g-RIPS Sendai 2023

NEC Group

Final Report

Authors:
Isaac Brown[1]
Kevin Zhou[2]
Soichiro Sato[3]
Soto Hisakawa[4]

Mentors:
Dr. Masaki Ogawa[+]
Dr. Yasser Mohammad[*]

[1] The Ohio State University
[2] University of Illinois Chicago
[3] Musashino University
[4] Kyushu University
[+] Academic Mentor, Tohoku University
[*] Industrial Mentor, NEC

August 8, 2023

Contents

1 Background

1.1 SCML

The purpose of this subsection is to provide a brief summary of the mechanics of the Supply Chain Management League game. Details can be found in the description documents on the SCML website: https://scml.cs.brown.edu/.

The Supply Chain Management League (SCML) is an annual league of the Automated Negotiating Agents Competition (ANAC), in which participants submit negotiating agents that compete within a simulated supply chain (an *SCM world*) consisting of multiple factories which buy raw materials from and sell final products to one another. Each factory is assigned an agent that manages its negotiation strategy, and after a certain number of days, each agent is given a utility (i.e., profit) calculated based on the transactions it participated in, as well as costs for manufacturing and penalties for either selling items it was unable to produce or producing excess items it was unable to sell. The final goal of the competition is to produce higher average utility than the other competitors over many simulations.

The negotiation protocol used in SCML is a variant of Rubenstein's alternating offers protocol [4], in which a pair of agents take turns making offers for a finite number of rounds and/or a finite amount of time. One agent submits an initial offer, after which the agents alternate to choose one of the following actions:

1. Accept the offer
2. Reject the offer and submit a counteroffer
3. End the negotiation without any agreement

The process repeats until either an agreement is reached, one side ends the negotiation, or a pre-determined negotiation round limit is reached, in which case the negotiation ends without any agreement.

The simulation proceeds for some fixed number of days. On each day, an agent receives exogenous contracts, which dictate that it must buy (sell) a certain number of raw materials (final products) at a particular price. The agent then proceeds to negotiate concurrently with other agents using the above protocol. Contracts are executed once all negotiations have ended, before moving to the next day.

1.2 Reinforcement Learning

Reinforcement learning (RL) is one of the three basic machine learning paradigms, alongside supervised learning and unsupervised learning. It allows an agent to learn by trial and error in an interactive environment, using feedback from its own actions and experiences.

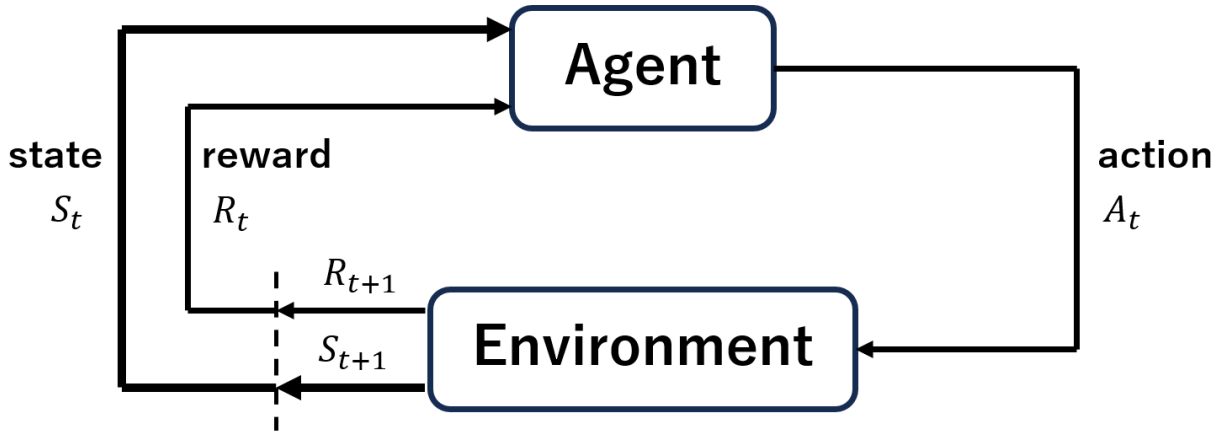

Figure 1: A typical reinforcement learning cycle

Figure 1 depicts the typical reinforcement learning cycle. The basic elements of RL include:

- Agent — A decision-making and action subject
- Environment — World in which the agent operates
- State — Current situation of the agent
- Action — The behavior of the agent
- Reward — Feedback from the environment in response to an agent's action
- Policy — Method to determine an action based on the agent's state

The agent and the environment send and receive information (states, actions, and rewards) to and from each other. By repeating this interaction, the agent can use their observations (states and rewards) to update their policy and learn better behaviors for obtaining the highest reward possible. This is the general idea behind reinforcement learning in policy optimization. The goal is then to maximize long-term benefits, in our case being profits in the SCM world.

The Markov decision process (MDP) comes into play when asking how to mathematically formulate this reinforcement learning problem. An MDP is defined by the 4-tuple $(S, A, \mathcal{P}_A, \mathcal{R}_A)$ where S is the set of all possible states[8], A is the set of possible agent actions, $\mathcal{P}_A$ is a set of conditional transition probability functions $P_a(s, s') = \Pr[s_{t+1} = s' | s_t = s, a_t = a]$, and $\mathcal{R}_A$ is a set of reward functions $R_a(s, s')$ returning the immediate (or expected immediate) reward from transitioning from state s to state s' under action a. This is a state transition model in which the probability of reaching the next state is determined only by the current state and action (independent of past states).

If the underlying MDP of an SCM game is known, this can yield optimal policies with respect to maximizing SCM rewards. The problem however is that a large majority of information about the SCM environment is not known to the agent, either being completely hidden or only being accessible in a restricted form (for example, every 5 days on a bulletin board). Through the use of RL, one can approximate the underlying MDP, despite the hidden information, and use this approximated form in deciding actions and updates to policies.

Reinforcement learning, like many ML frameworks, has many points of subtlety that must be considered carefully. In particular, using excessive state information when making observations leads to increased training time and the risk of overfitting to training data. At the same time, using insufficient state information leads to poorer agent performance than otherwise could be attained. Therefore, the decision of what information to include in agent observations is very delicate. This is especially apparent in the SCM game where even though much of the world information in SCM is hidden from agents, there is still a large amount of information that can be considered in agent observations, such as all current negotiation details, negotiation histories with all partners, current shortfall and disposal penalties, bulletin board information such as agent balances and total exogenous quantity data, etc.

With the hopes of developing agent negotiation strategies that perform well in a variety of settings, we pursued RL methods. Specifically, Q-learning was the first algorithm attempted before we began implementing Proximal Policy Optimization (PPO). Currently, the best RL agent we have is one trained with Q-learning, however both methods have obvious rooms for improvement in the tuning of hyperparameters, action spaces, and observation spaces.

