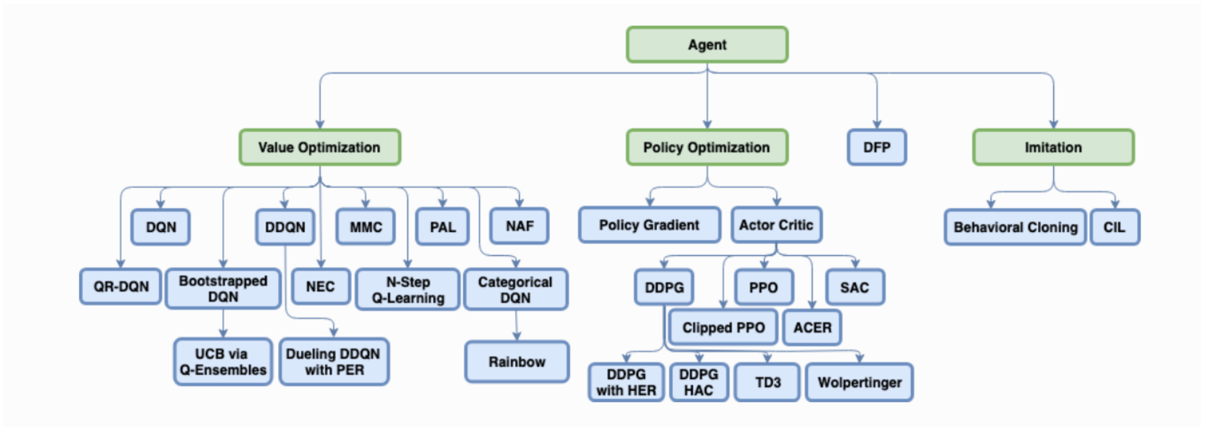
A compendium of thoughts:

Understanding continuous action space in Reinforcement Learning

Reinforcement learning (RL) methodology does not fit squarely into supervised or unsupervised learning as it won't deal with labeled or unlabeled data but instead interacts with an environment and learns best possible actions to maximize the outcomes (reward). If the agent interacts with the environment directly to collect the data, it will be processed in batches using replay buffers to update the policy; this is known as online RL. On the other hand, if data is collected by other means and has no direct interaction to the environment is known as offline RL. There are several applications to reinforcement learning from robotics, gaming, investment etc. One of the interesting applications of RL in generative AI is RHLF for finetuning large language models (LLM).

Some problems naturally lend themselves to RL settings like robotics and games where state and action space are discrete and finite. In many problems related to logistics and finance you will end up in a continuous action space which can be quite large. Another important factor to consider is the time horizon which can be very long or have no end. Think of an Online RL scenario where you need to collect and replay using a buffer which may not be possible. Defining the reward function also takes into consideration the time horizon. You can simply say your end state will have high reward but in all other states it is zero. This kind of approach will not be good for effective policy exploration. Another important factor is the strategy you employ in choosing an RL agent. If you do not choose the right agent, it may not converge. A comprehensive list of agents is shown below. There are several libraries which choose to implement all or part of them.

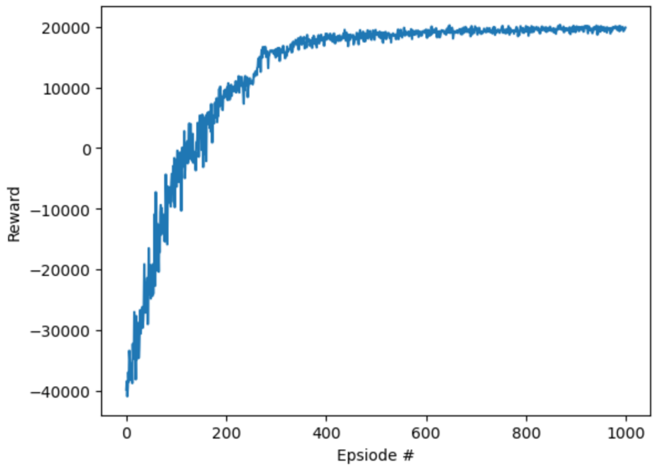


Now let us set up a scenario to understand empirically all the challenges addressed previously. We take a simple inventory example with a single product that has a fixed unit price with a certain inventory position. It takes a certain time for the ordered inventory to arrive. This item is on a fixed repeat order at price and variable cost of additional order. To control the capital used to buy the inventory there is a holding cost associated with it. Bottom line you can create an environment as complex or as simple depending on your requirement.

