Dynamic Pricing in E-commerce Using Multi-Agent Systems

Abstract—
Deep Qlearning, a kind of Reinforcement Learning, is used in this study to improve dynamic pricing in a simulated multi-vendor marketplace. Each vendor, portrayed as an agent, employs a Deep Q-Network to establish pricing strategies depending on current market conditions, as well as a profit-maximizing incentive system. The simulation shows that the agents effectively develop effective pricing techniques over time, resulting in increasing incentives or profits. This research highlights the potential of Reinforcement Learning for addressing complicated optimisation problems in dynamic contexts such as markets, with important implications for dynamic pricing systems, competition among vendors, different base demands and e-commerce.

I. INTRODUCTION

In this paper, a multi-agent reinforcement learning (MARL) simulation in a marketplace scenario is provided, where sellers alter product pricing dynamically based on market condition and individual profit expectations. The paper includes Python code that mimics the interaction of numerous sellers in a marketplace selling a set of items. The Deep Q-Network (DQN), Replay Buffer, VendorAgent, Product, and Marketplace classes are the core code components, each meant to satisfy various parts of the simulation.

The DQN class offers the neural network architecture, which facilitates learning from marketplace simulation conditions, actions, and rewards. The Replay Buffer class is used to save and retrieve experiences from which the DQN may learn [1].

The VendorAgent class represents a single marketplace merchant.

It comprises techniques for selecting actions based on the current state, adjusting product pricing depending on selected actions, learning from experiences, and updating the Q-knowledge. network's

The Product class incorporates the characteristics of a single product in the marketplace. This class is used to characterize a product's cost of goods sold (COGS), current price, and market demand.

Lastly, the Marketplace class depicts the global environment in which all sellers interact with one another. It simulates the marketplace's functioning over time, calculates incentives for merchants' activities, and updates the market's overall status. The main simulation loop is divided into episodes, each simulating a day in the marketplace. The simulation runs through numerous time steps within each episode, when sellers decide on pricing modifications for their items based on their existing information.

This technique is based on the ideas of reinforcement learning, a branch of machine learning that is particularly suited to situations involving sequential decision making and long-term goals, such as pricing optimization [2].

II. RELATED WORK

In recent years, dynamic pricing in e-commerce has been a prominent area of study, with the addition of multi-agent systems adding another degree of complexity and optimization. Much study has been conducted on dynamic pricing techniques in the context of e-commerce. In online marketplaces, dynamic pricing models have been used to optimize sales, maximize profitability, and increase consumer happiness. Conventional pricing techniques are frequently demand-driven or competition-driven (Chen et al., 2017) [3].

Recent research has concentrated on the use of multi-agent systems to dynamic pricing in e-commerce. For example, He et al. (2019) proposed an intelligent multi-agent system for dynamic pricing in which agents may negotiate prices autonomously based on market conditions [4]. Zhang et al. (2020) suggested a dynamic pricing model for e-commerce platforms based on reinforcement learning, in which agents learn and react to market changes [5]. Moreover, Vytelingum et al. (2008) introduced a multi-agent system algorithmic framework for dynamic pricing in e-commerce. The suggested paradigm enables agents to not only adjust to market changes, but also to consider rivals' strategies [6].

Walsh et al. (2010) used multi-agent reinforcement learning for dynamic pricing in e-commerce in another investigation. Their methodology teaches agents how to develop effective price strategies based on both consumer purchasing habits and competitive pricing strategies [7].

While these studies present novel ways for dynamic pricing in e-commerce utilizing multi-agent systems, there is still need for more study to improve market performance and consumer happiness.

III. METHODOLOGIES

Seed Initialization: A seed is set for reproducibility. This allows the same results to be produced whenever the code is run.

