Replenishment of Retail Stores
TCS Quantum Computing Challenge

Phase 2 Submission

Affiliation: Bloq

Nikhil Londhe, Grishma Prasad, Jay Patel, Pratik Kumar De, and LS Sreekuttan

March 8, 2024

Contents

1 Introduction

For a large retailer, replenishing the store estate with the right products at the right time requires striking the right balance between stock availability and supply chain cost. The challenge has multiple dimensions, including the rate of sale, range, delivery frequency, pack sizes, sales forecast accuracy, seasonality, distribution center capacity and throughput limitations, physical variations between stores, and store access limitations.

Currently, for a retailer with over a thousand stores and tens of thousands of different SKUs, granularity is traded off against speed of calculation, and hence, stores and SKUs are grouped. This results in a sub-optimal solution. There is immense potential to identify quantum-based solution approaches to address the need of SKU based service levels with the lowest possible working capital and cost to serve.

2 Methodology

Our challenge is using Quantum computing methods to tackle the problem of optimally replenishing retail stores. This problem has two levels of optimization. Firstly, we need to optimize at the store level by finding the optimal replenishment quantity the store will order from the distribution center (DC) to meet the demand while maximizing profit. This involves considering factors such as the stock at the store, the inventory holding cost, and the forecasted demand. Secondly, we need to optimize the DC level, where the DC receives replenishment requests from all the stores and decides how much it will replenish each store based on its stock, lead time for stores, and the expected profit. We focus on store-level optimization and have developed strategies that we discuss below.

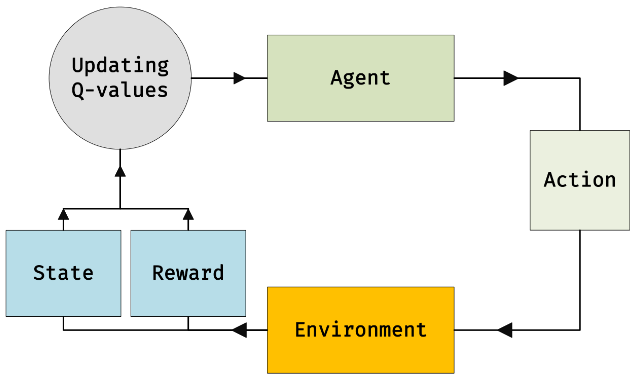

Figure 2.1: Flowchart for Q-Learning.

2.1 Quantum Reinforcement Learning

Markov Decision Process, or MDP in short, is a tool for studying and solving optimization problems. MDP is characterized by states, actions, transition probabilities, and rewards, with the Markov property ensuring that future states depend only on the current state and action. The goal is to find an optimal policy that guides the agent to take action, maximizing the expected cumulative reward. Previous attempts have dealt with replenishment problems using MDP[1][5]. The common approach is creating a state of store stock and demand. We will create states similarly but add available capital as an extra dimension. We define state $\mathcal{S}$ as,

$$\mathcal{S} = (FD, C, OH)$$

where OH is the on-hand inventory, FD is the forecasted demand, C is the available capital. OH is matrix of shape $N_p \times N_s$ where N_p is number of products and N_s is the number of stores. OH_{ij} means on-hand stock for ith product in the jth store. Action is the weekly replenishment we will make, denoted by X. Our decision variable is the replenishment decision. OH, X will be a matrix with the shape $N_p \times N_s$. The reward for a particular action will be calculated based on the sales that occur that week and will use the forecasted demand. We use a probabilistic model of sales 5.2, which is bound by a lower bound (0) and a higher bound (forecasted demand). We then sample from this distribution to get the predicted sales. The actual sale will depend on the sum of store stock (on hand) plus the order that the store places (replenishment).