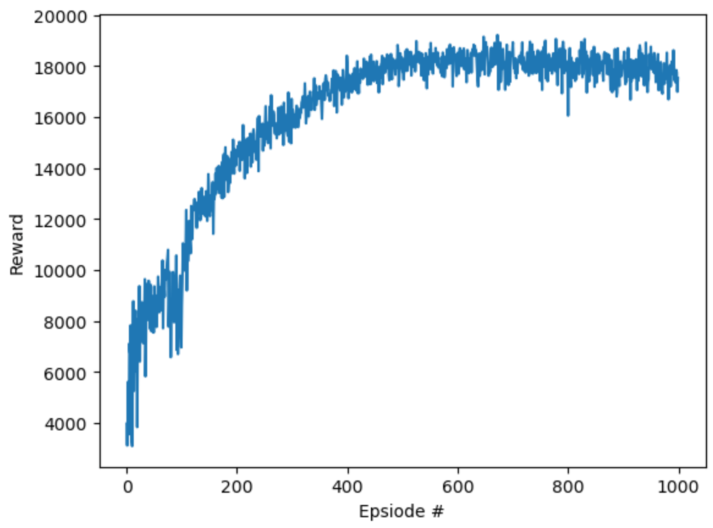
Given this, how do we define State, Action and Reward.
State – since there is a delay in fulfilling the order to have a considered inventory position over a period of week so we take the day of the week as state. So there 7 states
Action – Since we will have a limited budget to order let us say we fix these 20 units. So, any given day we can order new inventory from 0-20 i.e. 21 actions which is a continuous action space.
Reward – We can use profit as proxy for reward as units_sold*unit_priceholding_cost*inv_level - y*fixed_order_cost -action*variable_order_cost

Refer to the inventory environment class in invenv.py in the repo RLcontinousAction.
 We create a hypothetical historical data of inventory with a specific pattern Mon – Thu a mean demand of 3 with SD of 1.5. Friday with a mean of 6 and SD of 1 and weekend with a mean of 12 and SD of 2. If you want more realistic demand generation, you can use a Poisson distribution. The demand creation is done for 52 weeks in creaternddemand.py.

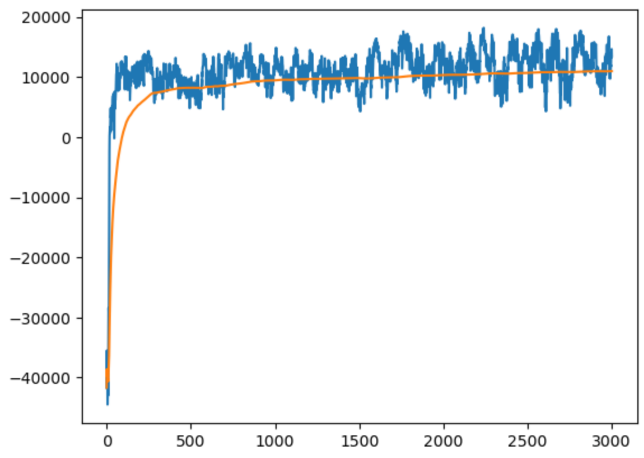
Since this is a value optimization, we choose with DQN. The network is defined with 2 hidden layers of 128 dimensions in modelq.py. DQN agent uses this network with a batch size of 128 and learning rate of 1e-4. The RL params Gamma the discount factor as 0.99 and Tau the soft update of the target network is set at 1e-3. DQN is defined in q_agent.py. We learn Q values for 1000 episodes in qRL.py. As you, the results below reward converging very smoothly.



In the initial setup of the environment, we assumed that we can only order max of 20 units hence we have 21 actions. What if we had a maximum order capacity of 500 then the action space is 501. This will quickly go out of control. Here there are few choices we can make. We can study historical inventory and sub-sample action space. One downside to the approach is we are assuming that historical actions are good enough for optimal value. Another approach is to discretize action space into manageable chunks. Let us take the previous example and reduce the action space to even units ordered. Refer to qRLStep.py for details. As you see this also produces comparable results as shown below. You can also think of other sophisticated heuristics like random walks and Bayesian optimization rather than simple discretization.



If we consider policy optimization agent say PPO instead of value optimization for this inventory problem, there will be convergence issues as well as the total reward is not comparable to DQN as shown below. Readers can refer to acPPOinv.py in the repo RLcontinousAction and tweak params to see how close they can get to DQN.



In industry these kinds of inventory or financial problems with continuous variables problems are defined as constraint problems with either Integer or Mixed integer programing. There are many commercial and open source software available, like CPLEX, GUOBI, Matlab, Pyomo, OR-Tools for solving constraint problems. But in many cases the constraint space definition becomes too complex and different heuristics need to apply to get to optimal solution. In many cases we want to get to the approximate solution or there may be heuristics we have not thought of, in such scenarios RL plays a crucial role. The RL frameworks have evolved to handle complex problems with distributed compute. rllib and Stable Baselines are well established mature distributed RL frameworks allowing it to scale. Non-RL frameworks are in batch mode but with ever increasing online retail with thousands of SKU we need real time solutions, RL play a pivotal role in addressing these kinds of problems.