Project Reinforcement Learning

Energy Management for Dynamic Data Centers

Daniele Virzì (2859327)
Sam Kramer (2853425)
Lennard Schuenemann (14865610)

Group 19

Faculty of Science
Vrije Universiteit Amsterdam
Dr. Francois Lavet

January 30, 2025

1 Problem Description

Energy management has become essential due to the move toward renewable energy and the requirement for effective operations. Reinforcement learning (RL) is a viable way to balance conflicting goals in energy systems because of its capacity to adjust and optimize judgments in dynamic contexts. The objective is to use reinforcement learning (RL) to create an operational control strategy for a data center with variable energy consumption and limited energy storage capacity. Under dynamic and stochastic conditions, such as fluctuating electricity prices, the data center must effectively manage its energy consumption, optimize energy expenditures, and communicate with the electrical grid.
The following are important components of the problem:

Energy Demand: The data center can buy energy at any time of day, but it must buy 120 MWh of energy every day by midnight (*force-buy mechanism*).

Energy Storage: Due to losses and financial agreements, the data center can sell electricity back to the grid at 80% of the price. It can also store up to 50 MWh of excess energy for the next day, with any additional excess energy being wasted.

Constraints: The maximum power rate for either buying or selling is 10MW, and the storage level cannot be lower than 0 MWh.

Stochastic Electricity Prices: Given the unpredictability of electricity pricing, decisions on whether to buy, store, or sell must be made in real-time using both past and current data.

Objective: Reduce energy expenses while maintaining your needs, utilizing storage, and adjusting demand to reflect changes in prices.

Reinforcement Learning: By learning the best energy techniques in an environment of unpredictable and unpredictable pricing, RL is well-suited to this problem.

The challenge is to find a balance between energy trading, storage, and consumption to save costs while preserving operational dependability and conforming to all regulations.

2 Environment Setup

The environment simulates the operation of a data center managing energy consumption and costs with an energy storage system (Figure 1). The state space provides the agent with the knowledge required to make decisions regarding energy buying, selling, or storing. It contains the current energy level, the current electricity price, and the time of the day. The action state provides a continuous value [-1, 1] representing energy buying, selling, or storing. A positive value suggests energy buying, while a negative value suggests energy selling, and 0 indicates holding the energy. The weight of the value indicates the percentage of energy that is transacted (no more than 10MW). The reward function is compared to cost savings and/or penalties. If energy is sold, the data center receives a positive reward,

suggesting cost savings. On the other hand, when energy is bought, the environment receives a negative penalty because the data center makes a cost.
Furthermore, the environment incorporates the constraints and assumptions outlined in the problem description. It is trained using three years of historical hourly electricity prices to replicate market conditions that influence the cost of buying or selling energy. Each episode corresponds to a full traversal of the three-year price history.

3 Tabular Reinforcement Learning

To solve the problem, we implemented an agent that acts in the environment using a tabular model-free algorithm, *Q-learning*.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \cdot \max_a Q(s',a) - Q(s,a)\big] \tag{1}$$

The key components of the Q-agent are:

State-Action Value Estimation: The agent stores the expected rewards for state-action pairings in a Q-table using *Bellman equation* (Equation 1).

Off-Policy Learning: The agent learns an optimal policy without actually implementing it.

Exploration-Exploitation Trade-Off: The agent follows an ϵ-greedy policy to balance exploration and exploitation. With probability ϵ, the agent chooses a random action (exploration), while with probability $1-\epsilon$, it selects the greedy action that maximizes the Q-value based on the current Q-table (exploitation).

Discrete State and Action Space: The Q-table is of dimension $|\mathcal{S}| \times |\mathcal{A}|$, where $|\mathcal{S}|$ represents the number of possible states, and $|\mathcal{A}|$ represents the number of possible actions. Each entry in the Q-table corresponds to the estimated value of taking a particular action in a specific state. Observations and action spaces in the environment are continuous, so discretization is crucial.

