CLD771 Minor Project - Final Report Using Deep Reinforcement Learning for scheduling gasoline blending and distribution (SGBD)

Samarth Bhatia (ch1190124@iitd.ac.in)
Hariprasad Kodamana (kodamana@iitd.ac.in)

November 15, 2021

---

1. Abstract We aim to solve the problem of scheduling gasoline blending and distribution, and by extension, general production scheduling. We make a model environment to simulate the insides of a chemical plant, and train on it using different algorithms to compare which gives us the best results and analyze those results.

2. Introduction

2.1 Reinforcement Learning Reinforcement Learning (**RL**) deals with the task of teaching agents what actions to take given a certain environment/state. These actions bring about change in the environments with the goal of receiving a reward. When the agent is a neural network, the field is said to be Deep Reinforcement Learning (**DRL**).

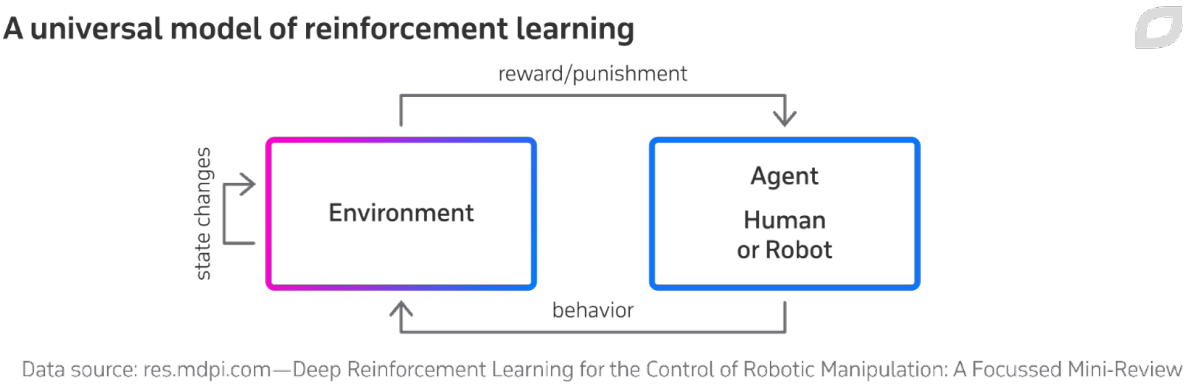

Figure 1: A universal model of reinforcement learning

These environments need to be modelled according to and is specific to whatever process we are trying to learn/simulate. The environments contain all rules related to how the action effects the environment, the completion/success of the process and any reward. They need to contain all balances e.g. mass balances, material balances, energy balances, etc. Modelling them precisely and correctly thus becomes essential for ensuring that we are learning what we intend to.

2.2 DRL Methods 2.2.1. Action-Value Methods

Action value methods are based on storing certain values for every action possible after a certain state is reached. This can be abstracted as a simple table where you lookup the state in the row indices and actions in the column indices. In the case of Q-Learning, the values inside the table corresponding to one state and one action are called Q-values. These basically tell us how "good" an action is after any given state. These are calculated using the Bellman equation.

Q-table	Action 1	Action 2	Action 3
State 1	0.12	0.57	**1.96**
State 2	**0.25**	-0.03	0.08
_ _ _ _	_ _ _ _	_ _ _ _	_ _ _ _
State n	0.34	**0.49**	-0.2

We can see an example of a Q-table in Table 1. Whichever action out of the three has the highest Q-value at that state will be chosen.

We can also write the Bellman eqn as:

$$\underbrace{\text{New } Q(s,a)}_{\substack{\text{New}\\ \text{Q-Value}}} = (1-\alpha)\underbrace{Q(s,a)}_{\substack{\text{Current}\\ \text{Q-Value}}} + \underbrace{\alpha}_{\substack{|\\ \text{Learning rate}}}\Big[\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\substack{|\\ \text{Discount rate}}} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given}\\ \text{new state and all possible actions}}}\Big]$$

The Q-value is updated depending on the Q-value for the current state added to the learning rate multiplied by all future rewards (i.e. the sum of current rewards and discounted maximum predicted reward at next state if we were to take action a).

Since we are storing all possible state-action value pairs as a table, this method would be most suitable for a discrete environment, where the separation of states is meaningful enough that the Q-values do differ by reasonable amounts. Q-Learning and Deep Q-Learning are examples of action-value methods.

