QIAgent : An Agent For Competing The SCML OneShot Track

Authors: Ran Sandhaus: ransandhaus@gmail.com, Ophir Haroche: ophirbh@gmail.com, Nadav Spitzer: nspitzern@gmail.com, Laor Spitz: laorsp7@gmail.com - Bar Ilan University

In this work we are aiming to compete on the OneShot track of the SCML competition, using a reinforcement-learning based approach.

1. Introduction

1.1. SCML & OneShot Track

In SCML competition, agents are placed in a production supply chain and play a part as a factory, where they buy products from other agents (factories), use these products to produce other products and sell the outcomes to the next agents in the production chain. “OneShot” track is a subset of this setting, where the production chain length is only 2 - i.e. we have one level of agents, called “L0” that buy “raw material”, generate an intermediate product and sell the other level of agents, called “L1”, which generate the final product and sell to the market. The agents in each level interact with each of the agents in the other level to negotiate via a variation of the alternate-offers protocol. The simulation works in “steps” (can be equivalent to days) where in each step, exogenous contracts are presented by the market (either the “raw materials” side or “end customer” side) and are valid only for the step. The agents should negotiate to fulfill the supply/demand that are expressed by the exogenous contracts. The agents profit by the difference between selling price and buying price (there are more expenses that affect the profit, such as production cost, and penalties on unfulfilled exogenous requested quantity). The purpose of an agent is to make the highest profit, and the winner is the agent who achieved the highest one. In this track, there is no need to do production scheduling and other tasks needed in other SCML tracks - the main activity here is the negotiation and its outcome - how much we buy/sell and at which price. Since this is a more well-defined problem in nature, we can focus on the negotiation component which makes this interesting to see what methods can work well in this context.

The OneShot game is defined in http://www.yasserm.com/scml/overview_oneshot.pdf
For a schematic of the supply chain see page 2.

1.2. Abstract - Our Approach

Since we can focus on negotiation optimization and agents can vary a lot from one to another in strategy and behavior, it encouraged us to pursue a learning approach, in which the agent tries to adapt how to activate against and react to each of the other agents he plays against, in order to maximize its profits.

Initially we started with a more “naive” approach, doing a kind of binary search for the optimal price, against each agent our agent negotiates with, throughout the simulation. After investigating this approach, including an attempt to tune it’s hyper parameters to achieve better results, given unsatisfying results we headed forward into applying a more robust reinforcement learning approach.

We decided to base on the **Q-learning** algorithm - a well-known and widely used reinforcement learning algorithm. We decided to base on it since it is a model-free algorithm and given a finite

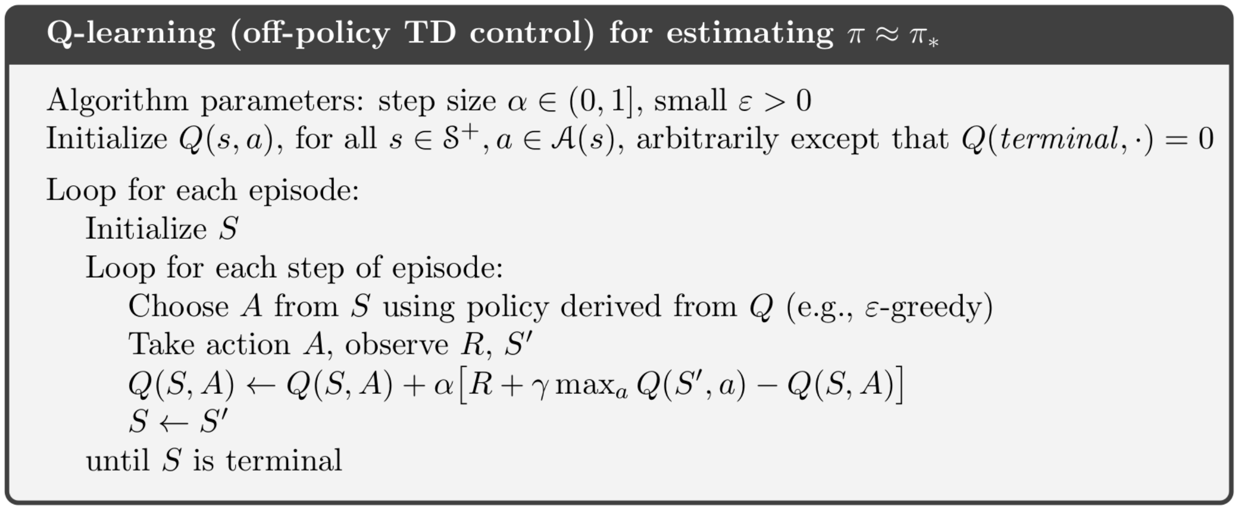
MDP, in case the problem is modeled correctly in the algorithm, it is guaranteed to converge to optimal policy to produce highest reward - here it is interpreted as the highest profit (here we try to learn the best policy, per opposing agent, for actions during negotiation). We further built on top of this algorithm, to fine-tune it to the negotiation problem and specifically the SCML setting, to make it achieve higher scores faster.