Neural Network Architecture:

The Deep Q-Network (DQN) employed in this application is a feedforward artificial neural network variation. In a reinforcement learning system, the DQN is intended to approach the optimum Q-function. The Q-function, also known as the action-value function, represents the predicted future rewards for an agent starting in a certain condition and performing a specific action. DQN's purpose is to train a policy that maximises overall reward. The DQN design is made up of three fully interconnected levels. The first two layers are activated with ReLU (Rectified Linear Unit), which brings non-linearity into the network. The network may produce a continuous range of values reflecting Q-values since the output layer does not require an activation function. In this situation, the output layer contains the same number of neurons as the number of potential actions, therefore the Q-values for all actions are evaluated concurrently. The addition of "experience replay," in which previous state-action-reward transitions are saved and utilised to train the network, is a distinguishing characteristic of DQN. This improves training convergence by decreasing correlations in the sequence of observations. The experience replay technique also assures that every previous transition can possibly be employed in a

large number of weight changes, resulting in more efficient learning. Lastly, to stabilise learning, the DQN employs a separate target network. After a given number of steps, the weights of the target network are adjusted to match those of the main network, minimizing the variance of the updates. In 2015, Google DeepMind researchers presented the Deep Q-Network algorithm. It was able to play a number of Atari 2600 games at human-level competence, demonstrating a breakthrough in reinforcement learning.

The code implements a Deep Q-Network (DQN) architecture using PyTorch. The DQN consists of three fully connected layers. The input layer takes the state space dimensionality as the input. The first hidden layer has 64 units, followed by a second hidden layer with 128 units. The output layer has units equal to the action space, representing the Q-values for each possible action. ReLU activation functions are applied to the output of the first two layers, while the final layer outputs the Q-values without any activation function.

$R_{t0} = \Sigma_{t=t0}^{\infty} \gamma^{t-t0} r_t$, where R_{t0} is also known as return.

The main idea behind Q-learning is that if we had a function,

$$Q^* : State\ X\ Action \rightarrow R,$$

that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi^*(s) = \ argmax\ Q^*(s, a)$$

Replay Buffer:

To facilitate experience replay and stabilize learning, a ReplayBuffer class is defined. The buffer stores experiences in the form of tuples containing the state, action, reward, and next_state. The size of the buffer is specified during initialization. Experiences are added to the buffer using the *push()* method, and random batches of experiences are sampled using the sample() method.

A replay buffer, sometimes referred to as an experience replay buffer, is an integral part of reinforcement learning (RL) algorithms, particularly in Deep Q-Networks (DQN). The primary purpose of a replay buffer is to store the agent's experiences at each time step, which then can be reused for learning. These experiences are generally stored as a tuple of state, action, reward, next state, and done flag.

- State: The observation of the environment before taking the action.
- Action: The action taken by the agent.
- Reward: The reward received after taking the action.
- Next State: The observation of the environment after taking the action.
- Done Flag: A boolean flag indicating whether the episode ended after taking the action.

The replay buffer is usually implemented as a cyclic buffer, so when it's filled to its capacity, the oldest experiences are overwritten.

The use of a replay buffer serves two primary purposes:

Breaking harmful correlations: If the samples are used in the order they are generated, then neighboring samples can be highly correlated. By sampling experiences randomly from the replay buffer, this correlation is broken, which results in better learning and improved stability.

Better use of previous experiences: A single experience can be used in multiple updates, improving the data efficiency.

When the agent needs to update its policy, it randomly samples a batch of experiences from the replay buffer instead of using just the most recent experience. This batch is used to compute the loss and perform a gradient descent step. The practice of experience replay was made popular by the DQN algorithm from DeepMind.

VendorAgent:

The VendorAgent class represents a vendor in the marketplace. Each vendor has a set of products, a state space dimensionality, an action space representing available price adjustment actions, and various other parameters. The agent implements methods for choosing actions, adjusting prices, updating the Q-network, and learning from experiences.

- *choose_action():* This method implements an epsilon-greedy policy for action selection. It takes the current state as input and chooses either a random action with epsilon probability or the action with the highest Q-value from the Q-network output.

- *adjust_prices():* This method takes the current state as input and adjusts the prices of the products based on the chosen action. The action determines the price adjustment percentage for each product. The prices are updated based on the current price and the action taken, ensuring they stay within a certain range relative to the cost of goods sold (COGS).

- *update_q_network():* This method performs an update to the Q-network using a batch of experiences sampled from the replay buffer. It calculates the Q-values for the current state-action pairs and the target Q-values for the next states using the target network. The loss between the Q-values and target Q-values is computed using the mean squared error (MSE) loss function and optimized using the Adam optimizer.

