (because the shop is too full).

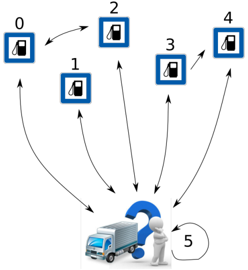
J_{costs}

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

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

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

for some constant C_{costs} that makes J_{costs} be dimensionless. L_i the capacity of truck i.

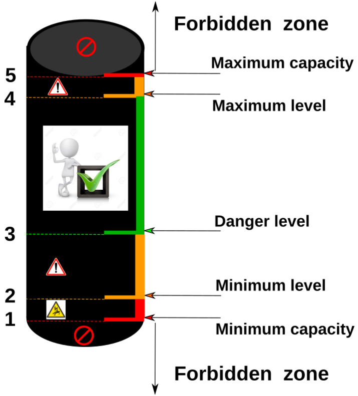


J_{levels}

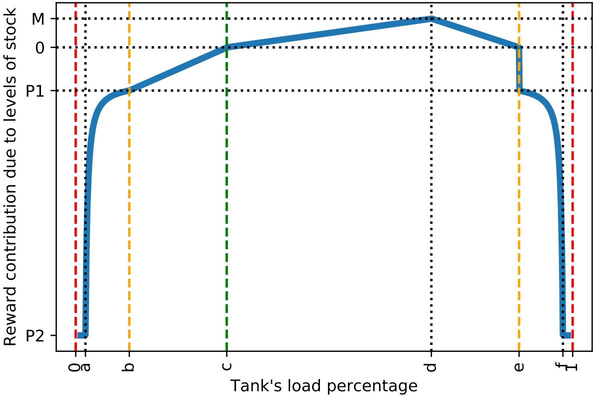
If $x^{(i)}(s')$ denotes the **percentage of stock available** in shop i.

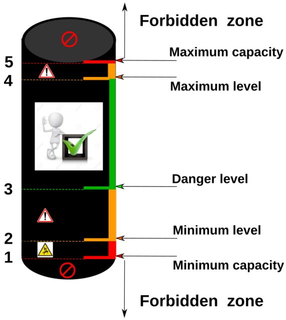
Then we consider the following decomposition for the levels of stock contribution:

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



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





Outline

Summary: Reinforcement learning and Model

Problem: how, when and where have the trucks to be sent to the shops in order to maximize the sum of discounted rewards R_t: find π^*.

- We have a MDP model for $\mathcal{S}, \mathcal{A}$ and R,
  - n shops, k trucks, (and some assumptions)
  - **States**: $s = (c_0, ..., c_{n-1}) \in \mathcal{S}$,
  - **Actions**: $a = (p'_0, ..., p'_{k-1}) \in \mathcal{A}$.
  - **Rewards** function:

$$R(s, a, s') = \mu_1 J_{\text{costs}} + \mu_2 J_{\text{levels}} + \mu_{3,1} J_{\text{extra},1}, \qquad \mu_i \geq 0,$$

- but we don't know the transition operator T. Thus, we need RL algorithms that do not need prior knowledge of T: e.g., **Q-learning**.

The Q values: a way to quantify goodness of state-action pairs

Definition

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:

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[R_t \middle| s_t = s, a_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\tau} \gamma^k r_{t+k} | s_t = s, a_t = a\right]$$

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

Optimal Q-values and an optimal policy

Definition

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

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

Assume we know $Q^*(s, a)$ 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

$$\boxed{\pi_Q(s) = \arg\max_{a \in \mathcal{A}(s)} Q^*(s, a), \qquad \forall s \in \mathcal{S},}$$

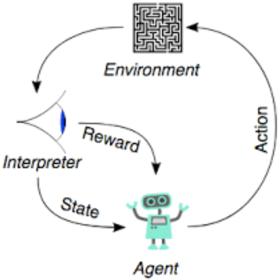
and it is an optimal policy according to the definition above.

Learning by experience

To learn the optimal Q values one simulates different episodes E_j using some exploration-exploitation policy π for selecting actions:

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

- The **most exploitatory** action-selection criteria would consist on using π_Q for the current learned Q-values if the current state is known, otherwise chose an action randomly.
- The **most exploratory** criteria would be the one which selects an action completely random in every step.