An important aspect of MDP is the state space. In our case, it is all the possible replenishments we can make. Figuring out all possible actions (replenishment) requires determining the total number of partitions of all possible products we can buy at any point in time. Partition theory deals with these types of problems, and there is no analytical formula to calculate the number of partitions for any given number; rather, we have to count it manually. Due to this, at any given state, it is generally difficult to know all possible actions. So, we devised our solution to have an MDP for every combination of products and stores. Therefore, our new MDP state s_{ij} will be,

$$s_{ij} = (FD_{ij}, C, OH_{ij})$$

where, definitions as same as before but OH_{ij} and FD_{ij} are scalars now instead of multidimensional matrices. We will classify product/store combinations into high, medium, and low profitability classes. So, we will run MDP for combinations with high first, medium second, and low third. This will ensure that we prioritize products and stores that are highly profitable.

Reinforcement Learning

We use Quantum Reinforcement Learning (QRL) to tackle the above-mentioned MDP. Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment. The agent receives feedback through rewards or punishments based on its actions, guiding it towards learning optimal strategies. RL involves exploring the environment to discover the most rewarding actions while exploiting known strategies. Key components include states, actions, rewards, policies, and value functions[3].

QRL[2] is similar to classical RL, wherein you have a quantum state that acts as an agent to interact with the environment. The agent interacts with the environment via actions and gauges them by rewards. A series of actions form a policy, and we want to find one that maximizes the cumulative reward. Popular algorithms like

Q-learning and deep reinforcement learning methods, such as Deep Q Networks (DQN), have been successful in applications ranging from game playing to robotics and autonomous systems. Figure 2.1 is a diagrammatic explanation for a Q-learning based algorithm. Q-learning is a model-free reinforcement learning algorithm wherein the agent learns the value of actions over all the possible states.

$$P = min(S, OH + X) \times (P_c + P_r) - X \times (P_c + P_h) - OH \times P_h - max(X - D, 0) \times P_h \tag{1}$$

Where,

- P: Weekly profit.
- D: Demand quantity.
- S: Predicted Sales.
- OH: On-Hand quantity.
- X: Replenishment.
- P_r: Profit per unit for product.
- P_c: Procurement cost per unit for product.
- P_h: Holding cost per unit per week for product.

1 is the reward function we designed and implemented using an Open AI gym environment.

3 Results

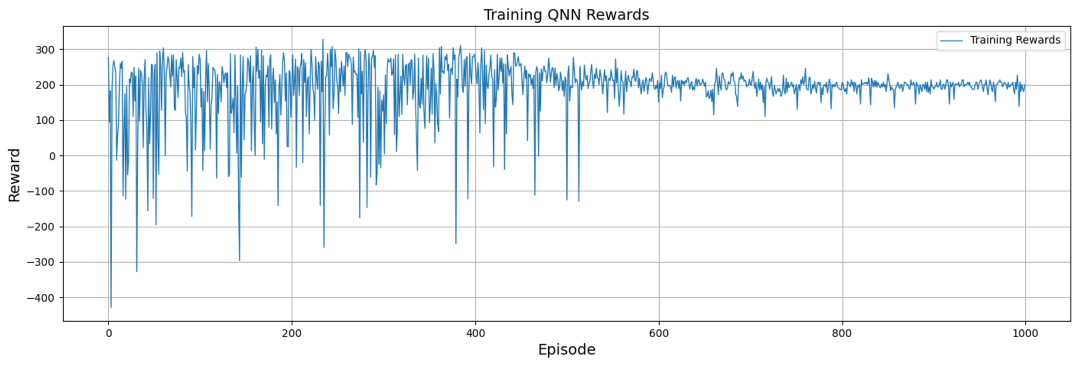

Figure 3.1: Training QRL on an ideal simulator (AerSimulator) for 1000 episodes.