1.2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization is a family of reinforcement learning algorithms introduced by OpenAI in 2017 [7]. These algorithms are *policy gradient* methods, which means that they operate maintaining some (differentiable) policy (often modeled using a neural network), recording the reward of an action taken by the policy, and using the reward to compute a label (often an estimate to the *advantage* of a given action) in order to update the policy. Prior to the introduction of PPO, policy gradient methods often suffered from being very sensitive to the choice of stepsize—a stepsize too small will learn extremely slowly, while a stepsize too large can cause drops in performance if the gradient update becomes too large, since we may stray away from a policy that is known to be good.

PPO addresses this issue by utilizing ideas from Trust Region Policy Optimization (TRPO). Specifically, PPO modifies the objective function being optimized to prevent large updates to the policy. It accomplishes this by "clipping" probability ratios in the objective function to remove incentives for changing policy action probabilities by more than some fixed relative amount. The exact details of how it accomplishes are outside the scope of this report, and the interested reader can refer to the original PPO paper for a detailed explanation. The original paper also includes data from tests indicating that PPO performs comparably or better than many other state-of-the-art reinforcement learning algorithms. Additionally, PPO has advantages over other prior attempts to address the issues with other current popular policy gradient methods, such as being far simpler to implement and applicable to a wider range of environments than algorithms such as TRPO and ACER.

1.2.2 Q-Learning

Q-Learning is a reinforcement learning algorithm whose policy is dictated by a learned table of values called a Q-table. For each observation state, the Q-table keeps a list of values measuring the "quality" of

taking action a at the given state. Due to this method of having a table of all state-action pairs, the default Q-learning method only applies to environments with discrete action and state spaces. Luckily, this isn't too much of an issue in SCM since many important world characteristics have discrete values and all actions (negotiation decisions) are discrete. At the conclusion of learning, the default policy returned by Q-learning is a deterministic one in which at a given state, the action with the highest Q-table value for that state is executed. [6].

Suppose that the agent in state s_t chooses an action a and the state transitions to s_{t+1}. At this time, $Q(s_t, a)$ is updated as follows.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \text{max}_a Q(s_{t+1}, a)] \tag{1.1}$$

- α — Parameter that determines the size of the update $(0 < \alpha \leq 1)$
- γ — Parameter that determines how much of the Q value of the transition destination is used. $(0 \leq \gamma \leq 1)$ The closer it is to 0, the more importance is given to immediate rewards.

In Q-Learning, as learning progresses, the TD error (the error between the current value estimate and the value estimate at the next time step) is learned to be smaller.

Our decision to use Q Learning came about as we considered it easy to implement in code and adjust for learning as needed.

2 Heuristic-based Agents

Our first attempts at designing agent strategies centered around making modifications to the most recent SCML OneShot winner, QuantityOrientedAgent (QOA), with the goal of beating QOA and the second place finisher CCAgent (CCA) in direct competition. For these initial strategies, we used QOA (whose strategy is written below) as a framework for our agents:

- **Response Strategy:**
  - If my quantity needs are satisfied, end the negotiation.
  - Else if my opponent's offer is for a quantity that's less than my current quantity needs, accept the offer.
  - Else if we are near the end of the negotiation period and my opponent's offer takes me closer to my goal quantity so that

    $$|\text{my\_quantity\_needs} - \text{opponent\_offer\_quantity}| < \text{my\_quantity\_needs},$$

    then accept the offer.
  - Else, reject the offer.
- **Proposal Strategy:**
  - For quantity, if my needs are significant, offer half of what I need. Otherwise, ask for the exact quantity that I need.
  - For price, if we are near the end of the negotiation period, offer the price that's best for my negotiating partner. Otherwise, insist on the price that's best for me.

Our variations on QOA began with greedy adjustments before creating what we believed to be "smarter" agents that took their level into account. Details and code of all agents listed below can be found in the github repository"grips-nec-2023":

- Greedy QOA
  - Tries to insist on obtaining quantity at its best price by keeping a record of when a negotiation partner offers quantities that it's happy with (less than our current needs).

  - Only accepts offers from partners when the quantity is less than its needs and the price is its best price.
  - Uses default QOA proposal strategy when an opponent has not offered a quantity it is happy with.
- Half-Greedy QOA
  - Same behavior as greedy QOA, except that when countering an opponents offer with our best price, it divides the opponents recorded "happy" quantity by 2 if they requested a large amount of goods (≥ 5).
  - This division by 2 is based on the defualt QOA's proposal strategy.
- Nice QOA
  - Same behavior as QOA, except that it always offers the price that's best for the opponent, regardless of the negotiation round.
  - By far the simplest agent strategy with the shortest code amongst all agents listed here, including QOA.

Upon implementing these agents, we saw results that were lower than we initially expected which lead us to investigating the SCML world further. After analyzing world contracts over many simulations, we realized that the total exogenous quantity input to the world was always greater than or equal to the exogenous quantity output. In an environment where agents are performing at somewhat of a high level, this should mean that level 0 agents are in a more desperate position than level 1 agents. We then decided to begin developing agents that had level-dependent behavior to account for this input-output gap:

- Nice/QOA Hybrid
  - At level 0, uses "Nice QOA" strategy from above.
  - At level 1, uses default QOA strategy.
- Half-Desperate/Half-Greedy Hybrid
  - At level 0, uses a very similar strategy to default QOA, except that when an opponent offers a quantity that's greater than our needs, we now record the offered amount and make slight adjustments to counter offers based on that.
  - At level 1, uses the "Half-Greedy QOA" behavior from above.
- QOA/Half-Greedy Hybrid
  - At level 0, uses the default QOA strategy.
  - At level 1, uses the "Half-Greedy QOA" behavior from above.

After running world simulations in attempts to determine which of these agents was best, the results yielded no decisive conclusion. Specifically, agent performance is massively dependent on the field of opposing agents they are playing against. The best results obtained from this were that some agent strategy (say strategy X) consistently won games when the opposing agents were from a fixed set (say $\{A, B, C\}$), but when a new agent is added to the opponent set ($\{A, B, C, D\}$), the agent's dominance disappeared. Our inability to develop agent strategies that dominated opponents in a wide-variety of opposing strategy settings (even when the total set of opposing strategies is known in advance) led us to considering machine learning strategies for agent strategy development.