2.2.2. Policy-Gradient Methods

These rely on updating policy parameters using gradient descent according to the rewards received. [1]

$$J(\theta) = \mathbb{E}_\pi\ [r(\tau)]$$
$$\theta_{t+1} = \theta_t\ +\ \alpha \nabla J(\theta_t)$$

As in the figure, we need to maximize J, the expected value of all rewards. We can do this by parametrizing the policy as π_θ, and then using a gradient ascent step on θ. We get:

$$\begin{aligned}\nabla \mathbb{E}_\pi[r(\tau)] &= \nabla \int \pi(\tau) r(\tau) d\tau \\ &= \int \nabla \pi(\tau) r(\tau) d\tau \\ &= \int \pi(\tau) \nabla \log \pi(\tau) r(\tau) d\tau \\ \nabla \mathbb{E}_\pi[r(\tau)] &= \mathbb{E}_\pi[r(\tau) \nabla \log \pi(\tau)]\end{aligned}$$

Then, using some manipulation from the value of π_θ, we get:

$$\log \pi_\theta(\tau) = \log \mathcal{P}(s_0) \; + \; \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \; + \; \sum_{t=1}^{T} \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)$$

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[r(\tau) \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \right) \right]$$

So, we can see why these are called policy-gradient methods: they utilize a policy π_θ and the update step involves the gradient of the objective function J. These are much more suited for continuous environments. The REINFORCE algorithm and PPO (Proximal Policy Optimization).

2.2.3. Actor-Critic Methods

Both action-value and policy-gradient methods have their downsides, like being too resource consuming and inefficient and learning occurring only after an episode is complete. So, we introduced Actor-Critic methods, which work like policy-gradient methods but with the advantage of Temporal-Difference learning (learning as we go, within each episode as well).

They consist of two networks: the Actor network and the Critic network. The Actor Network decides what action is to be taken. The Critic Network informs the actor how good the action taken was, and how it should adjust for receiving a better reward. The learning of the actor is based on policy gradient approach. In comparison, critic evaluates the action by computing a value function.

A2C (Advantage Actor-Critic), A3C (Asynchronous A2C), DDPG (Deep Deterministic Policy Gradient) are examples of actor-critic methods.

2.3. Our Problem The environment is essential and

3. Literature Review In this short literature review, I will be summarizing two papers and what is relevant to us in this project.

- "Scheduling of gasoline blending and distribution using graphical genetic algorithm"[2]: This paper by Bayu et al. was about the problem of SGBD using a GGA. They consider certain variables and parameters associated with SGBD such as flow rates of products, costs of components, demand of orders, costs of changeovers of products in blenders and product tanks, and tardiness costs etc., which makes for a close-to-reality situation. Using a graphical genetic algorithm (**GGA**), they achieve better results than a industry standard mixed integer nonlinear programming (**MINLP**) model.
- "A deep reinforcement learning approach for chemical production scheduling"[3]: In this paper, the authors Hubbs et al. apply Deep Reinforcement Learning (**DRL**) to a production problem in Chemical Engineering in the form of a Advantage Actor Critic (**A2C**) algorithm, which involves 2 neural networks, one for the actor (which decides what to do based on the current state), and one for the critic (which approximates a value function to objectively tell how good the actions taken by the actor are). DRL **outperforms** the baseline mathematical (Linear Programming) models by a significant amount. The baseline models include Shrinking Horizon MILP (**SHMILP**), Deterministic MILP (**DMILP**), Perfect Information MILP (**PIMILP**) and Stochastic MILP (**SMILP**). The PIMILP provides an optimistic **upper bound** of performance for the problem at hand.

4. Modeling In RL, we need to model new environments for new tasks. This can quickly make the environments unorganized, and lead to confusing function calls/API design. `gym` by OpenAI standardizes environments, and makes the environments structured. Since `gym` uses a fixed API, it makes the interaction with environments intuitive and standard. We can drop-in new agents without needing to change the environment. This also makes benchmarking of agents possible. So, I decided to keep the environment `gym` compliant.

I have started to implement an environment similar to the one used in "Paper 1"[2]. This will also be available as an open-source python package when finished (with less detail) and is regularly being updated on Plutonium-239/deeprl-in-sgbd. We can represent our environment with a graphic as shown in figure.