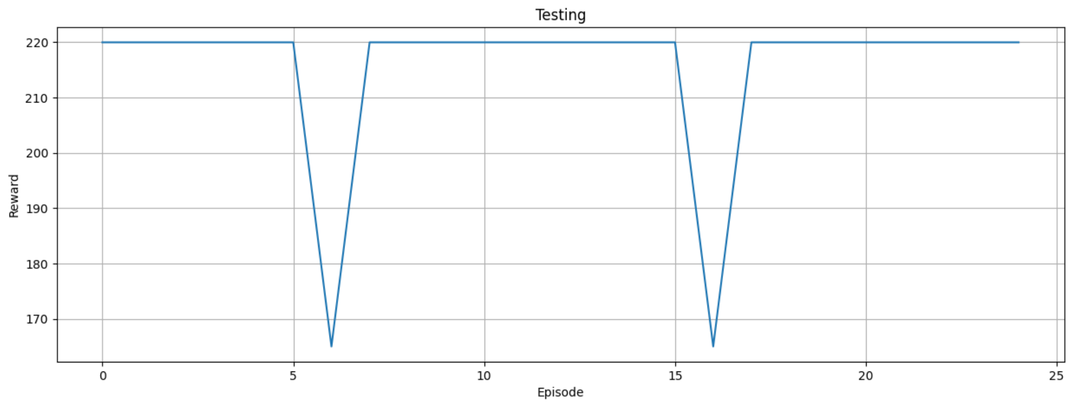

Figure 3.2: Testing the trained QRL on an ideal simulator (AerSimulator) for 25 episodes.

We have run our quantum model on Aer Simulator and IonQ Aria 1. We have also compared the quantum model with the classical model of reinforcement learning, in which the agent is a neural network. In Figure 3.1 and Figure 3.2, we can see the performance of the quantum agent using the Q-learning algorithm in testing and

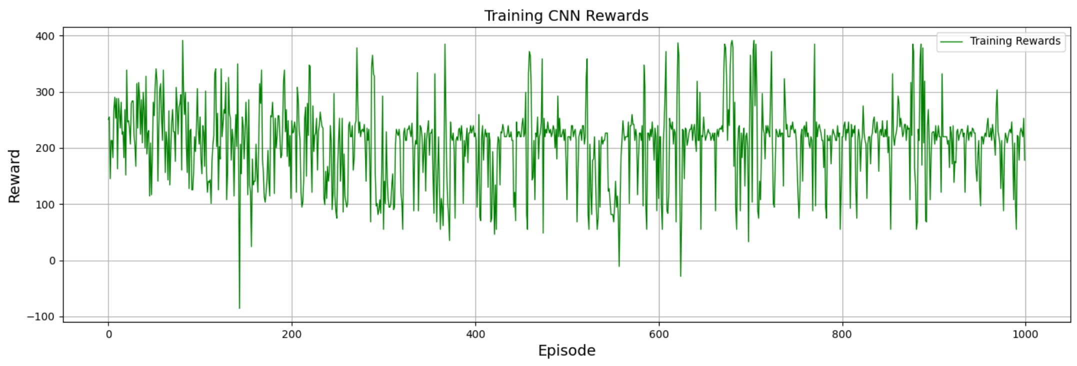

Figure 3.3: CRL model training for 1000 episodes.