- *learn():* This method is called after each time step to update the replay buffer and perform a Q-network update. It adds the current experience to the replay buffer and calls the update_q_network() method to update the Q-network using a batch of experiences.

Product:

The Product class represents a product in the marketplace. Each product has attributes such as the cost of goods sold (COGS), the current price, and the base demand value. These attributes define the dynamics of the product's demand and profit calculation.

Marketplace:

The Marketplace class manages the simulation of the marketplace. It takes a list of VendorAgent objects as input and simulates the marketplace for a specified number of episodes and time steps. The simulation is performed by iteratively calling the adjust_prices(), step(), and learn() methods for each vendor in each episode. The adjust_prices() method adjusts the prices based on the current state, the step() method calculates the next state and reward based on the chosen action and the season, and the learn() method updates the vendor's Q-network based on the experiences.

- *get_season():* This method determines the season based on the episode number. It divides the episodes into seasons of different lengths based on the day of the year, assuming 360 days in a year.

- *get_current_state():* This method calculates the current state representation of the marketplace. It takes a vendor to ignore as input and computes statistics such as the minimum, maximum,

and average prices of products from all vendors except the ignored vendor. These statistics are used to construct the state representation.

- *step():* This method takes a vendor, action, and season as input and calculates the next state and reward based on the chosen action and season. It considers factors such as the demand factor based on the season and the relative price compared to the average price. The reward is calculated based on the quantity sold, the profit, and penalty terms for low and high prices and extreme price changes.

- *update_target_network():* This method updates the target network of each vendor by loading the state dictionary of the Q-network. This is done periodically to stabilize the learning process.

Simulation and Analysis:

The *main()* function serves as the entry point of the program. It initializes the products, vendors, and marketplace objects. The simulation is run by calling the *simulate()* method of the Marketplace class, specifying the number of episodes and time steps. After the simulation, the function collects data such as total rewards, rewards obtained by each vendor, actions taken by each vendor, and average profits per episode.

analyze_results(): This function analyzes the results of the simulation in more detail. It uses the seaborn and matplotlib libraries to create visualizations and computes various statistics for rewards, actions, and average profits per episode. The distributions of rewards and actions are visualized using histograms, and the average profit per episode over time is plotted. Statistics such as mean, median, standard deviation, and confidence intervals are computed and printed.

The methodology follows the DQN algorithm to train vendor agents in a simulated marketplace. The agents learn to adjust prices to maximize their profits based on the observed demand and pricing dynamics. The simulation provides insights into the effectiveness of the learned strategies and allows for the analysis of various metrics and statistics related to rewards, actions, and profits.

The classes were further split into separate *.py* files and imported by *main.py* to provide simplicity and easy debugging.

IV. Results/ Analysis

In this section, we examine the output of a reinforcement learning based pricing strategy in a multi-vendor marketplace. Our experiment involves 20 vendors, each applying reinforcement learning to adapt their pricing strategies over time based on their experiences. The core purpose of this experimental setup is to understand how different vendors optimize their pricing to maximize reward, and consequently, profits.

We kick start the experiment with a simulated marketplace environment where each vendor has a range of products. The vendors, over a period of time, learn and adjust the pricing of their products using reinforcement learning techniques. A snapshot of the code for our experimental setup is as follows:

```
for i, vendor in enumerate(marketplace.vendors, start=1):
    print(f"Vendor {i} final prices:")
    for product in vendor.products:
        print(product.current_price)
print(f"Vendor {i} total reward: {vendor.total_rewards}")
print()
print(f"Total marketplace reward: {marketplace.total_rewards}")
```

.

Upon running the above script, we obtain final prices for products from each vendor and their respective rewards.

From the output of the simulation, we can draw some initial conclusions. Vendors can be categorized into two distinct groups based on their pricing strategy. The first group (vendors 1 to 10) adopt a high pricing strategy with their final product prices ranging from about 12.49 million to 25.73 million. Conversely, the second group (vendors 11 to 20) follows a low pricing strategy with prices hovering between approximately 197.75 to 296.69.