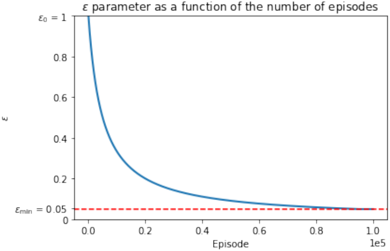
ε-greedy exploration policy

An interesting compromise between the two action-selection extremes is **the ε-greedy policy**:

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

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

Policy π_ε executes the greedy policy π_Q with probability $1-\varepsilon$ and the random policy with probability ε.



The Q-learning algorithm

procedure Q-LEARNING(γ, 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**
        $s_0^j \leftarrow \texttt{Random\_choice}(s \in \mathcal{S})$ ▷ Initialize the system
        for $t \in \{0, ..., \tau - 1\}$ **do**
            Choose $a_t^j \in \mathcal{A}$ based on 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:

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

        end for
    end for
end procedure

Outline

Model used in the simulations

- Case $n = 5$ shops, $k = 2$ trucks.
  - Shops of capacity 100, 200, 100, 800, 200.
  - Trucks of capacity 70, 130.
- We discretize **states**: $s = (c_0, ..., c_{n-1})$ in 4 subintervals for each shop $\rightarrow Q(s, a)$ is a tabular function,
- **Actions**: $a = (p'_0, ..., p'_{k-1})$.
- Rewards function:

$$R(s, a, s') = \mu_1 J_{\text{costs}} + \mu_2 (J_{\text{levels}} + J_{\text{extra},1}), \qquad \mu_1, \mu_2 \geq 0.$$

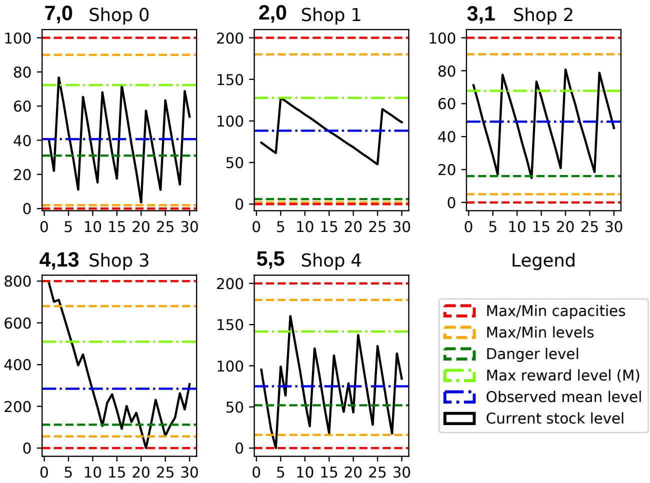
- Q-learning simulations of 100.000 episodes of length $\tau = 30$ (days) $\rightarrow$ 7/8 hours for training.

Simulations

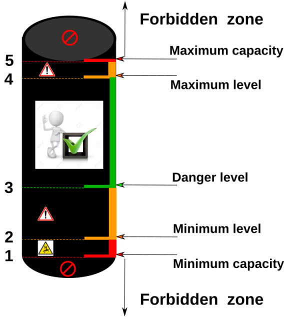
$$R(s, a, s') = \mu_1 J_{\text{costs}} + \mu_2 (J_{\text{levels}} + J_{\text{extra},1}), \qquad \mu_1, \mu_2 \geq 0.$$

1) Deterministic consumption rates without economical costs: $\mu_1 = 0$.
2) Deterministic consumption rates WITH economical costs: $\mu_1 \neq 0$.
3) Stochastic consumption rates ($\pm 25\%$) without economical costs: $\mu_1 = 0$.
4) Stochastic consumption rates ($\pm 25\%$) WITH economical costs: $\mu_1 \neq 0$.