2. Approach In Detail

2.1. Reinforcement Learning & Q-Learning

Reinforcement learning is a type of learning scheme where an agent learns from gathered experience on his actions, by tuning his actions per environment's state based on past rewards. The agent learns a policy: a mapping between states to actions.

Q-Learning is a reinforcement learning algorithm. It is the flow described in the following pseudo code:



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

2.2. SCML Negotiation Terms

A negotiation issue - something we're negotiating on. Can be either price or quantity.
An offer - a pair of (unit price, quantity).

2.3. Negotiation-Adapted Q-Learning

2.3.1. General Settings

Important general settings:

1. Q-learning is formalized for discrete states and actions. Since in this problem we have continuous values for some items (such as price), we apply a **discretization technique** in order to make Q-Learning usable for the problem and also in order to optimize memory and runtime performance (trading off with learned policy's quality).
2. Since opposite agents are expected to be different, we apply a Q-Learning process per opposite agent.

3. Q-Learning algorithm is implemented with a ε-greedy strategy where the exploration probability drops as simulation progresses; this is in order to enable more exploratory nature on early stages to be able to escape local maxima and pursue a higher profit in the longer term and eventually reduce the exploration to maximize profit in obtained settings in later parts of the simulation.
4. A Q-Learning process per opposite agent is doing the learning loop throughout the entire simulation run. In Q-Learning terminology, an episode of the algorithm is a single negotiation process and a step of the algorithm is a pass between states of the negotiation process, within the negotiation process.

In order to apply Q-Learning, we need to define the problem in terms of states, action and rewards. These will be defined next.

2.3.2. <u>States</u>

In the OneShot game, our agent tries to maximize its profit by having negotiations where the sell price is higher than the buy price. A negotiation happens in a single step of the “world” (simulation). Above we mentioned that there will be a learning process per opposite agent, thus the Q-Learning model will be applied to a bilateral negotiation process of our agent against another single agent.

During a bilateral negotiation, our agent’s “environment” has the following properties:
- Current needs - requirement to sell/buy items as specified by the exogenous contracts;
- The current offer we see from the opposing agent we’re negotiating with - this is only a part of the state if we’re in the middle of the process (in the beginning, when we’re proposing an offer, we still don’t have a counter offer to consider)

From these environmental properties we derive the states possible in the system:
- Initial state options:
    - **n** - Initial quantity requirement specified by the exogenous contract on the step. It can be a value within 0 to the number of production lines (this is defined by the game rules). This is the initial state in case propose() is called first, i.e. in the beginning of the negotiation the system called our agent to give an initial proposal.
    - **(po,qo, n)** - in addition to the initial quantity requirement, we have an initial counter offer (meaning the opponent was called first and the first time the system called us, it is with the respond() method).
      po is the unit price proposed and qo is the quantity proposed.
      po,qo value ranges are dictated by some minimal & maximal value per of these issues, defined by the system per agent.