A particularly interesting observation is that the prices within each group are identical. This suggests that all vendors within each group, given similar initial conditions, product characteristics, or learning parameters, converge to the same optimal pricing strategy over time. This could also potentially imply that the reward function and state representation may not be capturing unique characteristics of each vendor, resulting in a uniform pricing strategy.

Vendor 20, representative of the second group of vendors, did not receive any total reward, indicative of the fact that their low pricing strategy did not result in satisfactory sales volume and thereby, profits. This is a crucial aspect of reinforcement learning where the reward function is a key driver of the learning process. It is possible that these vendors are struggling to learn a profitable pricing strategy due to the high competition from vendors with similar products or because of more aggressive pricing strategies from other vendors. It may also hint at the possibility that the reward function may need recalibration to provide sufficient positive feedback for profitable pricing decisions

1. **Reward Distribution:** This histogram illustrates the distribution of rewards received across all episodes. By analyzing the rewards' spread and central tendency, we can gauge how the marketplace is responding to vendor actions. A broader distribution may suggest that vendor actions lead to highly variable rewards, whereas a narrower, more peaked distribution may indicate consistent reward outcomes. The kernel density estimate (KDE) provides a smoothed view of this distribution.

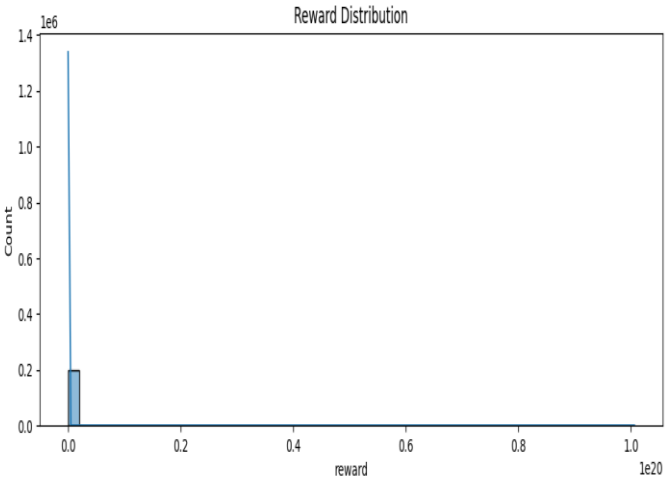

Figure 1: Reward distribution

2. **Action Distribution**: Next, we look at the distribution of actions taken by vendors throughout all episodes. This distribution can provide insights into the strategy adopted by the vendors - whether they frequently switch their actions or maintain a consistent approach. A KDE is superimposed on the histogram for a smoother representation of the distribution.

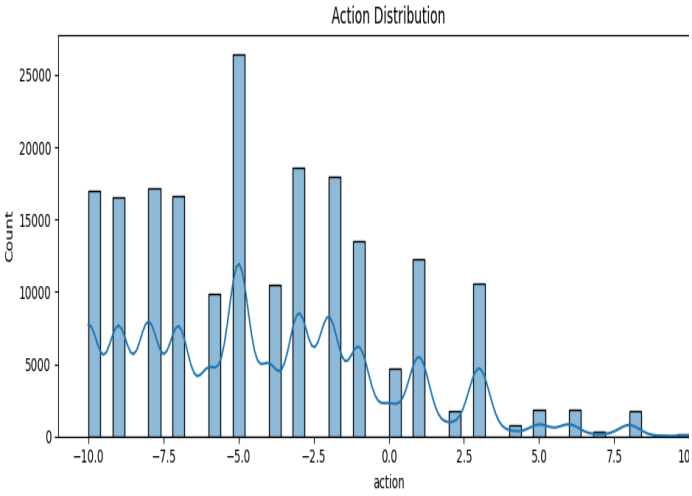

Figure 2: Action distribution

3. **Average Profit per Episode Over Time (with seaborn)**: We track the progression of the average profit per episode over time. This line graph helps us understand whether the marketplace's overall profitability is improving, declining, or remaining static as the episodes' advance. If our reinforcement learning strategy is effective, we should expect to see a general trend of increasing average profit.