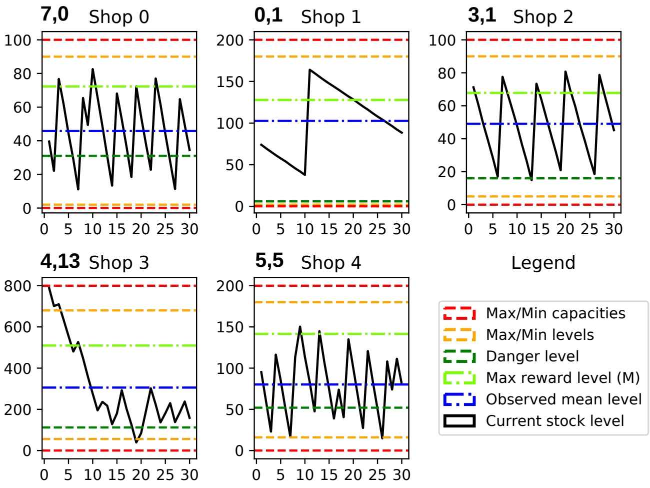
1) Deterministic + $\mu_1 = 0$



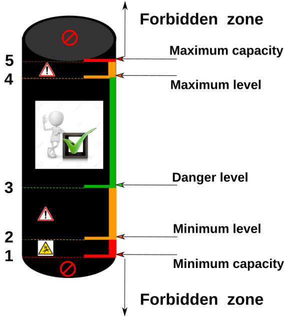
40 trucks sent, $J_{\text{costs}} = 2870$



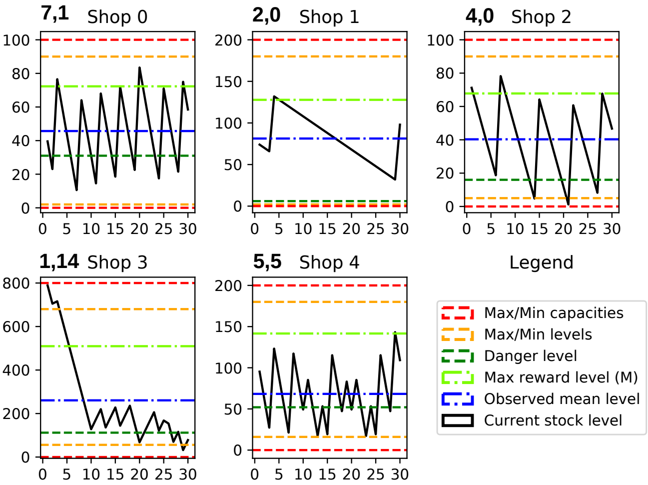
2.1) Deterministic + $\mu_1 \neq 0$ (10^{-6})



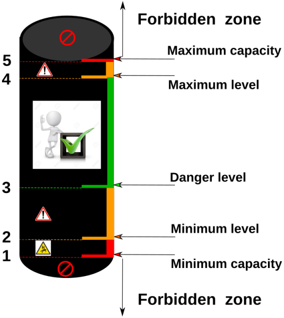
39 trucks sent, $J_{\text{costs}} = 2861$



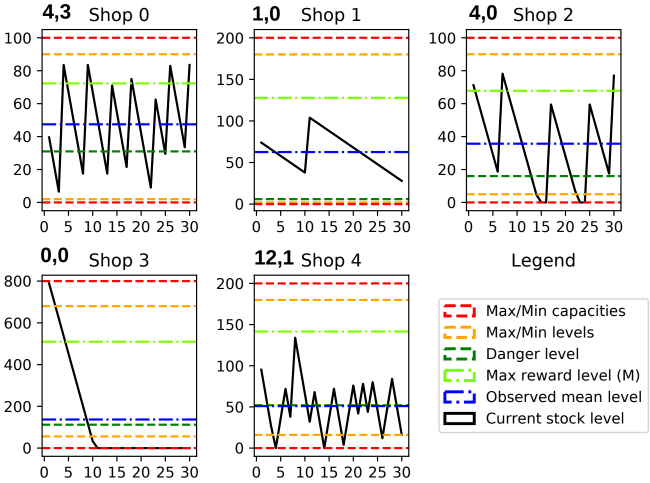
2.2) Deterministic + $\mu_1 \neq 0$ (10^{-4})



39 trucks sent, $J_{costs} = 2777$

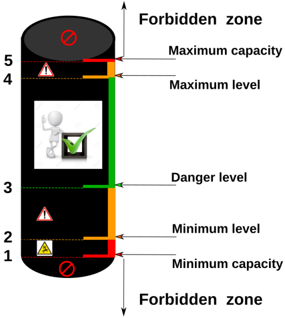


2.3) Deterministic + $\mu_1 \neq 0$ (10^{-3})

