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

Dani Salgado

Advisor: Toni Lozano

Master's thesis

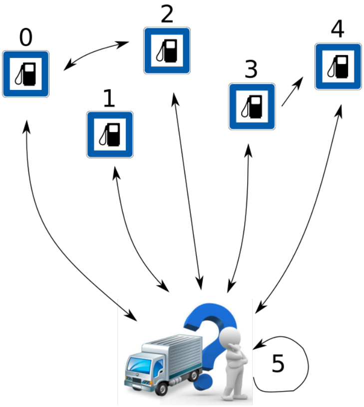
Universitat Autònoma de Barcelona

GRUPO AIA

Outline

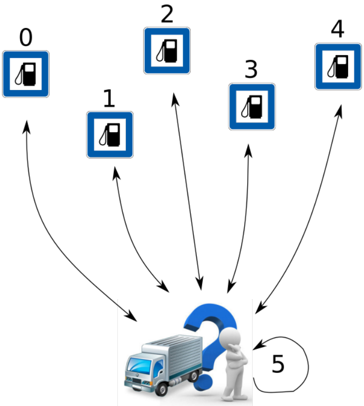
Problem statement

- Our client owns a chain of stores specialized in a particular product, and he/she pays to a transport company to bring their product from a depot 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 product available for costumers (among other possible constraints).



Initial assumptions

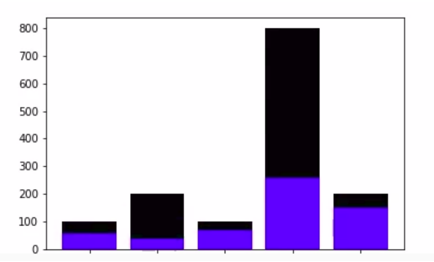
1. We have a system of n shops and k trucks.
2. We assume **single unloads** (a truck can only go to one shop everyday).
3. Trucks leave the depot fully loaded and return completely empty.
4. The price that our client has to pay to the transport company is given by some cost function J_{costs} which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

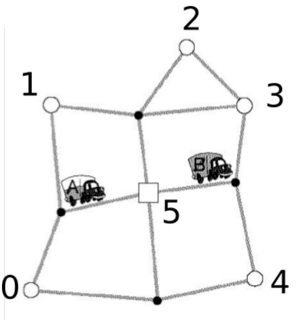


Outline

States and Actions

- n shops, k trucks, (remind assumptions), $\tau = 30$ (each step t of an episode will correspond to one day)
- **States**: $s = (c_1, ..., c_n)$, c_i the stock of shop i.
- **Actions**: $a = (p'_1, ..., p'_k)$, p'_i the position of truck i.





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).

Conclusions from Classical Reinforcement Learning (Q-learning)

Q-learning could be **efective** for a small number of shops and trucks, **but the number of states-actions explodes**:

$$|\mathcal{S}| = 4^n, \text{ if we discretize each shop in four parts}$$
$$|\mathcal{S} \times \mathcal{A}| = 4^n \times (n+1)^k \approx 36800 \text{ for n=5, k=2}$$

but

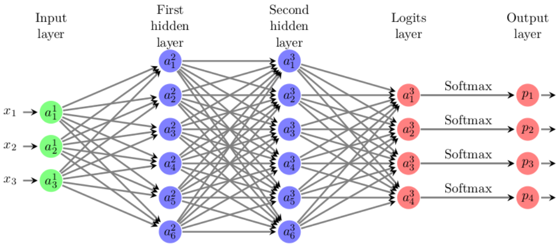
$$|\mathcal{S} \times \mathcal{A}| = 4^n \times (n+1)^k \approx 10^{16} \text{ for n=20, k=3}$$

Even so, discretizing in only 4 parts for each shop we are losing a lot of information about the system.