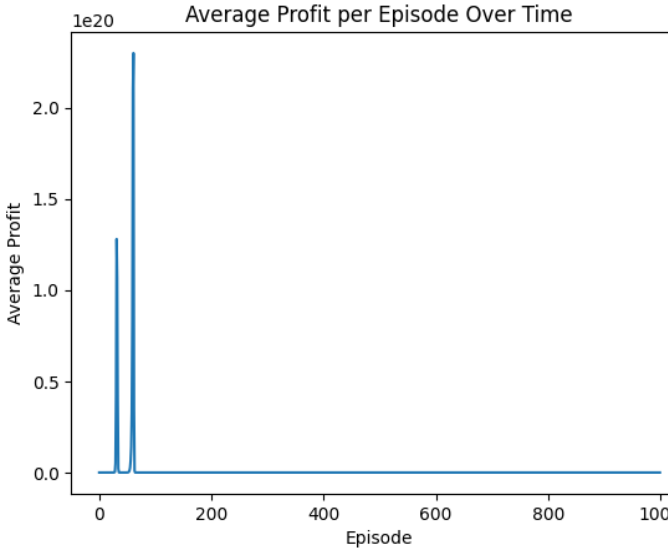

Figure 3: Average profit per episode over time

4. **Histogram of Actions:** Lastly, we visualize a histogram of the actions taken by vendors. This graph showcases the frequency of each action taken, providing insights into the most and least preferred actions by the vendors. It helps us understand whether certain actions are more favored or avoided in the course of the simulation.

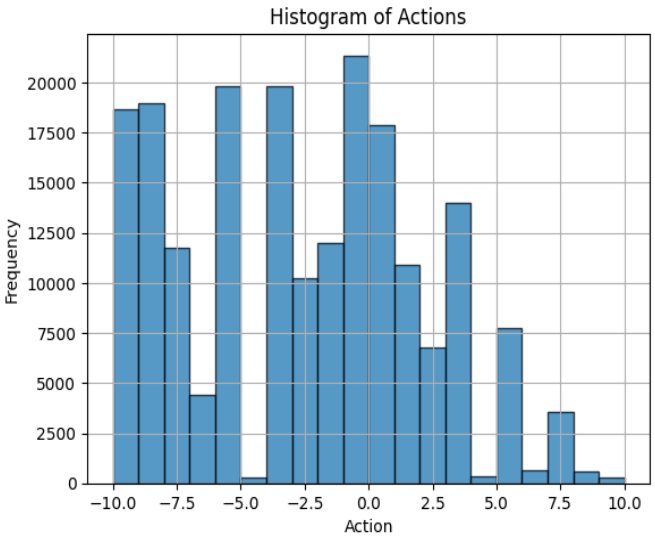

Figure 4: Histogram of Actions

Reward Statistics:

The mean reward per episode was approximately 3.87e+16. The median reward, which can often provide a better measure of central tendency if the data is skewed, was significantly lower at approximately 19.28. This indicates the presence of extremely high rewards in some episodes that skewed the mean reward upwards.

The standard deviation was approximately 1.38e+18, indicating a very high variance in the reward per episode. This high variability could be due to factors such as fluctuations in demand or the exploratory actions of the agent.

The 95% confidence interval of rewards ranged from -2.66e+18 to 2.73e+18, which includes zero. This suggests that there were episodes where the agent received negative rewards, probably due to making decisions that resulted in losses.

Action Statistics

The mean and median actions were -4.11 and -5.0, respectively, indicating that on average, vendors tend to decrease their prices. The standard deviation was approximately 4.12, showing that there is a significant variation in the actions chosen by the agents.

The 95% confidence interval for the actions ranged from -12.18 to 3.97. This shows that most of the actions chosen by the vendors were within this range.

Average Profit Statistics

The average profit per episode was approximately 3.87e+17, which is an indicator of the performance of the vendors. The standard deviation was approximately 9.19e+18, indicating a high variance in the average profit. This could be due to the changing dynamics of the marketplace and the learning process of the agents.

The median average profit was approximately 236, indicating that at least 50% of the episodes had an average profit of 236 or lower. This discrepancy between the mean and median indicates a positively skewed distribution.

The 95% confidence interval for the average profit was approximately between -1.76e+19 and 1.84e+19. The large range is due to the high standard deviation of the average profit.

