Supply Chain Management League 2020

By: Dolev Mutzari

ID: 208984112

Table of Contents

Introduction

The SCM world simulates a supply chain consisting of multiple factories that buy and sell products from one another. The factories are represented by autonomous agents that act as factory managers, and the challenge is to submit an agent that is the most profitable.

Solution Design and Overview

The agent We have developed is built component-wise. This enables programming different tasks of the agent separately, and by doing so we can also evaluate each component independently. In addition, this kind of object-oriented design is more elegant and easier to debug. My final agent consists of 3 (+1) main components:

1. Trading Strategy

This component is responsible for deciding the quantity and price for buy and sell at every time-step. It specifies some goal which later the other components will try to fulfill. We divide this component further into 2 sub-components:

- **Trade prediction strategy** – a pre-negotiation component that decides the quantities and prices to negotiate about based on prediction of future market behavior. My implementation of this component is based on a learned deep network for prediction. The specification will be described in the implementation details section.
- **Signing strategy** – a post-negotiation component that decides what agreements to sign as contracts. We simply sign all the contracts for two reasons:
  - Firstly, empirically, learning a signing strategy wasn't helpful, perhaps because most concluded contracts are also signed.
  - Secondly, we count on the negotiation component to accept and offer only contracts that my agent can handle. Thus, once a contract is concluded, there is no reason for not signing it.

  We believe the signing strategy is important only if in the negotiation strategy you purposefully make offers you cannot execute in hope that the other agent will respond by making a counteroffer that is more beneficial.

2. Negotiation Control Strategy

This component is responsible for proactively request negotiations, responding to negotiation requests and conducting concurrent negotiations. We divide this component further into 4 subcomponents:

- **Negotiation manager** – a pre negotiation component that decides which negotiations to accept and which to engage in**.** Our negotiator manager consists of 2 sync controllers, one that handles the selling contracts and one for buying contracts.
  Note that this means that the negotiation manager doesn't synchronize selling and buying. As an example, if there was a great sell deal that the seller sync controller made, maybe it is okay to buy at a higher price than what the initial trading strategy has demanded. Such a logical step is not possible in our negotiation manager, and can be considered for a further optimization of our submitted agent.
- **Sell/buy sync controller** – A controller is a class that manages multiple negotiations. Both of our controllers use the above trading component to set the goal of the negotiation – how much products to buy/sell and at what maximal/minimal cost/price. The SAOSyncController in addition

synchronizes the process of offering and responding to offers. By inheriting from this controller class, our controllers are used for syncing multiple contracts on the same time.
Our controllers find the best offer currently received. If the negotiation is about to end or this offer has a utility above some threshold, it is accepted. Otherwise, it is sent to all other negotiators as our new offer. We respond to the partner that gave us the best offer with the offer with the maximum predicted utility.

- **Negotiation Algorithm** – Our negotiation algorithm uses 2 utility functions:
  - The first utility function is used for evaluating an offer. This is a simple linear function of the price (positive for selling and negative for buying) and the quantity. We need this utility function in order to decide which offer is best.
  - The second utility function is used for predicting the response of the opponent negotiator. Given this function, we can predict the opponent response for several offers, and decide which one to take (according to the first utility function which evaluates offers).

  In the implementation details section, we will describe the implementation of the negotiation algorithm based on these 2 functions.

3. *Production Strategy*

Decides what to produce at every time step. We use SupplyDrivenProductionStrategy, which basically converts all inputs possible into outputs. The reason to do so is that We count on the trading strategy and negotiation mechanism to buy only necessary inputs.

Furthermore, this incentivizes the agent to have more negotiations and therefore earn more money. This works better than DemandDrivenProductionStrategy which produces only the products that are needed for the next day, because in this scenario we are less ready in advance for making additional negotiations or to handle unexpected offered negotiations from the opponent agents.

4. *Save History*

This component is used for saving the history or view of an agent in the SCML world. This helpful "plugin" is useful for mainly 2 reasons:

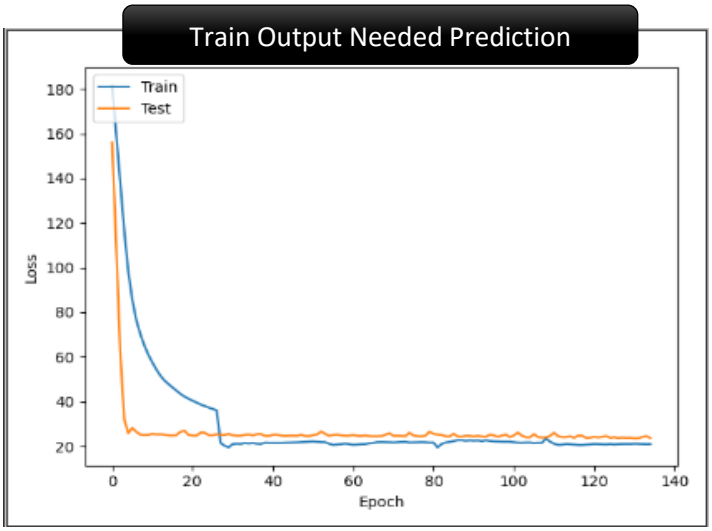
- Data gathering – the history wrapper enables us to record everything that an agent saw in the simulation. The information and experience he gained can be used for learning and training afterwards.
- On runtime, the agent can use the data he saw in the past in order to predict future behavior. This applies both for the predicting future market behavior for the trading strategy, and for predicting the opponent's reaction to offers best on the contract bargaining so far.

Implementation Details and Results

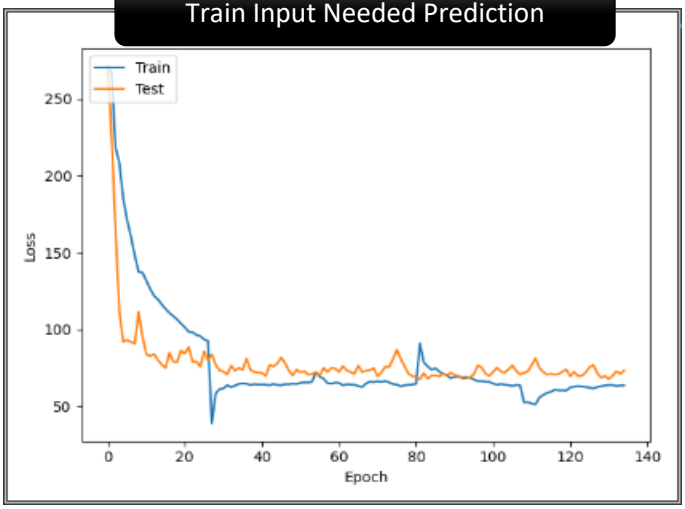
Learning a trade prediction network – we are interested in predicting the amount of inputs needed and outputs needed on each time step in the future. These predictions will be used as explained by the negotiation mechanism. We should discuss about both feature extraction and learning phase. Fortunately, the feature extraction in this case is mainly done by the save history wrapper component. The main problem is that this is allegedly not a tagged data, so it is not clear how supervised learning can be used.
However, using offline learning here works quite well. The main idea is that, after a run of the SCML world is finished, we know in retrospect, on each time-step, how many resources were needed on each

time step. We can use this data for learning, and then the prediction used on the learned model can be done online.
We therefore chose to learn a recurrent network, suitable for propagating memory from the past for future prediction. We trained 2 separate networks, one for predicting the inputs needed and one for predicting the outputs needed. The deep network design consisted of a 2 layered LSTM network, and an MLP squeezing layer. We now present the graphs of loss vs number of epochs.

Train Output Needed Prediction



Train Input Needed Prediction



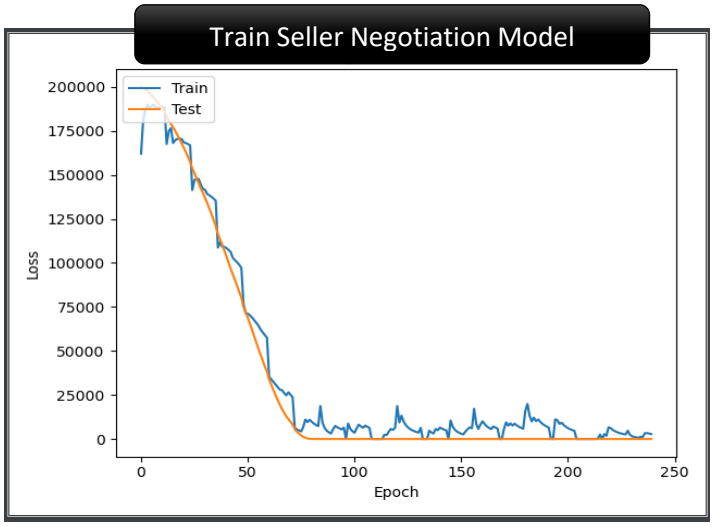
Opponent Modeling - Learning an opponent prediction network