- Intermediate state options:
    - **(pm,qm,po,qo,n)** - a state where we previously offered (pm,qm) (unit price of pm and quantity of qm) and got a counter offer of (po,qo) and our current external needs are n. Taking (pm,qm) in the state is not mandatory, but since agents in the general case can be stateful players, taking this into account may improve the learning of their behavior. As specified before, the pm,qm,po,qo po,qo value ranges are dictated by

some minimal & maximal value per of these issues, defined by the system per agent.

- Terminal (final) state options:
  - **END** - negotiation ended without acceptance, i.e. there's no deal (either our agent ended it or the opposite one did).
  - **ACCEPT** - negotiation ended with acceptance, i.e. there's a deal (either our agent accepted a counter offer or the opposite one accepted our offer - one agent is going to sell to the other).

2.3.3. Actions

The actions executed by the model are:
- **(p,q)** - an offer of q items in price p per item;
- **"end"** - the agent uses this action in response to a counter offer - in order to end the negotiation without accepting.
- **"accept"** - the agent uses this action in response to a counter offer - in order to accept the counter offer, hence a negotiation ends successfully - a deal is signed.

2.3.4. Rewards

The "rewards" are naturally related to what the agent gains from the negotiation eventually. This means that the profit achieved during a successful negotiation will affect the rewards considered by the algorithm. This makes sense, since we want the agent to learn how to maximize its profit and Q-Learning maximizes the reward, so it is just logical to directly define the reward using the profit. We get the following immediate reward assignments when associated with moving to states:

State transitions	Reward	Reason
To intermediate state	0	No profit - we didn't do any profit during this transition, nor do we know how to measure it's contribution to making final profit
To ACCEPT state	A function of the profit earned*	Profit is a direct way to estimate reward

*The profit being made is calculated using the following method:
- In case you're in L0 level ("seller" - you profit from selling with a price bigger than the exogenous price):
  $Profit = (agreed\ price_{unit} - exogenous\, price_{unit}) * quantity - production\, cost$

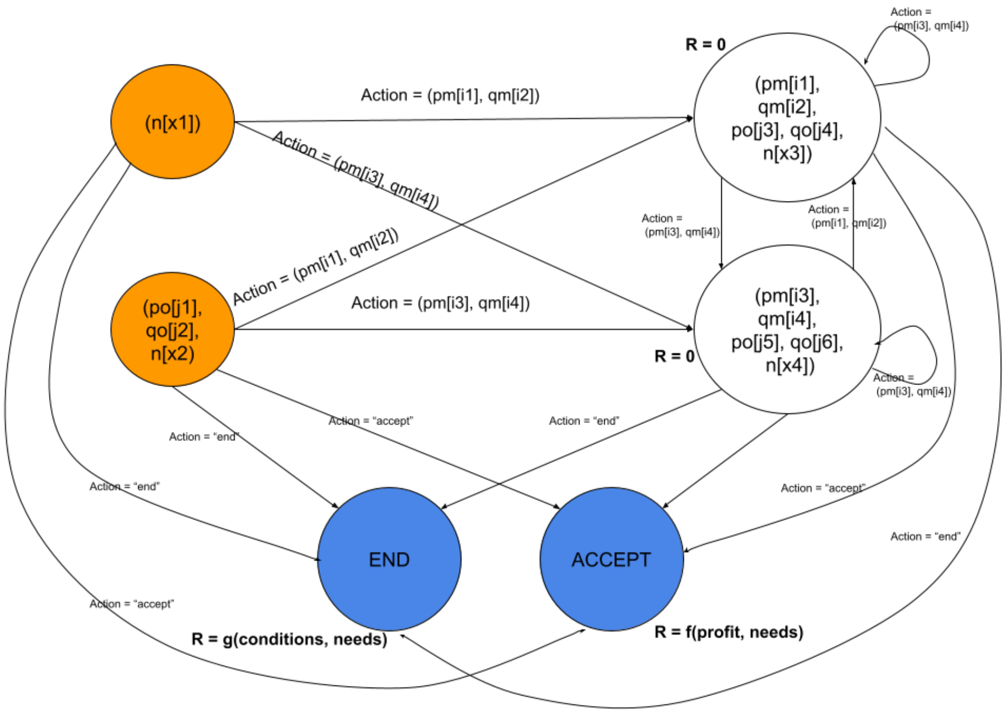
- In case you're in L1 level ("buyer" - you each from selling with a price smaller than the exogenous price):
  $Profit = (exogenous\, price_{unit} - agreed\ price_{unit}) * quantity - production\, cost$