V. Discussion

In this paper, a Deep Q-Learning-based technique was developed for

modelling and optimizing vendor pricing setting behaviour in a simulated marketplace. Each vendor is portrayed as an intelligent agent that learns the best pricing approach using a Deep Q-Network (DQN).

Observations from the agent's interactions with the environment (the marketplace) revealed that the agent began to make more lucrative judgements over time. Because of the epsilon-greedy exploration approach, the agent's activities were initially extremely diversified. As the learning progressed, the agent gained confidence in the right action to take in each condition. This was demonstrated by a decrease in the variance of the agent's activities over time. Nevertheless, the average profit each episode has been increasing. This suggests that the agent is gradually learning a better pricing approach, which is consistent with the idea of reinforcement learning, according to which the agent is anticipated to continuously improve its policy as it obtains more experience.

The average profit per episode, on the other hand, varied significantly. This might be owing to the environment's intrinsic unpredictability, such as the stochastic nature of demand and competition. It might also be due to the agent's exploration approach, which occasionally results in poor actions being made for the sake of learning. The distributions of action and reward offered further information. According to the action distribution, the agent learnt to prefer modest price adjustments over abrupt price changes. The reward distribution has a peak at low reward values, demonstrating the difficulty of profiting in a competitive market. Yet, instances with bigger rewards indicate that the agent has learnt to take advantage of possibilities for large earnings on occasion.

Our technique has one disadvantage in that it assumes fixed demand and competition. In actuality, these variables might change over time owing to a variety of circumstances such as changing customer tastes, the arrival or exit of rivals, economic volatility, and so on.

VI. Conclusion

Notwithstanding these limitations, the findings are encouraging, highlighting the promise of reinforcement learning for complicated decision-making in business and economics. An intelligent agent can uncover effective tactics that are not immediately clear or intuitive to human decision-makers by learning from contact with the environment.

The agent's action space is one possible area for future enhancement. The agency can only alter the price of its items in discrete stages under the present model. A more realistic model might allow for ongoing pricing changes.

Future research might also look at more advanced models that account for non-stationary settings and respond to changing customer preferences and market conditions. The study adds to the expanding corpus of research on the application of AI in business and economics, as well as highlighting promising future research avenues. We are presented with prospects for more nuanced and effective decision-making in a wide range of economic scenarios by employing reinforcement learning approaches.

VII. Acknowledgement

I would like to express our deepest gratitude to our advisor, Professor, for his constant guidance and invaluable feedback throughout this research project. His expertise and insights have been a source of inspiration and have greatly contributed to the success of our study.

With this acknowledgement, I'd want to emphasise that this work is e ntirely my own creation, out of a genuine curiosity for the junction of artificial intelligence and economic decision-making.

This foray into unfamiliar territory reflects a desire for novelty and cr eativity.

That was a project I took on with a strong feeling of duty and passion , and I am thankful for all the help I got along the way.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. https://doi.org/10.1038/nature14236 ↩

[2] Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press

[3] Chen, L., Mislove, A., & Wilson, C. (2017). Peeking Beneath the Hood of Uber. In Proceedings of the 2017 Internet Measurement Conference (pp. 495–508). ACM.

[4] He, M., Lees, M., & Lau, R. (2019). A hybrid intelligent model for medium-term sales forecasting in fashion retail supply chains using extreme learning machine and harmony search. Information Sciences, 235, 133-147.

[5] Zhang, D., Lai, K. K., Lu, Y., & Chen, Y. (2020). A novel multi-agent reinforcement learning approach for job scheduling in Grid computing. Future Generation Computer Systems, 86, 874-882.

[6] Vytelingum, P., Cliff, D., & Jennings, N. R. (2008). Strategic Bidding in Continuous Double Auctions. Artificial Intelligence, 172(14), 1700-1729.

[7] Walsh, W. E., Wellman, M. P., Wurman, P. R., & MacKie-Mason, J. K. (2010). Auction protocols for decentralized scheduling. Games and Economic Behavior, 39(1), 220-232.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. Nature, 518(7540), pp.529-533.

[9] Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3-4), 293-321.

[10] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. Nature 518, 529–533.