Outline

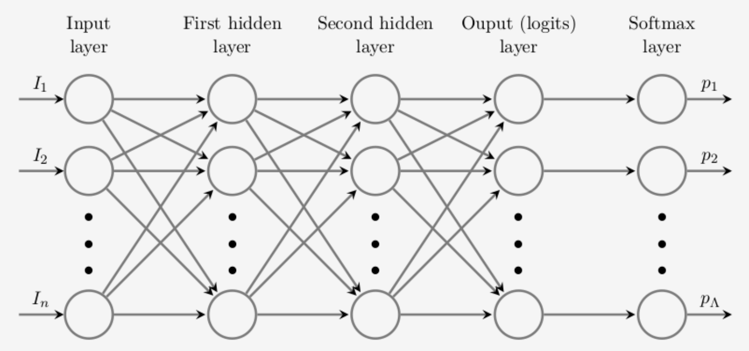
What is next?: Deep Reinforcement Learning

- Make the problem scalable to "arbitrary" number of shops and trucks.
- Use Deep Neural Networks for Reinforcement Learning: **Deep Policy gradient algorithm** (our approach).
- A DNN will be a parametrized policy π_θ, and the goal is to optimize this policy by updating the parameters θ (the weights and biases).



DNN architecture

The inputs of the network are the states $s = (c_1, ..., c_n) \subset \mathbb{R}^n$, where c_i are the "current" stocks of each shop; $I_j = c_j \ \forall j$. There are $\Lambda = (n+1)^k$ output neurons, one for each possible action. We have: $p_i \equiv \pi_\theta(a_i|s)$.



Outline

How do we train our DNN?: Policy Gradient Theorem

The cost function to consider is the total discounted reward obtained in every episode,

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

Theorem (Policy Gradient)

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

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

Policy Gradient Theorem Estimation

PG theorem 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:

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

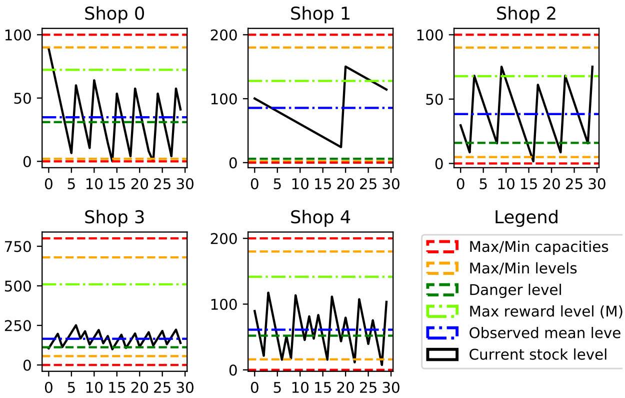
With this we can perform (for instance) a Gradient Ascent step to train the parameters of our DNN:

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

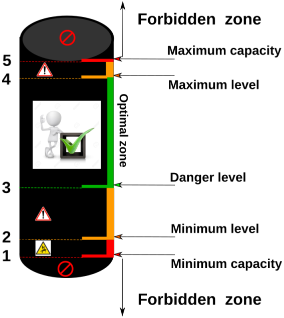
where $\alpha > 0$ is the learning rate.

Outline

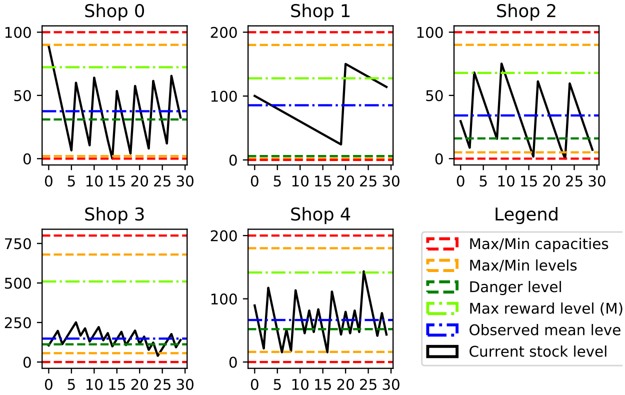
1) Deterministic without costs, $\mu_1 = 0$



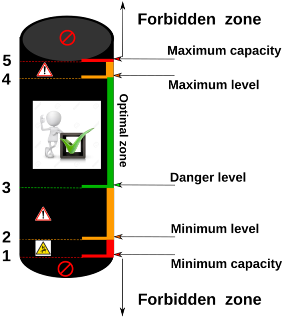
42 trucks sent, $J_{\text{costs}} = 33658$.