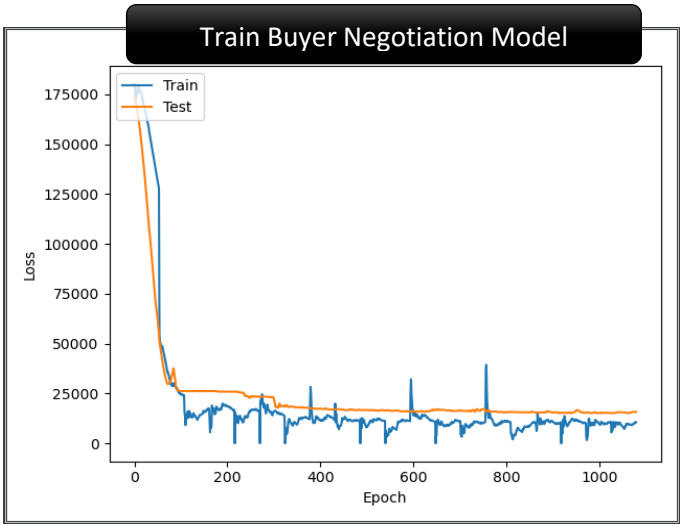
We are now interested in learning a function that predicts the behavior of an arbitrary opponent. This function should naturally depend on the history of the current negotiation. This is done by an additional save history component. Furthermore, this function should depend on the goal specified by the trading component, and the current SAOState (negotiation step, relative time and so on).

Similarly, using offline learning we can reduce to a supervised learning task. Indeed, after a simulation of the SCML world is complete, we know the retroactively the responses of the agents to all offers. We can thus try to learn the above deep network prediction function.

The structure of the deep network used here is the same, except for one key mitigation. The trading goal, is used as an auxiliary feature that is appended in time-step of the contract agreement. In this way, we can effectively train multiple networks in parallel.

Even more importantly, this is a natural way of transfer learning. Indeed, learning how to bargain when the goal is to buy X resources at maximal price of Y for $(X, Y) \in \{(X_1, Y_1), (X_2, Y_2)\}$, our network can generalize and deduce how to work for (X_1, Y_2) even if there was no such data in the train set. . We now present the graphs of loss vs number of epochs for these networks.





Evaluating an offer

As for the utility function for evaluating offers, a simple linear network does the job. All that this utility function should supply is encouraging the buyer to buy as much resources as possible, a the lowest price possible. Similarly, we can define a utility function for a seller by changing the sign of the price. The main observation or heuristic is that if we considered all the data carefully in order to predict the opponents behavior, the only thing left to consider, roughly speaking, is the offer itself. Note that we don't consider the offer's time, which is a place for improvement.

Bidding Strategy - How to make an offer?

We view the $propose$ method as an optimization problem. We would want to find offer, that maximizes the utility of the offer the opponent agent will respond. Clearly, we need both functions mentioned earlier in order to do so. We will now discuss how to smartly scan the space of optional offers.

The most naïve approach is to go over all the possible triplets of time, quantity, price, and offer the one that gives the best predicted utility. However, this is not a feasible solution. The first observation one could take is the following: when all the rest of the parameters is fixed, if the opponent buyer didn't agree to buy our product in price p, he probably won't agree to buy our product in price higher than p. Furthermore, we as sellers would prefer to sell in the highest price the opponent would accept. Thus, instead of going over all triplets, we can use binary search for the price component.

The second observation to take is that we shouldn't consider all possible times for negotiations, but only a finite horizon we can specify. This horizon is a hyperparameter for our agent, currently set to 5. In any case, early contracts are less risky, in the sense that we are more certain of our abilities in the near future. As for the quantity, we go over all possible configurations, from 1 to the predicted amount needed.

Acceptance Strategy – How to respond to an offer?

The $respond$ routine is implemented as follows: given a SAOState and an offer, we call propose routine, and if we get a higher utility, we reject, and otherwise we accept.

Elicitation Method – How the agent deals with preference uncertainty?