3 PPO Agent

One attempt at creating an RL agent for SCML is a "pure" RL approach, where we allow the agent to perform any action and give it zero initial guidance, so it must learn its policy entirely from scratch. To do this, we utilized the RL interface introduced in SCML v0.6.1, which consists of six primary components:

1. the *environment*, which is a custom Gymnasium environment, that handles simulating the SCML world and generating observations and rewards
2. the *world factory*, which defines a method for generating SCML worlds according to desired specifications

3. the *observation manager*, which defines how to data from the SCML world and encoding it in a way that is compatible with Gymnasium's observation spaces
4. the *reward function*, which determines the reward given to the agent based upon the change in state induced by a given action
5. the *action manager*, which defines how to decode an action generated by a Stable Baselines 3 policy (usually in the form of a 1-d NumPy array) into a set of negotiation responses
6. the *agent*, which is able to execute trained RL policies within an SCML world

An environment must contain a world factory, which is responsible for generating SCML worlds upon environment reset. The environment must also have an observation manager and reward function, which compute and encode the feedback given to the agent after an action is taken, as well as an action manager, which encodes the action generated by the agent. The observation and action managers must be compatible with the world factory in the sense that the managers must be able to handle any world generated by the factory; e.g., if the world factory is able to generate a world in which the agent is placed in level 0 and has 6 negotiating partners, the observation and action managers must be able to encode observations and decode actions in this setting.

Before detailing the results of our work, we make some comments on which of the components should be modified in order to improve performance of agents. The environment and agent exist primarily to allow the various frameworks (Gymnasium, Stable Baselines 3, and SCML) to work together and do not need to be customized. In addition, while it may seem reasonable to alter the action manager, there does not immediately appear to be a significant benefit from doing so. Attempts were made at adjusting the action space so as to make it easier for the RL agent to accept offers from negotiating partners and while we did not conduct any formal statistical analysis of how this compared to the default action manager, there were no performance differences so obvious as to lead us to investigate further.

Developing a custom world factory allows for custom training environments, since the world factory determines the worlds that the agent will be placed into during training. This can be useful if (for example) one wants to train the agent against other SCML competitors which are known to have good performance.

The components that are likely to have the greatest impact on agent performance are the observation manager and reward function. The observation manager has access to the current OneShotState, which contains a great deal of information about the world as seen from the agent's perspective at a given moment in time. It is certainly possible to include all the information contained in the OneShotState in the observation; however, including more information in the observation is likely to increase the sample complexity for learning (since it will take longer to explore a larger observation space), and increases the risk of overfitting to features that are of low relevance. On the other hand, restricting too much information from the observation space can hamper performance if the agent does not have access to information that should inform its strategy. Hence, designing the observation manager involves managing the tradeoff between performance and sample efficiency, by determining the most important features to include in an observation.

The reward function can also have a large impact on performance, since the reward function is what determines which behaviors are incentivized and which are discouraged. In SCML, the goal of the agent is to maximize profit, and so it seems reasonable to use daily profit as the reward function (indeed, this is the default reward function). However, there are some potential issues with this reward function, such as the fact that profit is only calculated at the end of a day when contracts are executed, and so the reward for a good action may not appear until several steps after the action was taken. For this reason, it may be beneficial to alter the reward function (reward shaping) in order to provide better feedback to the agent and allow it to converge more quickly to a strong policy.

3.1 Custom observation managers

In our work, we implement three new custom observation managers.

The first custom observation manager is the `BetterFixedPartnerNumbersObservationManager`. It is based off the `FixedParterNumbersObservationManager` which is included in SCML v0.6.1, but makes some changes to omit extraneous information (some of which may be useful in environments other than SCML 2023, such as the number of active lines), while also including some potentially helpful state information (such as the number of same-level competitors). It encodes:

- all current incoming offers
- the currently needed quantity
- the current round of negotiations
- the current number of same-level competitors
- the current relative simulation time
- the current disposal cost (normalized to a value between 1 and 10)
- the current shortfall penalty (normalized to a value between 1 and 10)
- the relative price increase from input to output products (normalized to a value between 1 and 10)

The second custom observation manager is the `DictBetterObservationManager`. It utilizes a Gymnasium `Dict` space, which provides several benefits over a MultiDiscrete space (which is what is used by `FixedParterNumbersObservationManager`, `BetterFixedPartnerNumbersObservationManager`, and `FixedPartnerNumbersObservationManager_History`). The first is that it can combine `Discrete` spaces (which encode a finite number of distinct values) and `Box` spaces (which encode intervals), which allows some features that originally needed to be normalized to be expressed as exact values, such as disposal cost and shortfall penalty. The second advantage is that it allows for much greater readability, while not impacting performance at all, since `Dict` spaces are represented in Gymnasium as arrays. `DictBetterObservationManager` encodes:

- all current incoming offers
- the current needed quantity
- the current round of negotiation
- the current number of same-level competitors
- the current simulation step
- the current disposal cost (not normalized)
- the current shortfall penalty (not normalized)
- the production cost
- the current input product trading price
- the current output product trading price

The third custom observation manager is the `FixedPartnerNumbersObservationManager_History`. As mentioned above, it utilizes a Gymnasium `Multi-Discrete` space. This would likely be changed to a `Dict` space for the advantages listed above given more time to work on this project. This observation manager was our first attempt at developing an RL agent that tracked past observation history in making decisions and did so in a highly rudimentary manner. The new information tracked by `FixedPartnerNumbersObservationManager_History` in each timestep is

- the price and quantity of all current incoming offers
- the current needed quantity
- the current round of negotiation
- the current number of same-level competitors
- the relative simulation step (normalized)
- the current disposal cost (normalized)
- the current shortfall penalty (normalized)
- the relative price increase from input to output product

When this information is recorded, the observations from the previous n timesteps are all shifted in order to make room for this new observation, taking the place of the prior update's oldest observation. For example, in an SCMl world with 4 negotiating partners, the above list of observation items includes 15 values (2 values for each of the 4 partners, plus 7 partner-independent values). If the desired observation history length n is set to be $n = 3$, then the actual observation space utilized by the agent has dimension $15 * 3 = 45$ and it tracks the current observation, as well as the two most recent observations in descending order of recency.

3.2 Custom reward functions

The standard reward function simply returns profit from one step to another. Since the outcome of the SCML game is measured by the final profit of the agents, and profit on any given day is independent of profit

on other days in the OneShot competition, this is the most immediately obvious reward function. However, this reward function comes with some issues since rewards are somewhat spare (occurring only at the end of a day and not on each simulation step). To help address this, we implemented two new custom reward functions.

The first custom reward function is the `ReducingNeedsReward`. This reward function alters the standard reward function by additionally rewarding the agent for reducing the absolute value of the difference of input and output contracts. By doing so, the agent is given immediate reward for taking actions that move towards balancing total inputs and outputs, instead of having to wait until the end of the day to observe profits.

The second custom reward function is the `QuantityBasedReward`. This reward function completely ignores profit and only measures quantity. It is inspired by the notion that in the current version of the OneShot competition, quantity matters far more than price, and so the agent is incentivized to focus on securing the correct quantity. `QuantityBasedReward` may suffer from the fact that it does not take into account the primary objective of the OneShot competition. However, it has multiple potential benefits. For one, the fact that profit is a very volatile quantity that is highly dependent on random world variables can lead to the agent learning behaviors that aren't well representative of the actual SCML game environment. Hence, optimizing for quantity may result in greater performance stability. Another benefit of this quantity-focused reward function is that it's highly simplistic, at least when compared to profit-based rewards. Profit is a 2-dimensional reward dependent on both quantity and price, whereas quantity is a 1-dimensional reward, solely dependent on the mismatch of quantity traded and exogenous quantity. While this quantity reward function has a lower performance ceiling than the default profit reward since it completely neglects price, strategies that yield high returns from quantity-based rewards seem to be far simpler than those for profit-based rewards due to the non-consideration of price. This led us to believe that agents using `QuantityBasedReward` should have better initial training performance than those using the default profit rewards or `ReducingNeedsReward`.