3.1 Q-Learning (Baseline)

To perform in this custom environment, we implemented the following techniques:

State Discretization and Feature Selection: To reduce the complexity of the state space and mitigate the curse of dimensionality, we selected only the storage level and the hour as features to describe the state. The storage levels are discretized into two bins, ranging from empty to maximum capacity (0-72.5, 72.5-290). While some states may not be frequently explored, this discretization was chosen because it yielded the best empirical results.

Action Discretization: To enable the Q-agent to interact with the environment, the action space was discretized as $|\mathcal{A}| = \{-1, 0, 1\}$. Here, 1 represents the action

"buy" 100% of the maximum power rate (10 MW), 0 corresponds to *"hold"*, and -1 indicates *"sell"*.

Reward Shaping: The reward function is adjusted to encourage optimal utilization of resources by incorporating the storage level and average price into the reward calculation. The reward shaping function can be expressed as:

$$r_{\text{shaped}} = \begin{cases} r + \tau \cdot \text{storage\_level}, & \text{if action} = 0 \\ r + \tau \cdot \text{storage\_level} + 10 \cdot \text{average\_price}, & \text{if action} = 1 \\ r + \tau \cdot \text{storage\_level} - 10 \cdot \text{average\_price}, & \text{if action} = -1 \end{cases}$$

Here, τ (set to 0.15) is a scaling factor that determines the weight of the storage level's contribution, while average_price is incorporated to adjust rewards for specific actions based on market trends.

Moving Average: To compute the average price, we used an Exponential Weighted Moving Average (EWMA) with a smoothing factor $\beta = 0.983$ [Perry, 2010]. This approach ensures that recent price data has a higher weight, making the moving average more responsive to market trends while still retaining the influence of historical data.

The EWMA is calculated iteratively as follows:

$$\text{average\_price}_t = \beta \cdot \text{average\_price}_{t-1} + (1 - \beta) \cdot \text{price}_t$$

where average_price_t is the updated moving average at time t, $\text{average\_price}_{t-1}$ is the moving average from the previous time step, and price_t is the current observed price. We tried different approaches to computing the moving average daily, weekly, monthly, and episodely; we achieved the best result with the daily moving average.

In this implementation, bias correction was not applied, meaning the initial values may slightly underestimate the true average until the weights stabilize over time. This trade-off was acceptable due to the focus on long-term trends rather than short-term precision.

Adaptive Learning Rate: The learning rate starts with a default value of 0.1. To account for the possibility of noisy or variable rewards, an adaptive learning rate is used, defined as:

$$\alpha_{i+1} = \frac{\alpha_i}{\sqrt{\text{i}}}$$

where α_1 is the initial learning rate (0.1), and i refers to the current episode number. This approach ensures smaller incremental updates to the Q-values as training progresses, avoiding large shifts in the learned policy.

.

Decaying Discount Factor: The discount factor γ is initialized to 0.99 and decays at a rate of $\gamma_{\text{decay}} = 0.999$ until it reaches a minimum value of $\gamma_{\text{min}} = 0.03$. This gradual reduction allows the agent to initially prioritize long-term rewards during training and shift its focus toward short-term rewards as the training progresses, which is particularly useful in dynamic environments like energy management.

Epsilon Decay for ϵ-Greedy Policy: The exploration parameter ϵ is initialized to 1.0 and decays at a rate of $\epsilon_{\text{decay}} = 0.995$ until it reaches a minimum value of $\epsilon_{\text{min}} = 0.01$. This allows the agent to explore the environment extensively during the early stages of training, gradually shifting toward exploitation of the learned policy as training progresses. During evaluation, ϵ is set to 0, ensuring pure exploitation of the learned policy.

As illustrated in Figure 2, the agent demonstrates consistent learning across episodes with minimal variance. Notably, it converges rapidly, requiring fewer than 1,000 episodes, with each episode taking less than one second to complete. See the Q-table and the learned policy in Figure 3 and Figure 4.