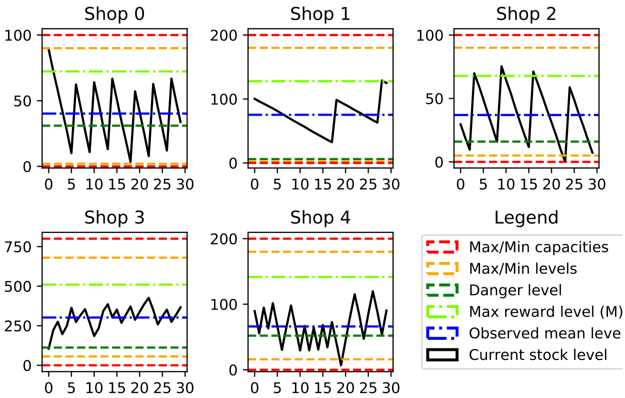
2.1) Deterministic with costs, $\mu_1 = 10^{-6}$



42 trucks sent, $J_{\text{costs}} = 33175$.

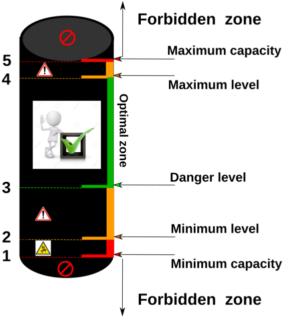


3) Stochastic without costs, $\mu_1 = 0$ (10% noise)

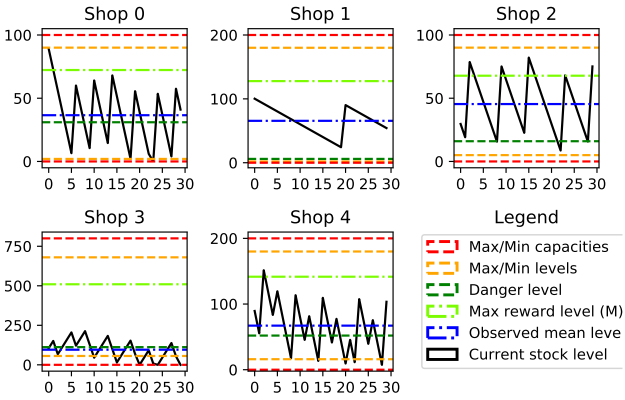


49 trucks sent, Shops 3 & 4: 30 small and 19 big,

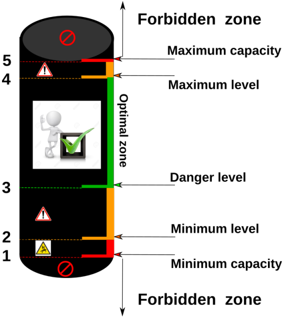
$J_{costs} = 34212$



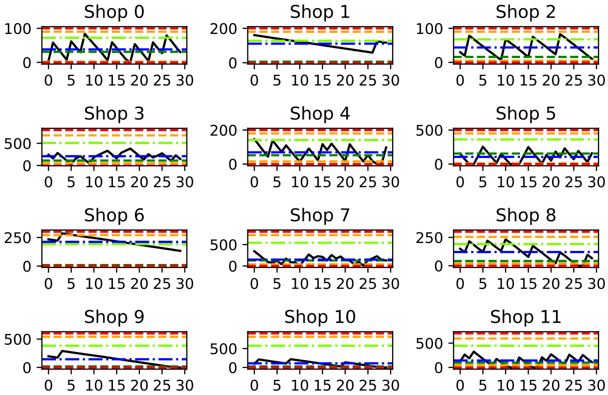
4) Stochastic with costs, $\mu_1 = 10^{-3}$ (10% noise)



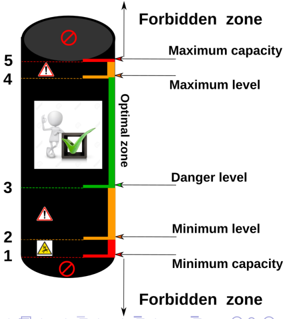
40 trucks sent,
Shops 3 & 4: 17 small and 23 big,
$J_{\text{costs}} = 31073$