3.3 Custom world factory

In our work, we implement a new custom world factory, `FixedPartnerNumberWithOpponentsOneShotFactory`. This world factory is a modification of the default `FixedParterNumbersOneShotFactory` that is meant to allow for passing a collection of negotiating agents that are designated as "opponents" as in an SCML tournament simulation. The factory selects a random subset of opponents, and places one copy of each opponent from the subset in the world (alongside the RL agent), and populates the rest of the world with default agents. This allows the agent to be trained in an environment more closely resembling a tournament setting, and also allows for training against agents that are known to have good performance in order to improve performance.

3.4 Use of PPO

In our work, we used the Stable Baselines3 implementation of PPO (described in § 1.2.1) as the RL algorithm. Practically speaking, it is not difficult to utilize any already-implemented RL algorithm that is able to interact with Gymnasium environments, provided that the algorithm is able to handle the action and observation spaces of the `OneShotEnv`. Stable Baselines3 implements several state-of-the-art RL algorithms, and among their implementations, PPO and A2C (Advantage Actor Critic) are able to handle the MultiDiscrete actions space in the default SCML action manager. We decided to use PPO because it is a well-established algorithm that has been used to achieve good results in many applications, such as in OpenAI Five [5], an AI system designed to play the video game Dota 2.

3.5 Effect of reward shaping - `ReducingNeedsReward` vs. `QuantityBasedReward`

As discussed previously, we introduced two new reward functions to address potential issues with only using profit as the reward. We can see the effect of reward shaping through two agents that were both trained using the `DictBetterObservationManager`, but with the two different reward functions. Figure 2 shows mean reward per episode during training for the agent using ReducingNeedsReward, and Figure 3 shows its score after running in 50 test world simulations. The mean reward per episode shows a large amount of fluctuations in training (albeit with an overall upward trend), due to the volatility of profit as a reward.

Figure 2: Mean reward per episode during training using ReducingNeedsReward

```
Model PPO_L0_4-partners_20230803-144720_1024000-steps_ReducingNeedsReward scores:
[('SyncRandomOneShotAgent', 0.9865932264467818),
 ('SingleAgreementAspirationAgent', 0.8889543832704154),
 ('OneShotRLAgent', 0.7801311137155078),
 ('GreedyOneShotAgent', 0.6383094013967752)]
```

Figure 3: Average scores for ReducingNeedsReward in 50 world simulations

In contrast, Figure 4 shows mean reward per episode during training for the agent using ReducingNeedsReward, and Figure 5 shows its score after running in 50 test world simulations.

The agent trained with `QuantityBasedReward` shows much greater stability in training compared to the agent trained with `ReducingNeedsReward`. This is to be expected since `QuantityBasedReward` only depends on quantity secured, a much more stable value compared to the profit. However, we notice that it plateaus in training after only about 50,000 steps. Additionally, it does not score as well in testing as the agent trained with `ReducingNeedsReward`, likely due to the fact that it no longer takes into account factors such as price, which have a small but still significant effect on overall profit.

It is worth noting that both of these agents still have fairly poor performance, which indicates that there is still much work to be done in terms of optimizing the training process (whether in reward shaping or elsewhere).

3.6 Future work

Our initial results indicate that significantly more work needs to be done to understand how to properly tune the parameters for training in order to obtain good results. To this end, we have written easy-to-use scripts for training and testing that will allow for rapid iteration. It is our hope that with more experimentation and training time, we will be able to develop an effective fully RL-based agent.

4 Q-Learning Agent

In contrast to the PPO-based agent discussed above, another method we attempted was to significantly restrict the possible actions for the RL agent to a set of actions that are likely to be effective. This method was able to achieve more immediate results, which we describe in this section.

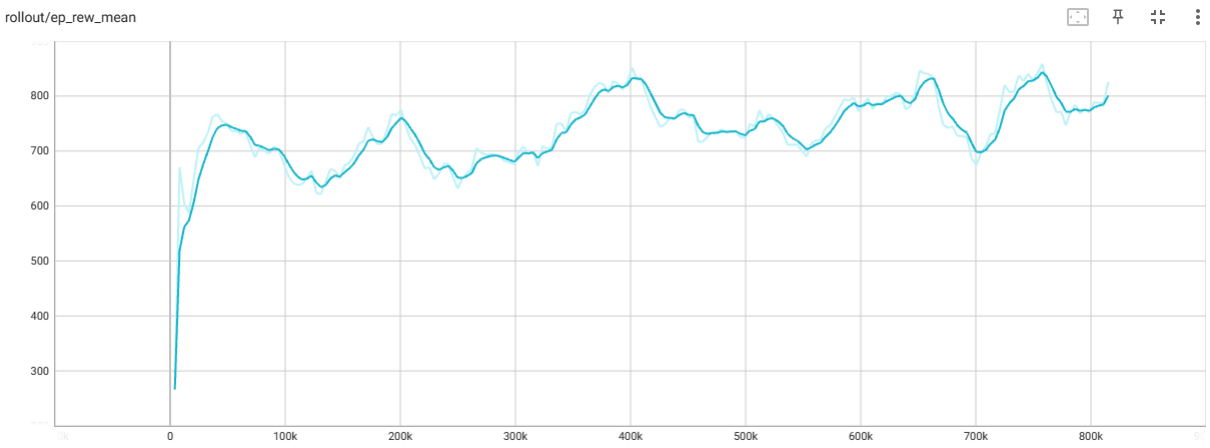

Figure 4: Mean reward per episode during training using QuantityBasedReward

```
Model PPO_L0_4-partners_20230803-145313_716800-steps_QuantityBasedReward scores:
[('SyncRandomOneShotAgent', 0.9671532567289708),
 ('SingleAgreementAspirationAgent', 0.8820922716538375),
 ('OneShotRLAgent', 0.7320648120499951),
 ('GreedyOneShotAgent', 0.6513504833039025)]
```

Figure 5: Average scores for ReducingNeedsReward in 50 world simulations

4.1 Method of Q-Learning

4.1.1 Simplifying the action space

In the SCM game, there is too much state information for Q-learning if everything is considered. Furthermore, the action space is massive and takes too much time in our training to fully explore state-action combinations. Thus, we implemented the following simplifications to the action space and the behaviors targeted for learning to try and improve agent performance:

1. Decrease the number of choices considered when making proposals by offering the same proposal to all partners

$$(2 \times 10)^{\text{number of partners}} \rightarrow 2 \times 10 \text{ (or less)}$$

2. Completely remove consideration of partner offers in the learning process by using a fixed acceptance strategy. Hence, we are only learning how to propose offers with this agent.