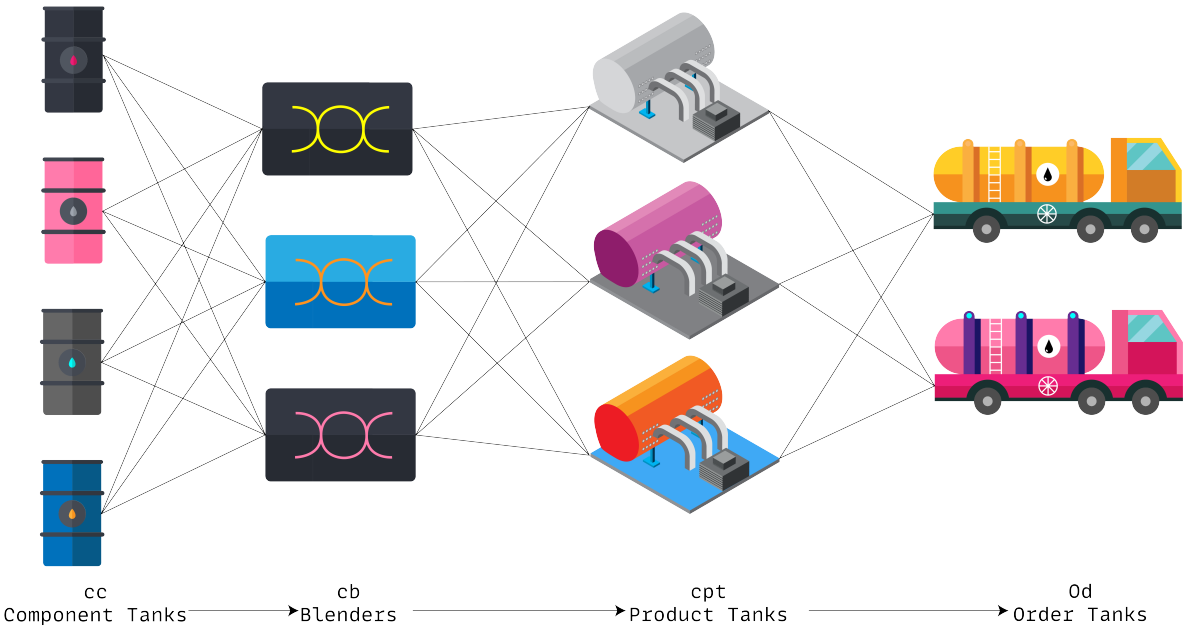

Figure 2: Model Diagram

Component Tank, Product Tank and Order Tank resource credits to vecteezy.com, all other graphics are made by me

4.1. `gym` Environments All `gym` environments have these basic attributes/methods:

1. `observation_space`: the space of all possible values that can be observed. In our case these are inventory levels in all the different units (component tanks, blenders and product tanks), the amount of changeover in the blenders and the product tanks, tardiness in orders, flow rates and amounts flowed in between different units.

I have modeled all these as continuous variables using the `gym.spaces.Box` class

```
gym.spaces.Box(low= 0, high= 1, shape= (5,3))
```

This makes a continuous space ranging from `0` to `1` in the form of a `5x3` matrix. An example value of

this can be:

$$\begin{bmatrix} 0.23 & 0.47 & 0.88 \\ 0.05 & 0.19 & 0.61 \\ 0.45 & 0.27 & 0.37 \\ 0.01 & 0.10 & 0.97 \\ 0.00 & 0.29 & 0.11 \end{bmatrix}_{5\times 3}$$

2. `action_space`: the space of all possible actions that can be taken. Currently they are modelled as binary variables representing whether material is transferred between
   1. component tanks `{i}`and blenders `{n}` : matrix of shape `i*n`
   2. blenders `{n}` and product tanks `{j}` : matrix of shape `n*j`
   3. product tanks `{j}` and order tanks `{o}` : matrix of shape `j*o`
   4. and if product `{p}` is assigned to blender `{n}` : matrix of shape `p*n`

This is done efficiently using the `gym.spaces.MultiBinary` class, which allows us to create a one-hot/sparse matrix as shown below:

```
gym.spaces.MultiBinary(shape= (4,2))
```

This can be sampled at random using the `.sample()` function which will give us:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}_{4\times 2}$$

3. The `reward` function: For our environment and in Paper 1, the objective to minimize has been taken as the sum of the component costs, changeover costs and tardiness costs.