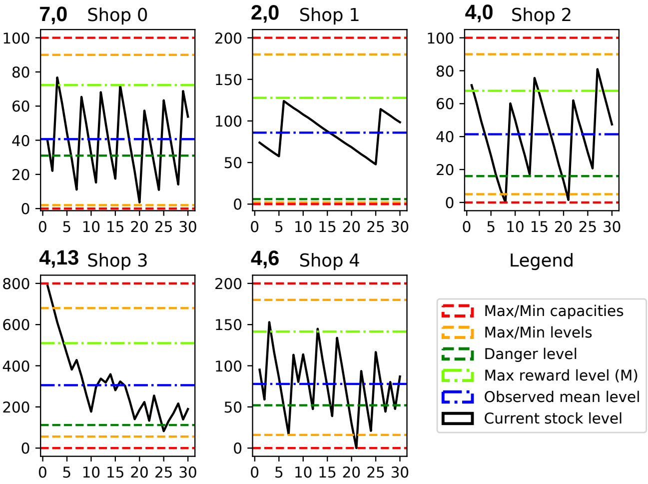


25 trucks sent,

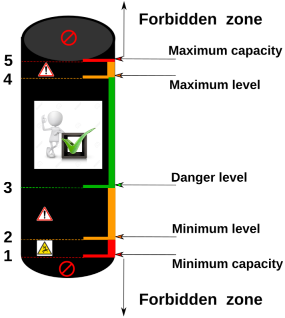
$J_{\text{costs}} = 1182$



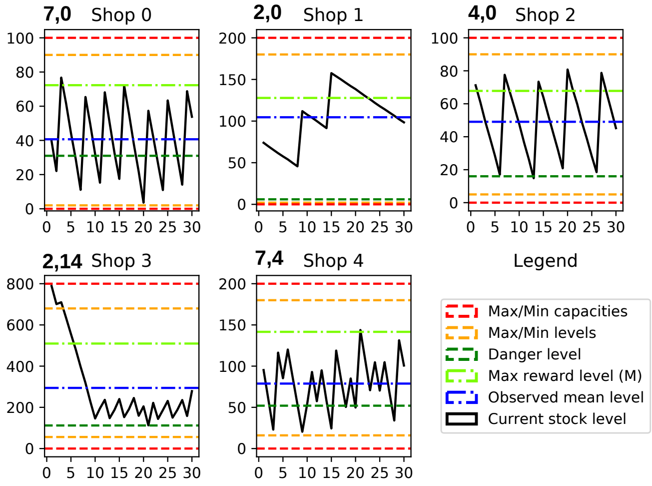
3) Stochastic + $\mu_1 = 0$ (25% noise)



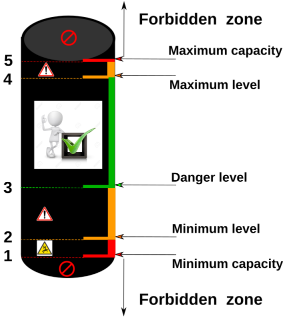
40 trucks sent, Shops 3 & 4: 8 small and 19 big, $J_{\text{costs}} = 2868$



4) Stochastic + $\mu_1 \neq 0$ (25% noise)



40 trucks sent, Shops 3 & 4: 9 small and 18 big, $J_{\text{costs}} = 2820$



Conclusions

Q-learning podria ser **efectivo** para número de plantas y camiones pequeño, **pero el número de estados-acciones explota**:

$$|\mathcal{S}| = 4^n, \text{ si discretizamos en 4 partes}$$

$$|\mathcal{S} \times \mathcal{A}| = 4^n \times (n+1)^k \approx 36800 \text{ para n=5, k =2}$$

pero

$$|\mathcal{S} \times \mathcal{A}| = 4^n \times (n+1)^k \approx 10^{16} \text{ para n=20, k =3}$$

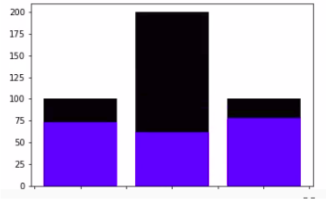
Aún así, discretizando solo en 4 partes en cada planta se pierde mucha información sobre el sistema.