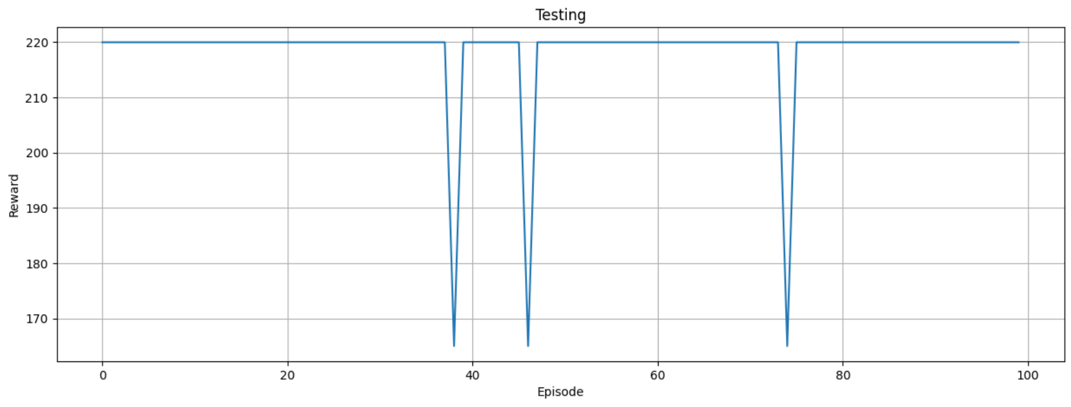

Figure 3.4: Testing the trained CRL model for 1000 episodes.

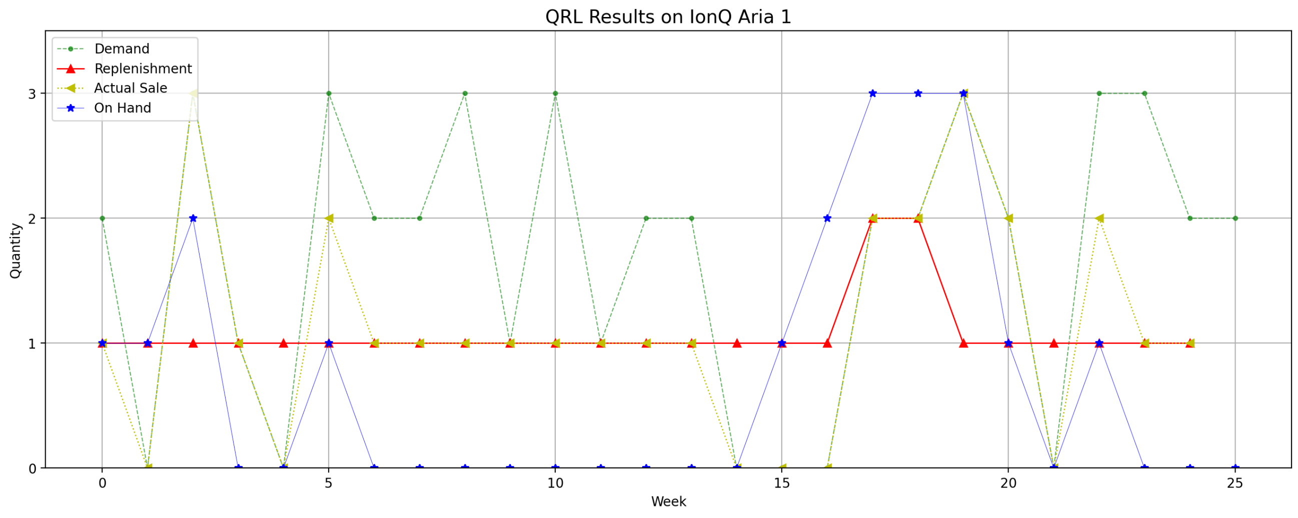

Figure 3.5: Decisions QRL model, which was run on the IonQ QPU Aria 1, took over 25 weeks.

Model	Best Training Reward	Testing Reward
QRL (AerSimulator)	327	218 (Mean - 25 episodes)
QRL (IonQ Aria 1)	-	206.8
CRL	391	219 (Mean - 100 episodes)

Table 1: Rewards for the graphs above.

Model	Time taken	Episodes
QRL (AerSimulator)	$\approx$ 35 mins	1000
QRL (IonQ Aria 1)	$\approx$ 3 hrs	1
CRL	$\approx$ 4 mins	1000

Table 2: Timing Models.

training. In 2, we have also mentioned the time it takes to run the code/circuits on CPU/QPU, and in 1, we have presented the rewards we get for all the models.