The first item above on its own is a significant reduction in the complexity of the problem being learned as the minimum number of negotiating partners an agent can have is 4, thus the reduction is from an action space with at least 16000 options to one with at most 20. When combined with the second item above, we were able to reduce the complexity of the problem to a more manageable point and saw significantly improved results over the PPO agents developed.

4.1.2 Acceptance strategy

The fixed acceptance strategy of our Q-learning agent is as follows:

1. If (our agent in level 0) or (our agent in level 1 and negotiation round ≥ 19):
   - Iterate over power set of all negotiating partner offers

- Record the gross income of all offer combinations whose total combined quantity does not exceed our current quantity needs
- Accept set of offers that has the highest gross income amongst these combinations that don't exceed our needs.

2. Else:
   - Iterate over power set of negotiating partner offers that are at our best price
   - Record the gross income of all offer combinations whose total combined quantity does not exceed our current quantity needs
   - Accept set of offers that has the highest gross income amongst these offer combinations at our best price, not exceeding our needs.

Our decision to have less greedy behavior when the agent is at level 0 came about from observing that the total exogenous quantity input in the SCM world is bounded from below by the total exogenous quantity output in the SCM world (often times being a strict difference rather than equal valued). We believe this puts level 0 agents in a somewhat more desperate position than level 1 agents, however we have not conducted any statistical tests to formalize/defend this claim.

4.1.3 State information

Our agent only needs to decide its proposed unit price and proposed quantity. Based on this, the information which we decided to include in the state is the current step, current needs, level, and current number of negotiating partners. Step and level are likely to be important in deciding the proposed unit price, and the current needs and current number of negotiating partners are likely to be important in deciding the proposed quantity. This reduced the number of possible states to 3360.

4.1.4 World and Model Settings of Q-Learning

- Each world configuration used in training was simulated for 50 days
- The agents trained against were the first, second, and third place finishers from the 2023 SCML OneShot competition (QuantityOrientedAgent, CCAgent, and KanbeAgent respectively).
- The actions taken by the agent during training were chosen randomly with a uniform distribution.
- The learning rate used was $\alpha = 0.2$
- The discount factor used was $\gamma = 0.999$

4.2 Analysis of `ByQValueAgent`

We compared the agent trained using Q-learning, `ByQValueAgent`, with the 2023 SCML OneShot competition first, second, and third place finishers. (n_steps = 50, n_configs = 10)

Table 1: analysis of tscores ByQValueAgent and 2023 Winners

agent name	mean	min	25%	median	75%	max
CCAgent	1.0863	1.0234	1.0739	1.0785	1.1154	1.1449
QuantityOrientedAgent	1.0854	1.0255	1.0661	1.0817	1.1131	1.1358
ByQValueAgent	1.0829	1.0206	1.0557	1.0723	1.0883	1.1778
KanbeAgent	1.0806	1.0147	1.0765	1.0785	1.0868	1.1333

Figure 6 and Table 1 show the results of testing, and we see that the `ByQValueAgent` attains 3rd place based on mean scores and 4th based on medians. One aspect of the learned behavior we thought may be causing poorer performance than desired is that the agent's actions are volatile in the sense that it sometimes chooses drastically different actions based on similar observations. In an attempt to address this, we made a new heuristic-based agent based on an analysis of the tendencies of `ByQValueAgent`.

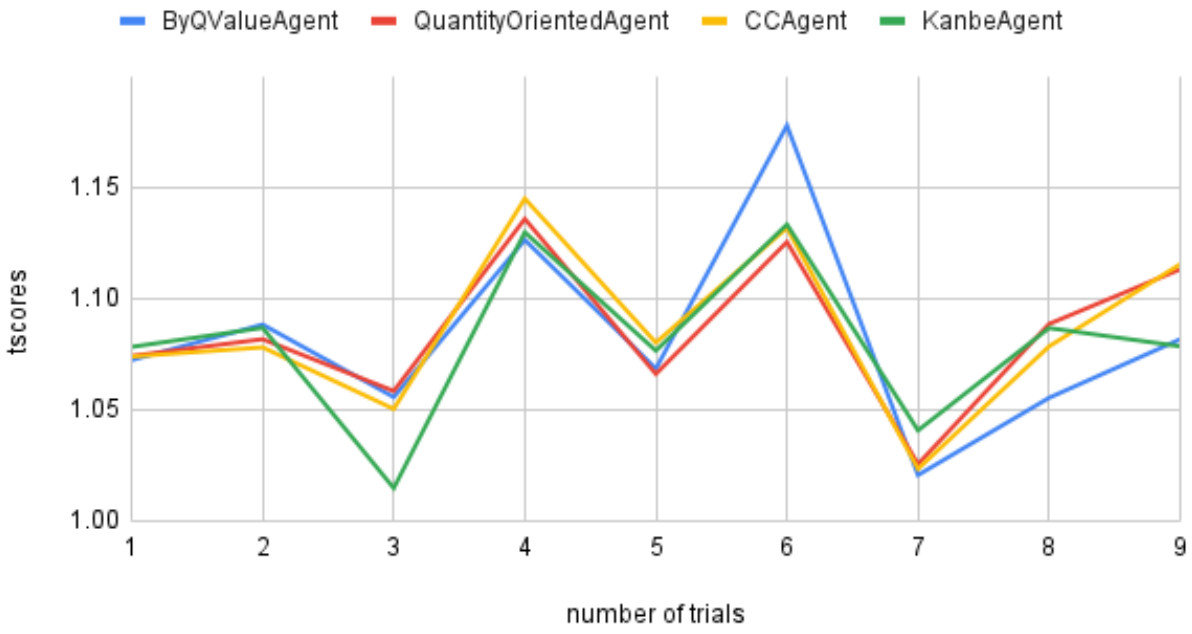

Figure 6: ByQValueAgent and 2023 Winners

4.2.1 Analysis of propose quantity

needs \ # current negotiation partners	1	2	3	4	5	6
1	1	1	1	1	1*	1*
2	1	2	2	1	1*	1*
3	2	2	2	2	2	2*
4	3	3	2	3	2	2*
5	3	3	3	3	3	3
6	4	4	4*	4	3	3
7	4	4	4*	4*	4	3
8	4*	4*	5*	5	4	4
9	6*	6*	6*	4	5	6
10	5*	5*	7*	7	6	6

Table 2: mean of best propose quantity by learning

needs \ # current negotiation partners	1	2	3	4	5	6
1	1	1	1	1	1*	1*
2	2	1	2	1	1*	1*
3	3	2	2	2	2	2*
4	4	3	2	3	2	2*
5	4	3	3	3	3	3
6	5	3	4*	4	3	4
7	6	4	4*	4*	3	3
8	7*	4*	4*	5	5	4
9	6*	6*	6*	4	4	6
10	4*	5*	7*	8	6	6