Outline

Toy system definition

To get started with **Deep Neural Networks** (DNN) we have studied the capability of a DNN of **learning a simple hard-coded policy**.

First, 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 $i \in \{0, 1, 2, 3\}$.

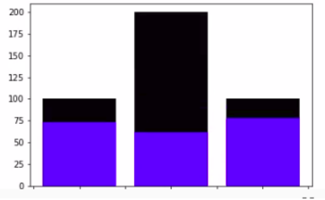


A simple hard-coded policy

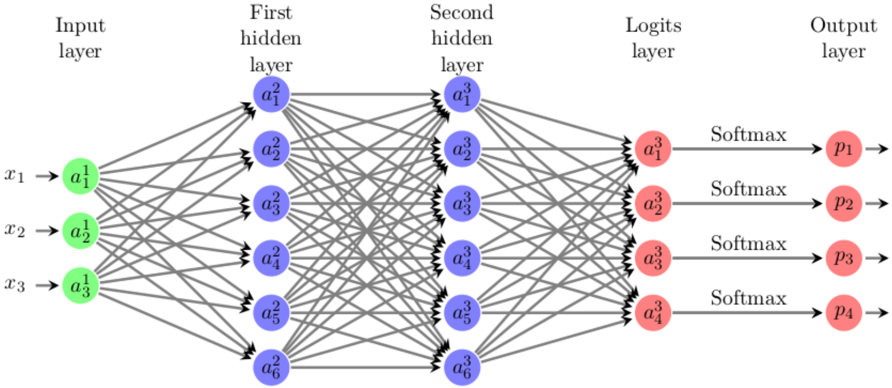
We consider the **policy π_m that sends the truck to the shop with minimum stock if possible** (i.e., if the load L of the truck fits all in the shop without surpassing the maximum capacities of stock), **or makes it stay at the depot otherwise**.
Mathematically, speaking, we define π_m as follows:

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

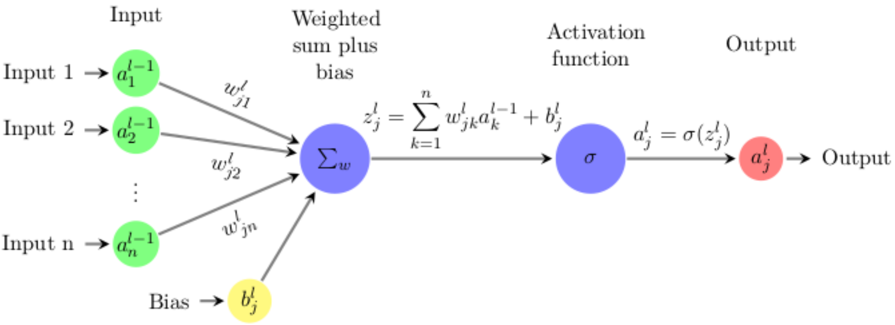
If we now generate a train dataset X containing possible states of the system and define a target set $Y = \pi_m(X)$, we are in the framework of classification.