All the graphs and rewards show that QRL with PQC performs similarly, if not better, than the CRL model. However, the QRL model's runtime is significantly higher than the CRL. This is mostly due to the training algorithm we use for the QRL model. The training algorithm for the QRL model involves learning from experiences of a certain batch size. This results in the number of model evaluations (circuit simulations in reference to QRL) being significantly more than that of the CRL model. The following calculation calculates the upper bound of the required circuit evaluations. Number of episodes $= N_e$, No. of training episodes $= N_t$, and Batch size $= b$. Then, the total number of circuit simulations $\leq (N_e + bN_t) \times 25$. We have also presented the decisions taken by the QRL model 3.5, which was run on the IonQ QPU.

4 Conclusion

To conclude, our team has developed a quantum-based approach using Quantum Reinforcement Learning (QRL) to address the problem of optimally replenishing retail stores. We have formulated the problem into a Markov Decision Process (MDP) framework, where the agent learns to make replenishment decisions based on the current state of each store, including factors such as on-hand inventory, demand forecast, and available capital. Our approach leverages the Q-learning algorithm to train quantum agents interacting with the environment to maximize cumulative rewards. In this case, this translates to maximizing profit while minimizing costs associated with inventory holding and procurement. Our quantum-based RL technique has performed at par with the classical RL method. This result was replicated in the simulator and also real quantum hardware. We used IonQ Aria for our experimentation. The promising observation of our result is that our QRL approach matched the CRL results with a fraction of the parameters. The CRL model required 133 times the parameters of our QRL model to give a similar outcome (QRL 6 parameters, CRL 800 parameters)

Scaling this problem to multiple stores is straightforward as we use a one-to-one mapping of state variables and qubits. However, the challenge lies in creating unique observables representing actions or replenishment in our scenario. Replenishment is constrained by available capital so choices may vary each cycle. Yet, we can't generate observables indefinitely as Pauli strings are limited to the n-qubit Pauli group. We must solve this limitation to scale this solution to a practical scenario efficiently. This is a hopeful step toward quantum computing providing a positive ROI. As we can see, our quantum approach requires fewer parameters to learn and thus requires smaller computing power. The current ML progress is limited by computing power; a fault-tolerant QC will enable us to train huge models cost-effectively, saving enterprises millions of dollars. Once the quantum hardware becomes more affordable, our results prove a strong alternative to classical methods and, hopefully, positive ROI.

5 Appendix

5.1 Parametric Quantum Circuit (PQC)

The Parametric Quantum Circuit used as part of the Quantum Reinforcement Learning algorithm is shown in Figure 5.1. It contains multiple Encoding, Rotational, and Entangling layers. In the example, it encodes three data points, while the number of parameters (weights) is 36, where every layer has six parameters.