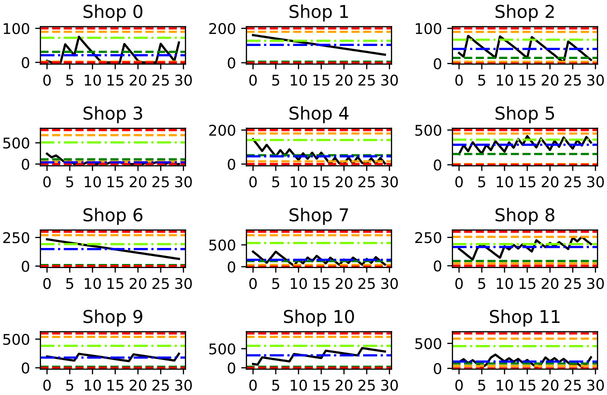
5) Stochastic without costs, $\mu_1 = 0$ (10% noise), $n = 12, k = 3$



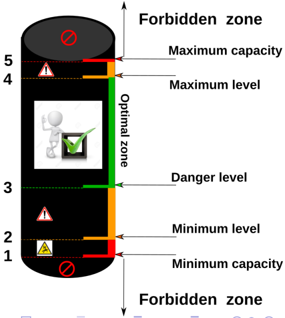
80 trucks sent, 30 small, 30 medium, and 20 big,



6) Stochastic with costs, $\mu_1 = 10^{-6}$ (10% noise), $n = 12, k = 3$



78 trucks sent, 30 small, 30 medium, and 18 big



Outline

Classical v.s. Deep Reinforcement Learning (I)

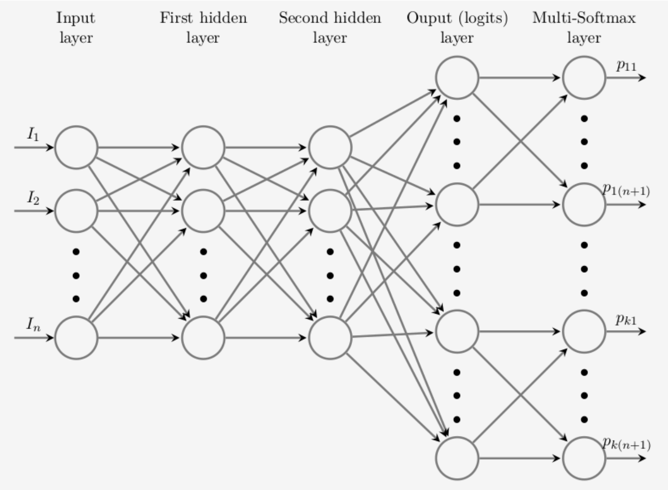
- The **scalability of Q-learning** in terms of the number of shops n and trucks k **is very bad**.

  If d is the number of parts in which we discretize the level of stock of shops, then, 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 stock of each shop. We just use the exact level of stock of each shop as input of 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 10^4 output neurons or more, would be impractical to train in a plausible amount of time.

Classical v.s. Deep Reinforcement Learning (II)

- That is the reason for which we did not consider more than 12 shops and 3 trucks:
  - For $n = 12, k = 3$: $|\mathcal{A}| = 2197 \sim 10^3$,
  - but for $n = 12, k = 4$: $|\mathcal{A}| = 28561 \sim 10^4$.
- Since the scalability is now limited by the number of output neurons $\Lambda = (n+1)^k$, 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.

More scalable "Deep Policy Gradient" for product delivery



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`

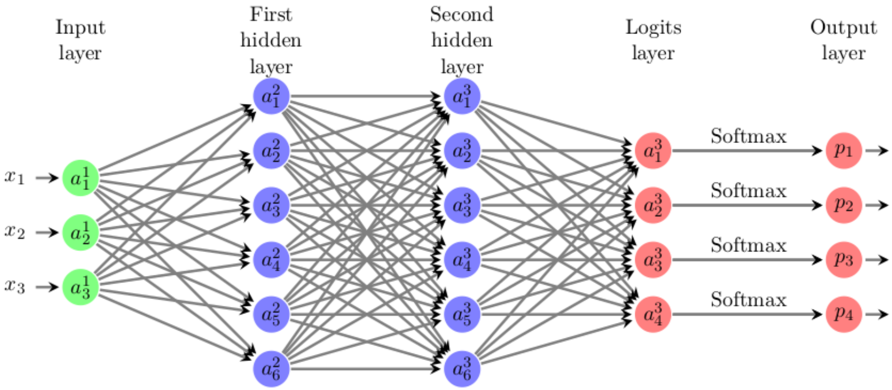
Outline

Outline

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 $\nabla_\theta \log \pi_\theta(a_t^j | s_t^j)$.