Further reward design is applied in order to make the agent learn to better (and hopefully, faster) assign correct significance and meaning (either positive or negative) during the simulation. The following considerations are taken in account:

1. When the agent is not fulfilling its whole exogenous quantity requirement, or in case it exceeds this amount, it is penalized. We can take this penalty into account in order to make the reward more realistic. Since the deviation is a result of all negotiations in a simulation step, the penalty for the entire deviation is scaled by the amount of negotiations executed successfully so far in the simulation step.
2. In the case that negotiation ends with no acceptance, we can naively say that the reward is 0, or maybe also take into account the excess penalty as discussed above. We expect the process to eventually optimize for maximizing profit, in this case. However, this may take a lot of simulation steps to converge and the simulation may be limited to fewer steps than needed (we can't know this ahead). Even if learning a pre-trained model - we still want reduce the convergence time. So in this case we tried optimizing the rewarding, to give different significance to different situations related to transitions into the END state. The situation differ by the following statuses:
    - Needs: how much we're left with to trade (or how much we exceeded) - being far a way from needs=0 in case negotiation fails, is usually penalized
    - What was the difference between our last last offer and the opponent's last offer, in relation to our needs. In case we don't have too much left, a large deviation is rewarded higher: we still have something to trade, but not so much to the level it will "disappoint us" - on the contrary, accepting a very different offer from opponent is likely to "disappoint us" more, since we conceded where we don't have a lot to lose. This is of course dependent on the deviation value and the needs: as the deviation goes down and needs go up, rewarding is assigned a lower value.

2.3.5. <u>Markov Decision Process</u>

As a result from modeling the states, actions and rewards as discussed - we get the following state diagram:



(n[x1])
(po[j1], qo[j2], n[x2])
(pm[i1], qm[i2], po[j3], qo[j4], n[x3])
(pm[i3], qm[i4], po[j5], qo[j6], n[x4])
END
ACCEPT
R = 0
R = 0
R = g(conditions, needs)
R = f(profit, needs)
Action = (pm[i1], qm[i2])
Action = (pm[i3], qm[i4])
Action = (pm[i1], qm[i2])
Action = (pm[i3], qm[i4])
Action = (pm[i3], qm[i4])
Action = (pm[i3], qm[i4])
Action = (pm[i1], qm[i2])
Action = (pm[i3], qm[i4])
Action = "end"
Action = "accept"
Action = "end"
Action = "end"
Action = "accept"
Action = "accept"
Action = "end"

In orange - initial states;
In white - intermediate states;
In blue - terminal states;

Note that the initial states and intermediate states in this diagram are just examples; They are replicated across all possible values of p,q,n (as discussed before when we presented the states in detail).