Table 3: median of best propose quantity by learning

Tables 2 and 3 show the mean and median suggested quantities based on `ByQValueAgent` for varying values of needed quantity and number of negotiation partners, where entries marked by * are ones where the Q-value is likely to be inaccurate because the scenario was not often seen during training. Based on these tables, we decided to use the following strategy for proposal quantity:

$$propose\ quantity = \begin{cases} needs & \text{(if number of partners= 1)} \\ \lceil needs/2 \rceil & \text{(elif needs} \leq 9\text{)} \\ \lceil needs/2 \rceil + 1 & \text{(else)} \end{cases}$$

Figure 7: propose unit price in level 0 by learning

Figure 8: propose unit price in level 1 by learning

4.2.2 Analysis of unit price

Figures 7 and 8 show the results of whether `ByQValueAgent` decides to propose using the best unit price or the worst unit price, based on the current negotiation round. These graphs indicate that proposing the best unit price is usually better than proposing the worst unit price, we don't have to compromise on unit price over time in either level.
So, we decide propose unit price as show below.

$$propose\ unit\ price = best\ unit\ price$$

It should be noted that the dominance of proposing our best price, like most aspects of strategy development in SCM, is highly dependent on opposing agent strategies. In particular, `ByQValueAgent` was trained against the top 3 finishers from the 2023 SCML OneShot competition, all of whom exhibited relatively generous acceptance policies. Specifically, two of these three agents completely neglected price when deciding whether to accept an opponent's offer or not, so it is not surprising whatsoever that `ByQValueAgent` learned that offering its best price was typically the best choice. This dominance in expected performance by offering your best price is not expected to persist in environments populated by greedier opposing strategies and for high-quality performance, we expect that the quality of offering at our best and worst price would need to be handled on an agent-by-agent basis. The most basic way of handling this would be to have some way of measuring the "greediness" amount in an opponent's strategy and classify them as either "greedy" or "not greedy" based on this, adjusting our strategy for the two different types. We did not have time to get to this at the end of this project and would be an interesting route for future methods.

4.3 Result of HeuristicAgent

4.3.1 Against 2023 winners

We compared new agent `HeuristicAgent`, which uses the proposal quantity and price strategies discussed in Sections 4.2.1 and 4.2.2, with the 2023 ANAC OneShot competition winners. (n_steps = 50, n_configs = 10)

Table 4: Analysis of HeuristicAgent and 2023 Winners

agent name	mean	min	25%	median	75%	max
HeuristicAgent	1.1087	1.0216	1.0813	1.1063	1.1354	1.1671
KanbeAgent	1.1005	1.1005	1.0860	1.0873	1.1248	1.1412
CCAgent	1.1098	1.0490	1.0716	1.0975	1.1234	1.1494
QuantityOrientedAgent	1.1097	1.0326	1.0841	1.0959	1.1191	1.1317

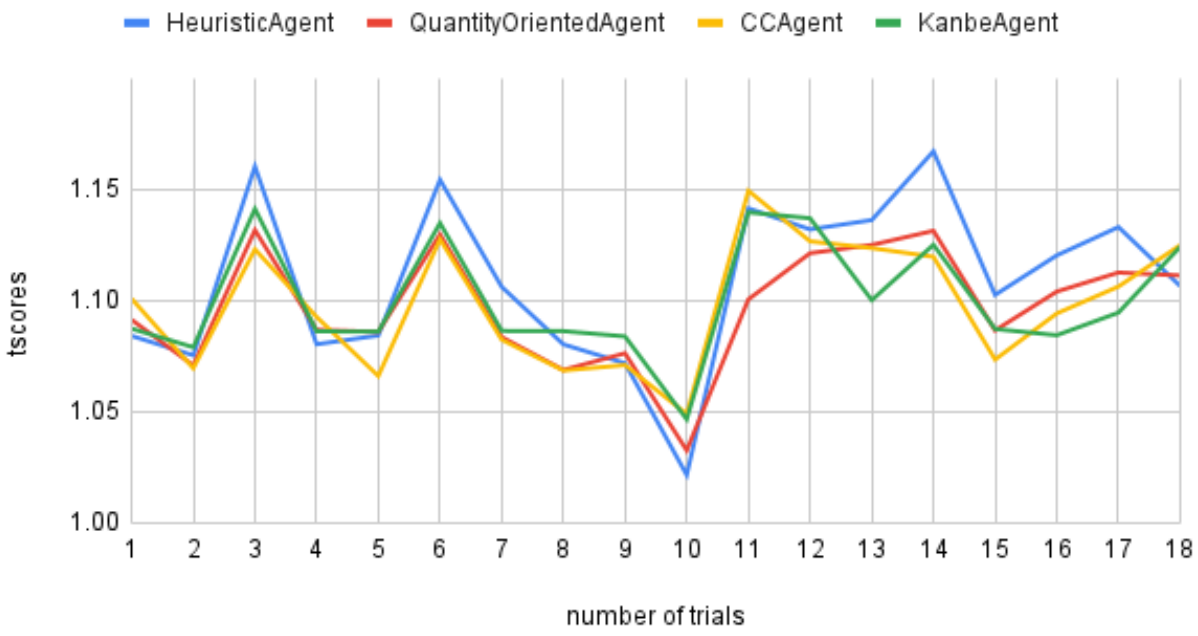

Figure 9: HeuristicAgent and 2023 Winners

The result was that our new agent `HeuristicAgent` was 1st place against the agents it was trained on by mean and median scores (but 4th place on minimum score).

4.4 Against other 2023 finalists

We also compared `HeuristicAgent` with other finalists from the 2023 competition which it was not trained against. (n_steps = 50, n_configs = 10). The results (Figure 10, Table 5) showed that `HeuristicAgent` conistently lost to a specific agent, `AgentVSCforOneshot`.

Table 5: analysis of tscores HeuristicAgent and 2023 other Finalist

agent name	mean	min	25%	median	75%	max
AgentVSCforOneShot	1.1186	1.0716	1.0990	1.1192	1.1337	1.1711
HeuristicAgent	1.1086	1.0340	1.0629	1.0862	1.1034	1.1648
Shochan	1.0776	0.9657	1.0458	1.0893	1.1166	1.1498
ForestAgent	1.0624	0.9022	1.0316	1.0791	1.1337	1.1545
PHLA	1.0369	0.8779	1.0158	1.0493	1.0881	1.1275
AgentSAS	1.0362	0.8786	1.0190	1.0506	1.0848	1.1240
NegoAgent	1.0283	0.9623	1.0134	1.0382	1.0505	1.0696

4.5 Considerations

To better understand why `HeuristicAgent` consistently lost to `AgentVSCforOneshot`, we examined the average needs at the end of each day for each agent (n_steps = 50, n_configs = 10, 2 try, with the 5 competitors).