DNN Architecture for classification

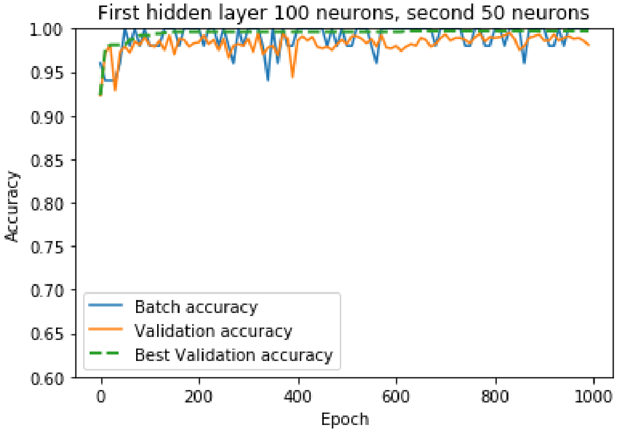


A single neuron structure

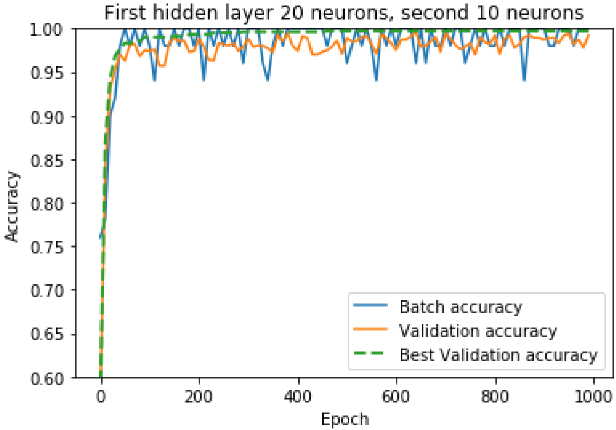


Outline

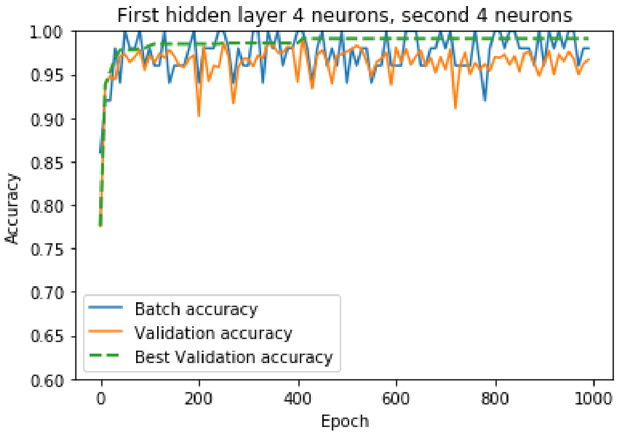
First hidden layer 100 neurons, second 50 neurons



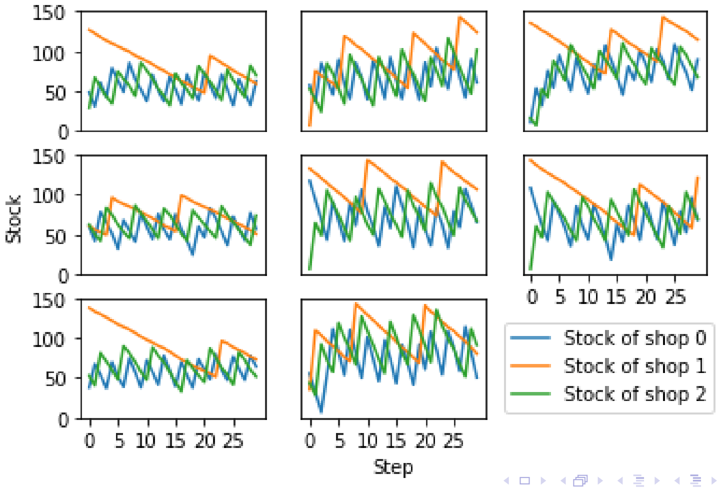
First hidden layer 20 neurons, second 10 neurons



First hidden layer 4 neurons, second 4 neurons



8 test episodes of length 30 for the learnt policy

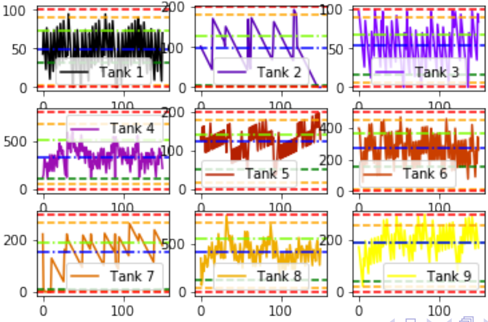


Outline

What is next?: Deep Reinforcement Learning

- Use Deep Neural Networks for Reinforcement Learning: **Policy gradient algorithm**.
- Try to make the problem scalable to "arbitrary" number of shops and trucks.

"Spoiler":



Thanks for your attention!
Any questions?

- **Master's thesis notebooks and figures**:

  https://github.com/dsalgador/master-thesis

- **Open AI gym environment for our product delivery problem**:

  https://github.com/dsalgador/gym-pdsystem