2.4. <u>Further Executed Improvements</u>

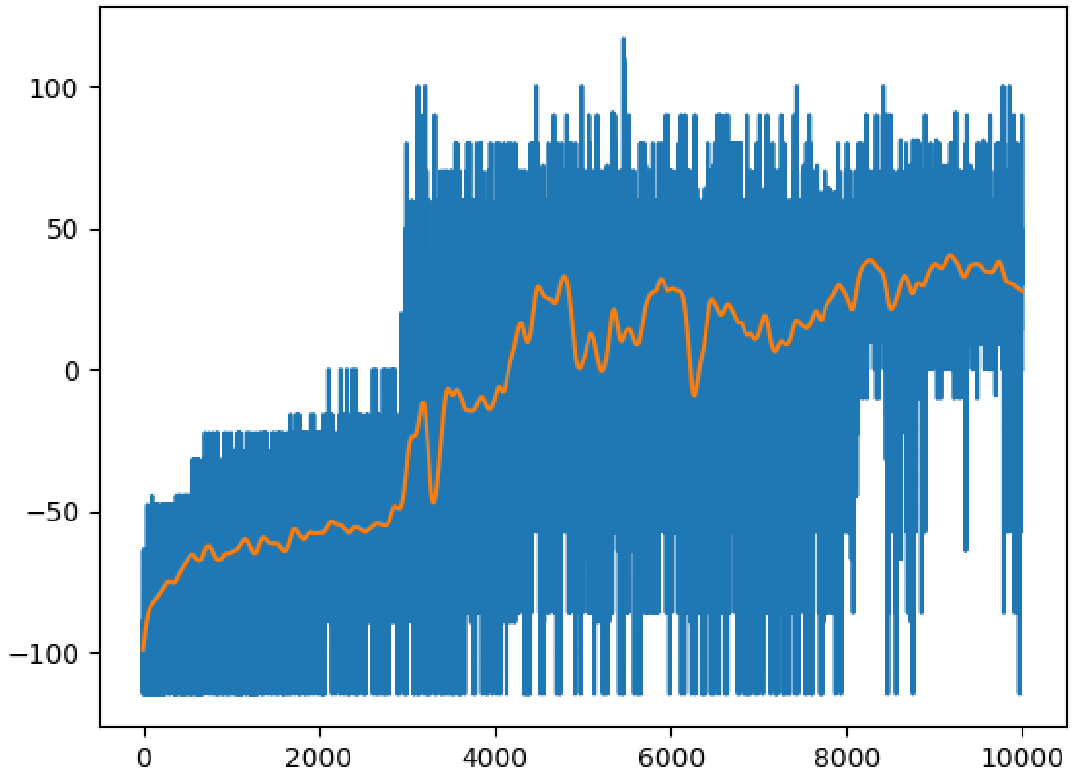
In order to better improve performance and provide faster learning, we applied additional techniques:

- Q table initialization that "prefers" (gives higher value) to more profitable actions.
- Pre-training: we added the ability to store the Q table learned by the agent and load an existing table from a file. We trained the agent against LearningAgent/GreedyOneShotAgent on 1-on-1 scenario for long period which seemed to stable a good learning behavior (high, stable rewards). The Q-table was saved to file and loaded as pre-trained model on next runs.
  Note that since the Q-table states depend on the world configurations and may even be per agent, this is just a "best effort" approach - it may not entirely contribute as state and action values might be different on real competition. However, it is still a good heuristic for initialization.
  We also added the option to gather unused (state,action) Q values into the newly learned Q table. This enables us to have more chance during trainings to build a robust database that cover a lot of possible states and actions, fitting to many configurations.

3. Evaluation

The agent was put to compete against GreedyOneShotAgent and in most cases performed better in one-on-one controlled scenarios and tournament runs. It was also superior to LearningAgent in most one-on-one controlled scenarios.

Here is an example of a graph showing rewards per episode (in blue - original signal, in orange - smoothed) - we can see the obvious learning curve: the agent learns to improve it's rewards as simulation goes forwards and eventually learns a stable, relatively highly rewarding strategy:



4. Further Possible Improvements

The following more sophisticated improvements are possible:

1. Hyper-parameter search using an upper reinforcement learning layer or bayesian optimization.
2. Using Deep Q-learning - the Q-table poses a constraint due to it's possible size, forcing lower accuracy in state and action spaces quantization and thus limiting the learning. DQN may help being able to approximate a lot larger table without having to store it or initialize directly.s

However DQN may be a challenge to implement, maybe even not possible since neural networks require specific and high computation resources we can't assume are available (and if not, this means long runtime which are also limited in the competition).

3. Mapping the Q-table states and actions to fixed values (scale it linearly) on saving of the table and doing an inverse mapping on load (specific to the values of the new game) - in order be able to directly learn from one simulation to the other and not just "best effort"
4. Examining other continuous space and action state RL algorithms.