The result is Figure 11 and 12. According to the result, the percentage of needs `HeuristicAgent` can fulfill is less than that of `AgentSCVforOneshot`, especially in level 1.

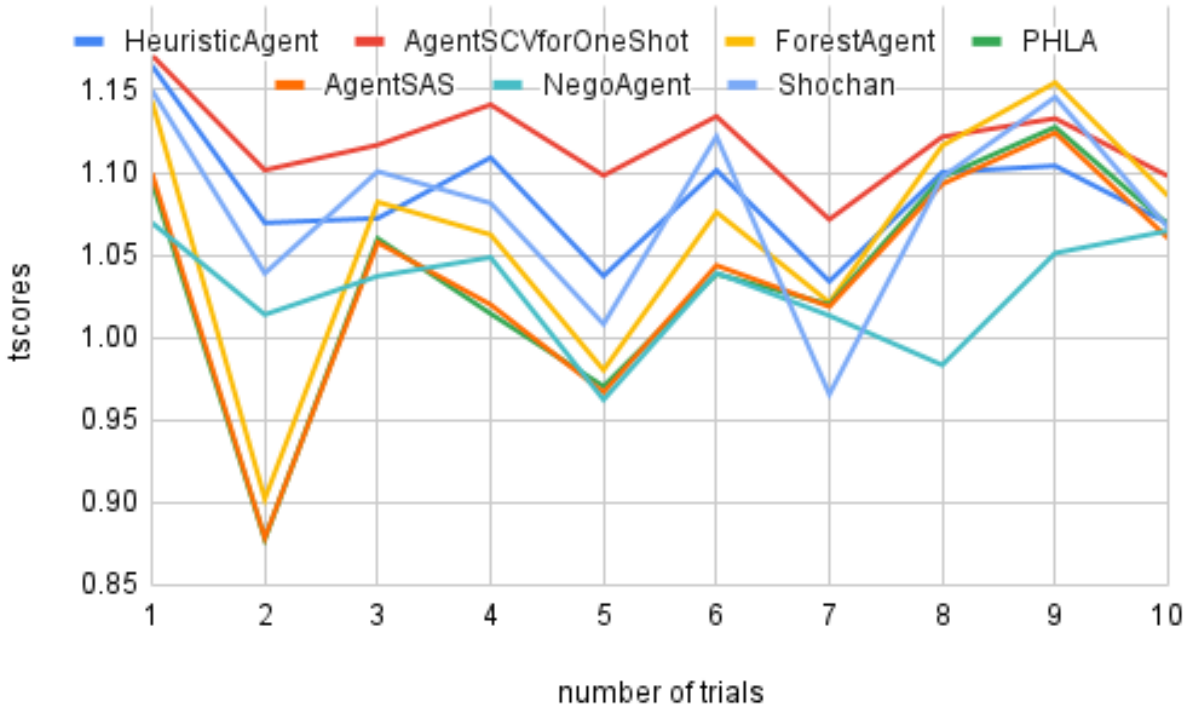

Figure 10: HeuristicAgent and 2023 other Finalist

4.5.1 New acceptance strategy

To try and improve quantity attainment, we adjusted our acceptance strategy to compromise unit price in level 1 more. This new agent `AdaptEnvironmentAgent` has two differences from `HeuristicAgent` in level 1.

- change the number of steps to completely compromise the unit price from 19 to 18
- compromise the unit price up to current_permit_needs per day even if the number of steps is less than 18.

At first, permit_needs is 0. And, we renew permit_needs as below every 10 days based on the average of remaining final needs for the last 10 days.

$$permit\_needs \to min\ \{permit\_needs + 1,\ 5\} \qquad (the\ average > 0.7)$$
$$permit\_needs \to max\ \{permit\_needs - 1,\ 0\} \qquad (the\ average < 0.3)$$

And, at the beginning of the day, current_permit_needs is permit_needs. If a contract is accepted at worst_unit_price, reduce the value of current_permit_needs by that quantity

4.6 Result of AdaptEnvironmentAgent

We conpared `HeuristicAgent` with `AgentSCVforOneShot` (n_steps = 100, n_configs = 10). And, we conpared `AdaptEnvironmentAgent` with `AgentSCVforOneShot` (n_steps = 100, n_configs = 10).

The results are Figure 13, 14 and Table 6, 7.

`AdaptEnvironmentAgent` now performs as well as `AgentSCVforOneShot`. However, depending on other competitors, it is still more likely that `AdaptEnvironmentAgent` will lose to `AgentSCVforOneShot` (as a side note, AdaptEnvironmentAgent still has good performance against the 2023 Winners it trained against). The results against `AgentSCVforOneShot` and the 2023 OneShot winners are presented in Figure 15 and Table 8 below.

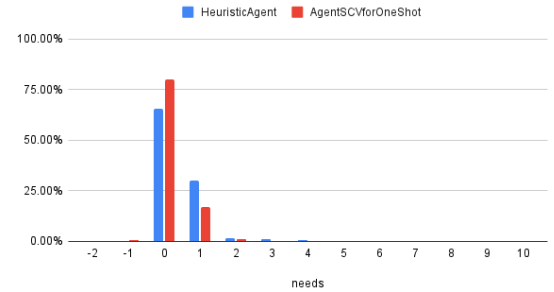

Figure 11: final needs of each days (level 0)

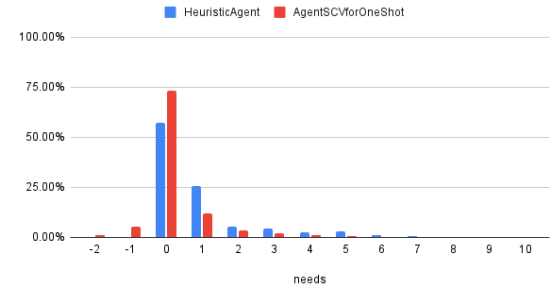

Figure 12: final needs of each days (level 1)

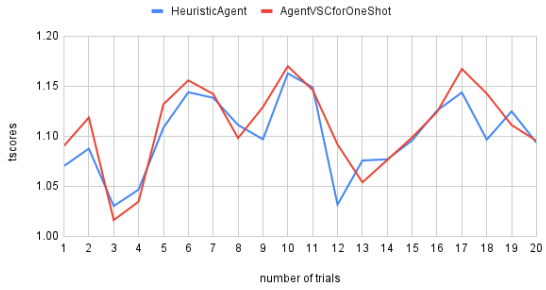

Figure 13: HeuristicAgent and AgentVSCforOneshot

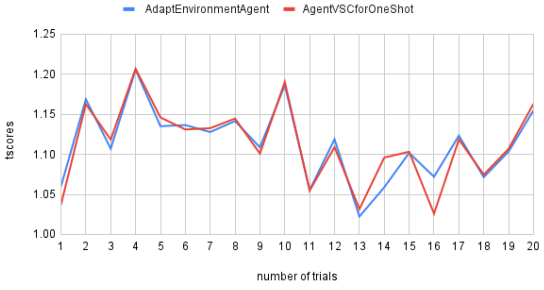

Figure 14: AdaptEnvironmentAgent and AgentVSCforOneshot

Table 6: analysis of tscores HeuristicAgent and AgentSCVforOneShot

agent name	mean	min	25%	median	75%	max
AgentVSCforOneShot	1.1097	1.0163	1.0913	1.1149	1.1423	1.1699
HeuristicAgent	1.1005	1.0303	1.0766	1.0967	1.1288	1.1627