We are going to see that we can use the **cross entropy** to compute these term.

DNN for classification (ex: 3 features , 4 classes)



Fixed $x \in X$ (train dataset), the final outputs p_i of the DNN are normalized to sum 1, so that they can be considered as a probability distribution $p(x)$ over classes.

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

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

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

where $y(x)$ is the corresponding label to the training observation x, and "i" refers to the i-th class.

Cross entropy loss

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

$$H(p, q) = \sum_i q_i \log p_i$$

In the policy gradient context, inputs are states $s \in \mathcal{S}$, so that the output of the DNN we considered can be thought as a probability distribution

$$p(s) = (p_1, ..., p_\Lambda) \equiv (\pi_\theta(a_1|s), ..., \pi_\theta(a_\Lambda|s))$$

over actions given states.

But now, what does play the role of the"true probability distribution" $q = q(s)$ that we had in a usual classification framework?

Remember we needed to know how to compute: $\nabla_\theta \log \pi_\theta(a_t^j|s_t^j)$

- Imagine that the vectorial representation of state s_t^j is the current input of the DNN that we are training (i.e., the levels of stock of shops at time t in episode j).
- Assume $\hat{a}_t^j$ 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 (this encoded vector plays the role of $q(s)$, $s = s_t^j$).
- The vector $\hat{a}_t^j$ can be obtained by sampling a number r between 1 and Λ with a multinomial distribution with the probabilities $p_1, ..., p_\Lambda$, 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.

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

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

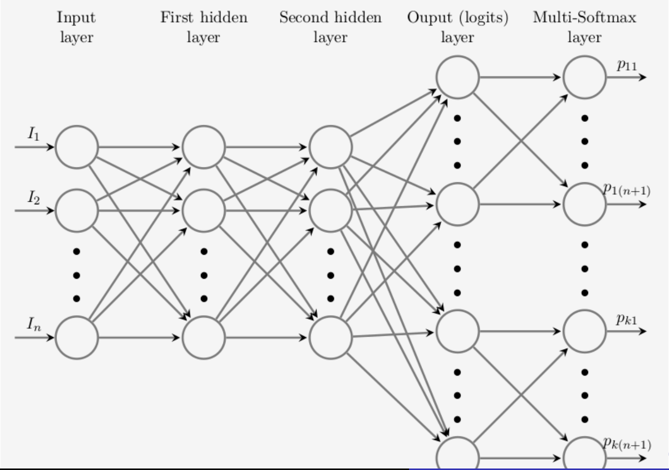
Now if we take gradients with respect to θ in the above equation we obtain

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

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.

Outline

More scalable "Deep Policy Gradient" for product delivery (I)



More scalable "Policy Gradient" for product delivery (II)

- 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 the previous slide can be thought as a function whose outputs are a tuple of the form $(\pi_{\theta_1}(a_1|s), ..., \pi_{\theta_k}(a_k|s))$, 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).
- If there is independence between the actions performed by each truck, we could define

$$\pi_\theta(a|s) := \prod_{i=1}^{k} \pi_{\theta_i}(a_i|s).$$

- In particular, we would have

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

More scalable "Policy Gradient" for product delivery (III)

Then we would have the following expression for estimating the gradient of the cost function according to PG Algorithm:

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

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 $i \in \{1, ..., k\}$)

More scalable "Policy Gradient" for product delivery (IV)

Similarly, we could follow the derivation of the cross entropy we did for a single softmax layer and arrive to the following expression for each $\nabla J^j_{\theta,l}$,

$$\nabla_\theta H(\hat{a}^j_{lt}, p_\theta) = \sum_{i=1}^{n+1} (\hat{a}^j_{lt})_i \nabla_\theta \log \pi_{\theta_i}((a_l)_i | s^j_t).$$

for $l \in \{1, ..., k\}$.

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 an average direction that makes each truck to perform "the best action (in average)".