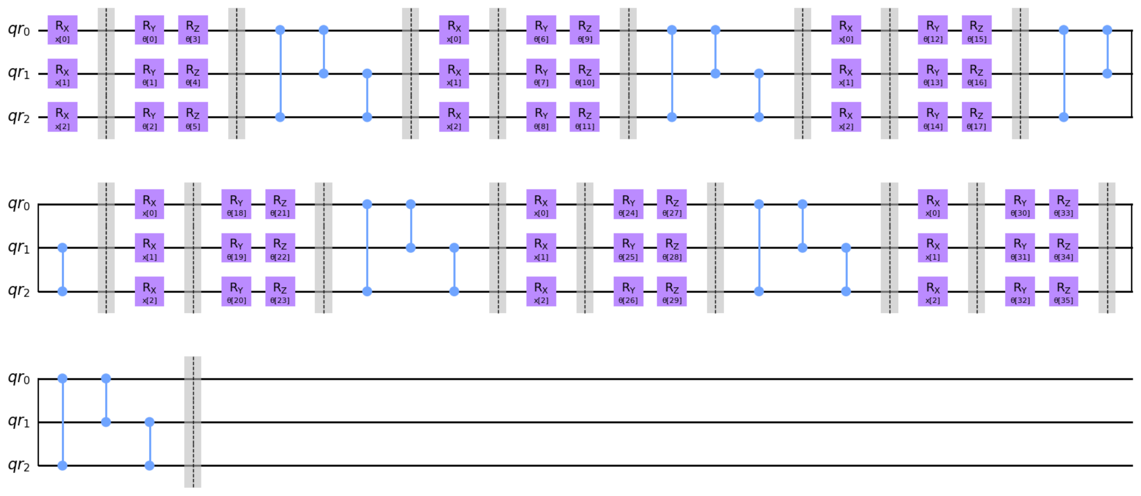

Figure 5.1: PQC

5.2 Probabilistic model of Sales

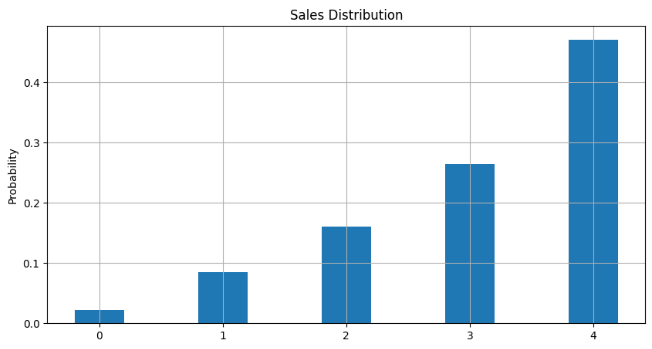

Figure 5.2: Example of a probability distribution for sales.

We model sales as a random variable that is sampled from a probability distribution, an example of which is given in figure 5.2. The figure shows five values ranging from zero to four, but in all our testing, we have four values ranging from zero to three.

5.3 Reinforcement Learning and Q-Learning

5.3.1 Classical Neural Network (Deep Q-learning)

This section describes some of the technical and business intuition behind our classical approach (detailed business-related conclusions can be seen in the Conclusions section of this report). We benchmark our quantum approach against this classical approach.

Training an RL model involves specifying our business decision criteria/variables (RL terminology for this is observation space). Based on the decision criteria, we would be taking an action. We can take a finite number of decisions/actions. We also need to specify all these possible actions under any given values the decision variables take (RL terminology is action space). Since our motto is to maximize profit as much as possible, we need to specify the logic behind how we will take action based on the rewards. All this is typically provided in a customized RL environment. We also added configurable product-specific details such as profit per item, holding cost per item per week, and cost per unit of the item in our environment. The business team could pass in a product ID and store ID, and these details can automatically be picked up from the dataset with these details.

1. Available capital: We would like to minimize our capital expense due to on-hold, and we also want to avoid excessively ordering items in the hope of excess demand for the item. This could be tracked with the available capital. If a weekly cost exceeds available capital, we terminate the entire episode and move on to the next episode, where all the decision variables are reset to some fixed initial values. We take an initial capital of 1000 and bound the capital between 0 and 10^7. The choice for the initial capital and the upper bound of the capital is not driven by business intuition but is easily configurable by the business team.

2. On hand: This variable reduces on-hold cost and prevents excess ordering of the item from DC when it's already in stock. This takes integer values between 0 and 100. Though the data does not indicate number of on-hold items as high as 100, we thought it better to choose a higher upper bound so that the model learns to avoid hoarding many items when the demands are clearly not that high.

3. Demand: This is an important variable to avoid less ordering or excess ordering of items from the DC. We have considered this an integer that varies between 0 and 3. The sales data, which comprises sales for several product/store combinations from 2020 to the first four months of 2023, indicates that there are only 15 weeks when sales for some product and store combinations were four units and exactly one week for a product and store when sales were five units. So, from a business point of view, it felt accurate to limit the anticipated demand to 3. Considering all items and store combinations, we noted that the maximum historical replenishment is 4. Still, we also noted that there are barely seven weeks when the replenishment in the past was 4 (considering all product/store combinations and out of several weeks from 2022 and some weeks in 2023). We randomly draw demand values for every step so that the model can learn to make decisions accurately even when no pattern is visible in the actual demands.