   So we can set the reward to be `-ve` of this sum alongwith the earnings from completing orders, as in Reinforcement Learning, we try to maximize the reward.

   reward = margin from orders−(component costs + changeover costs[blenders] + changeover costs[product tanks] +

4. The **`step`** function: This is the main function in the environment; which takes an action (any possible action as defined in `action_space`) and propagates it through the whole setup for one timestep. This includes updating all inventory levels, changing value of flow rates, checking what product has been formed and packaged for being sent to the order, and all other ***dynamic*** aspects of **SGBD**.

   In our environment, the `step` function takes an `action` in the one-hot/sparse matrix form and converts it into meaningful *changes* in the system like material being sent between different units, different products being mixed in blenders at each timestep, the changing of the components in blenders etc.

4.2 Orders I had tested out a few different techniques and had decided on this system to implement stochastic order generation. This system generates orders for the scheduling horizon specified by the environment's `k`.

The specifics of the order system needed to be ironed out before the environment could be finished (as taking a step in the environment invloves details like what order was delayed, what order was delivered, what are the costs (and income) of each order *[required for the reward function]*)

An order 'sheet'/'book' will look something like this:

idx	order_date	due_date	amount	margin	product
0	3	5	100.0	29.5	A
1	7	9	100.0	63.1	A
2	9	10	100.0	33.5	A
3	2	3	100.0	8.4	B
4	4	6	100.0	46.9	D
5	0	0	100.0	58.3	A
6	1	5	100.0	13.3	C
7	4	8	100.0	19.1	D
8	7	10	100.0	55.1	C
9	7	10	100.0	48.6	D

Every order has 5 properties alongside the `index`/`idx`:

1. `order_date:` The date on which that order is **placed** (so to the agent, only orders placed upto that date will be visible, for e.g., only orders with `order_date $\le$ 2`*[order ids 3, 5, 7, 9]* will be visible to the agent at day 0 of the simulation). This is sampled from a discrete uniform distribution from `0` to the last day, i.e. `k`(in our environment)
2. `due_date:` The last date by which that order is to be **delivered** without any tardiness costs (after that day tardiness costs will be applied as 25% for each succesive late day). This is taken as the `order_date` + a *delay*, sampled from a discrete uniform distribution from `0` to `k/2`. *[`k\2` was chosen arbitrarily so that most orders have `>2` days in between `order_date` and this `due_date`]*
3. `amount:` The **amount of product ordered** to be produced. Currently this is a constant.
4. `margin:` The **financial gain** from delivering that order on/before time. This is sampled as a number from a standard normal distribution (`Z(mean=0, std=1)`) multiplied by the amount (so that in future even if we change amounts to be unequal, this already takes it into consideration *[since more amount of product ordered usually refers to more money spent]*)
5. `product:` This tells us **which product** is ordered out of a variable number of products. *[For the environment the default is 5 products `A,B,C,D,E`]*. This is randomly chosen from all the products. *[i.e., a uniform discrete distribution]*

The orders are generated when the environment is initialized/reset. It also saves this order as a CSV file.

5. Agents Once our environment is completed, we can continue to test out how different agents are able to handle the task of scheduling production given the planning horizon and orders. I have already implemented two agents, DDPG and PPO. I am in the process of implementing an A2C agent and after that I will focus on getting some DRL baseline metrics first using packages like `stable-baselines` and also get MILP (Mixed Integer Linear Programming) algorithm baselines using ds4dm/**ecole**, a `gym`-like package. An advantage of using the `gym` package from the start is that I can almost directly import my environment without many changes in `ecole` instead of having to rewrite all the code for a new API, as it is built to mimic `gym`'s API.

I will also measure and document how all these algorithms perform on the task of scheduling gasoline blending and distribution.

6. References *The references are in the "elsevier-with-titles" format.*

[1] S. Kapoor, Policy Gradients in a Nutshell, Medium. (2018). https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d (accessed November 15, 2021).

[2] F. Bayu, D. Panda, M.A. Shaik, M. Ramteke, Scheduling of gasoline blending and distribution using graphical genetic algorithm, Computers & Chemical Engineering. 133 (2020) 106636. https://doi.org/10.1016/j.compchemeng.2019.106636.

[3] C.D. Hubbs, C. Li, N.V. Sahinidis, I.E. Grossmann, J.M. Wassick, A deep reinforcement learning approach for chemical production scheduling, Computers & Chemical Engineering. 141 (2020) 106982. https://doi.org/10.1016/j.compchemeng.2020.106982.