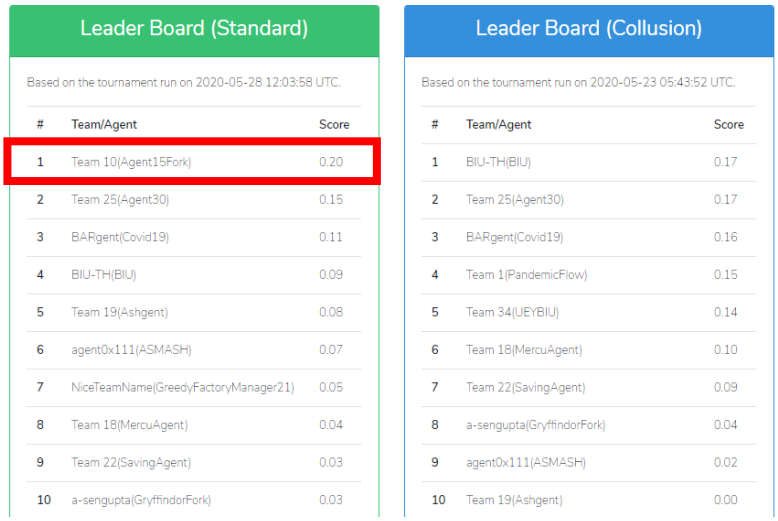
There are two sections where the agent should have an elicitation method.

The first is the propose routine, where our agent simulates the response of the opponent in his head for several proposals. The utility function is used for elicitation. Since it returns a float number, equalities should not occur, and even if they had, they are equivalently good for the agent (we may try to further improve our utility function, but equalities in general may always occur).

The second place is the sync controller, that should decide which offers to accept and which to continue negotiate. Our sync controller takes the best offer he got (decided by the utility function) and passes it as a response to the rest of the agents. Here again, equivalence in the utility should not occur, but if it does, we decide arbitrarily (basically, the way Python 3 decides which argmax to return, probably the first one on the list).

Evaluation

The evaluation of the agent was done by running small tournament of the agent against built-in agents such as the DecentralizedAgent and the BuyCheapSellExpensiveAgent. Moreover, the online leader board was used to test the performance of our agent against the rest of the competitors. At some point, our agent was at the top of the standard leader board!



Leader Board (Standard)

Based on the tournament run on 2020-05-28 12:03:58 UTC.

#	Team/Agent	Score
1	Team 10(Agent15Fork)	0.20
2	Team 25(Agent30)	0.15
3	BARgent(Covid19)	0.11
4	BIU-TH(BIU)	0.09
5	Team 19(Ashgent)	0.08
6	agent0x111(ASMASH)	0.07
7	NiceTeamName(GreedyFactoryManager21)	0.05
8	Team 18(MercuAgent)	0.04
9	Team 22(SavingAgent)	0.03
10	a-sengupta(GryffindorFork)	0.03

Leader Board (Collusion)

Based on the tournament run on 2020-05-23 05:43:52 UTC.

#	Team/Agent	Score
1	BIU-TH(BIU)	0.17
2	Team 25(Agent30)	0.17
3	BARgent(Covid19)	0.16
4	Team 1(PandemicFlow)	0.15
5	Team 34(UEYBIU)	0.14
6	Team 18(MercuAgent)	0.10
7	Team 22(SavingAgent)	0.09
8	a-sengupta(GryffindorFork)	0.04
9	agent0x111(ASMASH)	0.02
10	Team 19(Ashgent)	0.00

Fortunately, since our agent was built component-wise, we could test the effectiveness of each mechanism independently. This allows us to test the contribution of each mechanism by itself. Here are some tests we ran of our negotiation mechanism contribution to our agent:

N. scores = 36 N. Worlds = 12

	agent_type	score
0	agent.NegotiatorAgent	0.474338
1	scml.scml2020.agents.decentralizing.DecentralizingAgent	0.467026
2	scml.scml2020.agents.bcse.BuyCheapSellExpensiveAgent	-1.00499

Finished in 4m:26s

N. scores = 48 N. Worlds = 24

	agent_type	score
0	agent.NegotiatorAgent	0.258903
1	scml.scml2020.agents.decentralizing.IndDecentralizingAgent	0.103811
2	scml.scml2020.agents.random.RandomAgent	-0.421355

Finished in 4m:29s

Here are some tests we ran for our trading mechanism contribution to our agent:

N. scores = 48 N. Worlds = 24

	agent_type	score
0	agent.TradingAgent	0.326644
1	scml.scml2020.agents.decentralizing.DecentralizingAgent	0.25383
2	scml.scml2020.agents.bcse.BuyCheapSellExpensiveAgent	-0.185729

Finished in 1m:34s

N. scores = 48 N. Worlds = 24

	agent_type	score
0	agent.TradingAgent	0.343277
1	scml.scml2020.agents.decentralizing.IndDecentralizingAgent	0.0990075
2	scml.scml2020.agents.random.RandomAgent	-1.37484

Finished in 2m:13s