The sale considered in every step is the minimum of a sampled sale from a probability distribution (details in the Quantum Reinforcement learning section) and the max possible sales (sum of replenishment and on-hold). This sale and the other product details mentioned above, provide us the reward or net profit for a given item and store combination. One of the key outputs from every step in the environment is the next state or the set of values of the decision variables to carry out the subsequent step. Once the environment is ready, our main classical approach is deep Q-learning. This involves a standard neural network. We took two dense layers of 24 nodes, each with an activation function as a rectified linear unit or ReLU. The output layer is also dense, with as many nodes as possible actions (4 in our case) and linear activation. We decided to use this simple classical DQN model to avoid over-fitting.

Since this is an RL approach, exploration probability is an important factor to consider. In the initial steps of the training, the exploration probability should be high so that the model effectively learns about good and bad decisions based on different values of decision variables. As the model starts learning, the exploration probability should decrease so that the model doesn't have the scope to make arbitrary decisions and eventually converge. We took the starting exploration probability as 0.9, and our lower bound for exploration probability is 0.01. We reduce the exploration probability by 0.5 percent of its previous value with every step. This is a slow reduction rate, so the model has ample time to learn effectively. In every step, we choose a random number between 0 and 1. If that number is less than the exploration probability for that step, we take random actions. The logic here is that the higher the exploration probability, the higher the chances for a random number generator to pick a number less than the exploration probability and greater than 0. So, this serves our purpose because we want more random decisions with higher exploration probability. If the chosen random number is greater than the exploration probability, we choose the action as the node index corresponding to the max value we get as the output of our neural network.

Our other choices for learning rate (0.001) and optimizer (Adam) are based on choices that are known to give good results. The loss function we consider is the standard function for Q-learning that takes the square of the difference between the current/immediate reward and the target reward (taken as the discounted reward from the next step we get from our environment). The discount factor chosen by us is 0.99.

In every iteration (RL terminology: episode) of our training, we take 25 steps/weeks into account. This is because, based on the data we have, the forecasted demand given to us is for 25 weeks. We were thinking of alternatives, but from a business point of view, currently, we think taking a shorter forecasting horizon (not more than 25 steps) at the beginning is a better choice so that as the approach is tested in real-time, changes can be made to the decision variables and the logic.

5.3.2 Quantum Reinforcement Learning

We use Quantum Reinforcement Learning to solve the replenishment problem. Specifically, we use a quantum agent to make decisions (replenishment) for us. We have borrowed the idea of using quantum agents in the context of Q-learning from [4]. The quantum agent is a PQC from which we measure some observables. These observables represent the action. Since we chose four actions, we designed four observables to represent these actions. The observables are measured by calculating the expectation of Pauli strings. Since we have a three-qubit circuit, we can only create Pauli strings of length three. We choose Z_0, Z_1, Z_2, and Z_{012} as the Pauli strings as our observables. So, the quantum circuit will take in the state containing three variables and an array of four expectation values representing the action. The rest of the procedure is similar to the one described in 5.3.1.

References

[1] J. Goedhart, R. Haijema, R. Akkerman, and S. de Leeuw. Replenishment and fulfilment decisions for stores in an omni-channel retail network. *European Journal of Operational Research*, 311(3):1009–1022, 2023.

[2] W. Hu and J. Hu. Distributional reinforcement learning with quantum neural networks. *Intelligent Control and Automation*, 2019.

[3] R. Karthikeyan. *Quantum inspired algorithms for learning and control of stochastic systems.* PhD thesis, 2015.

[4] A. Skolik, S. Jerbi, and V. Dunjko. Quantum agents in the Gym: a variational quantum algorithm for deep Q-learning. *Quantum*, 6:720, May 2022.

[5] A. G. Zheng Sui and L. Lin. A reinforcement learning approach for inventory replenishment in vendor-managed inventory systems with consignment inventory. *Engineering Management Journal*, 22(4):44–53, 2010.