Table 7: analysis of tscores AdaptEnvironmentAgent and AgentSCVforOneShot

agent name	mean	min	25%	median	75%	max
AdaptEnvironmentAgent	1.1129	1.0224	1.0718	1.1137	1.1377	1.2055
AgentVSCforOneShot	1.1127	1.026	1.0906	1.1136	1.1449	1.2068

4.7 Future work

We want to do the following three things:

1. Consider acceptance strategies about in level 1 that are more adaptable to the environment
2. Optimize each parameter
3. Use higher-powered hardware in order to evaluate and optimize each parameter (in real simulation max of n_steps = 5000, but our hardware can only handle up to n_steps = 200 at most in n_configs = 10 and number of competitor = 4)
4. Implement Q-learning in determining how to accept opponent offers.

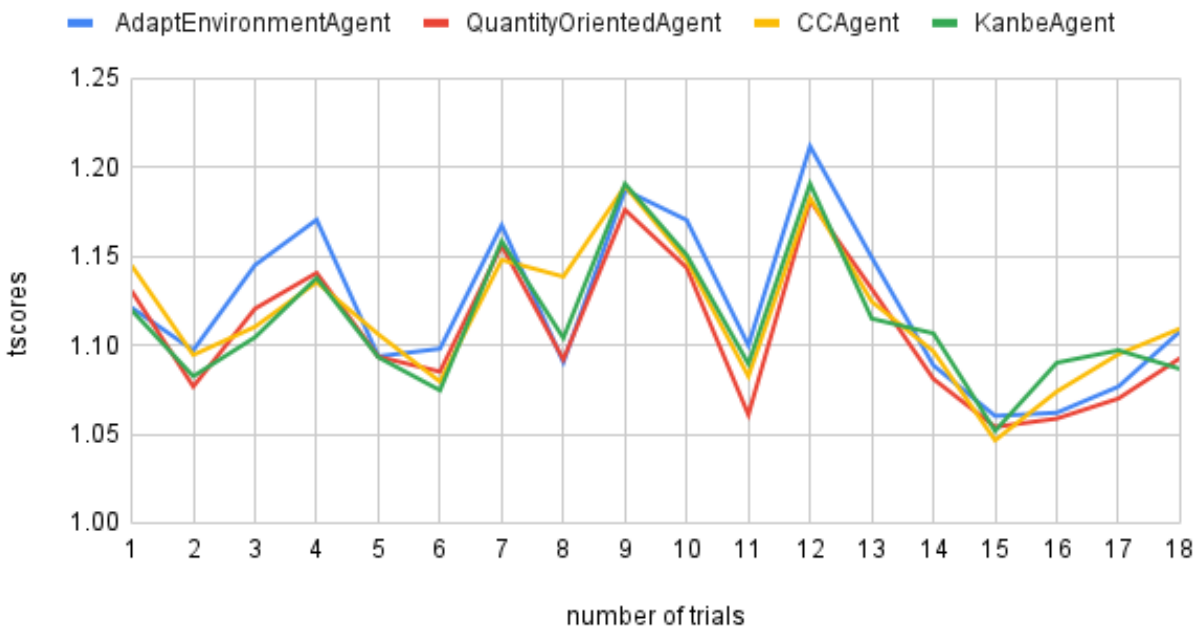

Figure 15: AdaptEnvironmentAgent and 2023 Winners

Table 8: analysis of tscores AdaptEnvironmentAgent and 2023 Winers

agent name	mean	min	25%	median	75%	max
AdaptEnvironmentAgent	1.1220	1.0602	1.0912	1.1038	1.1628	1.2118
CCAgent	1.1169	1.0465	1.0946	1.1098	1.1432	1.1894
KanbeAgent	1.1135	1.0518	1.0896	1.1042	1.1332	1.1907
QuantityOrientedAgent	1.1080	1.054	1.0777	1.0931	1.1383	1.1813

5 Conclusion

In our work, we developed several heuristic-based negotiating agents for the SCML OneShot competition, and also investigated two possibilities for RL-based agents. Our heuristic-based agents were primarily attempts to address the shortcomings of several previous competition winners, but failed to achieve consistently dominant results. In order to address this, we focused our attention on utilizing reinforcement learning. One of our RL approaches was to try to learn a strategy from scratch using the SCML RL interface. To this end, we have developed several custom observation managers, reward functions, and a custom world factory to train several RL agents. Initial testing indicates that our agents are still not performing well, but we have also developed a workflow for rapid iteration in the hopes of finding ways to optimize performance. Our second RL approach was to severely restrict the RL agent's action space to actions that have already been deemed to be good. This approach can potentially suffer from a lack of flexibility in its available actions; however, initial tests have already shown that it performs comparably or better than top-ranking agents from previous competitions.

References

[1] Yasser Mohammad, Enrique Areyan Viqueira, Nahum Alvarez Ayerza, Amy Greenwald, Shinji Nakadai, and Satoshi Morinaga. Supply chain management world. *In PRIMA 2019: Principles and practice of multi-agent systems*, pages 153–169, 2019.

[2] Yasser Mohammad, Shinji Nakadai, Satoshi Morinaga, and Katsuhide Fujita. *Supply Chain Management League(SCML) Automated Negotiating Agent Competition for Manufacturing Value Chain* , pages 351–359. Artificial Intelligence, 2020.

[3] Yasser Mohammad, Shinji Nakadai, and Amy Greenwald. Negmas: A platform for automated negotiations. In *PRIMA 2020: Principles and Practice of Multi-Agent Systems: 23rd International Conference, Nagoya, Japan, November 18–20, 2020, Proceedings 23*, pages 343–351. Springer, 2021.

[4] Ariel Rubinstein. Perfect equilibrium in a bargaining model. *Econometrica: Journal of the Econometric Society*, pages 97–109, 1982.

[5] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[6] Christpher John Cornish Hellaby Watkins. Leraning from delayed reward. *PhD thesis, Cambridge University, Cambridge, England.*, pages 220–228, 1989.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Anthony R. Cassandra Leslie Pack Kaelbling, Michael L. Littman. Planning and acting in partially observable stochastic domains. *Artificial Intelligence Volume 101, Issues 1-2*, pages 99–134, 1998.