3.2 Improved Q-Learning

By removing the price from the state space, the algorithm no longer considers hourly price fluctuations, causing it to behave deterministically each day regardless of market conditions. This leads to overfitting. To address this issue, we enhanced our baseline model by incorporating the hourly price feature into the state space using four quartile-based bins. While this adjustment resulted in slightly worse performance, it allowed the algorithm to adapt dynamically rather than following a fixed daily pattern. The difference between the two policies is illustrated in Figure 6 and Figure 7.

4 Deep Reinforcement Learning

Once we achieved promising results with our tabular RL agent as a baseline, we transitioned to Deep Reinforcement Learning (Deep RL). By leveraging neural networks, the agent was able to approximate complex state representations and effectively learn an optimal policy. We experimented with both value-based and policy-based approaches to improve upon the performance of our baseline model.

4.1 Double Deep Q-Network (DDQN)

Q-learning struggles with large state spaces due to its reliance on a tabular Q-function. Deep Q-Networks (DQN) address this by approximating the Q-function with a neural network, enabling learning in high-dimensional environments [Mnih et al., 2013]. However, DQN suffers from instability and overestimation bias, which DDQN mitigates through a fixed target network and a decoupled action selection process [van Hasselt et al., 2015].

4.1.1 Fixed Target Network

Using the same network for both target calculation and Q-value updates causes instability. To address this, a fixed target network $\hat{Q}$ is updated periodically to stabilize learning. The Q-learning target is modified as:

$$Y_t = r + \gamma \max_a \hat{Q}(s', a), \tag{2}$$

where $\hat{Q}$ is the target network, reducing the correlation between target values and learned parameters [Mnih et al., 2013].

4.1.2 Double Q-Learning

Q-learning tends to overestimate Q-values due to the maximization step over future rewards. DDQN reduces this bias by using separate networks for action selection and evaluation [van Hasselt et al., 2015]:

$$Y_t^{DDQN} = r + \gamma Q(s', \arg\max_a Q(S_{t+1}, a); \Theta_t^-). \tag{3}$$

Instead of directly maximizing over the target network, the online network selects the action, and the target network evaluates it, mitigating overestimation.

4.1.3 (Prioritized) Experience Replay

To improve sample efficiency and stability, experience replay [Mnih et al., 2013] stores past experiences and samples them randomly, reducing data correlation. Prioritized experience replay further enhances learning by sampling experiences with higher expected impact based on the TD error [Schaul et al., 2016]:

$$p(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \quad p_i = |\delta_i| + \epsilon. \tag{4}$$

This prioritization is balanced with importance sampling to correct bias while ensuring recent experiences are replayed at least once.

See Figure 8 to see training rewards of the DDQN agent.

4.2 REINFORCE with baseline

We implemented the standard REINFORCE algorithm [Williams, 1992], and extended it with a value-based baseline in order to reduce variance in the updates. The update rule is as follows:

$$\theta = \theta + \alpha \sum_{t=0}^{T} (G_t - b(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{5}$$

We chose to use this method since it is easy to implement and enables learning a stochastic policy as well as a continuous actions. It also has an unbiased estimator of the policy gradient since it uses the Monte Carlo return. By including a baseline, we also eliminate the algorithms main drawback.

4.3 Proximal Policy Optimization (PPO)

As a more advanced policy method, we chose to implement PPO, which we think would be particularly suited to the environment since it replays experiences within a batch even though it is on-policy, thus being able to learn from limited data, while also learning a stochastic policy which naturally enables exploration [Schulman et al., 2017].

The PPO objective function is:

$$J(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t \right) \right] \tag{6}$$

where: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, A_t is the advantage estimate, and ϵ is the clipping parameter.

The policy parameter update rule is:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{7}$$

For the critic (value function), the update follows:

$$L_V(\theta) = \mathbb{E}_t \left[\left(V_\theta(s_t) - V_t^{\text{target}} \right)^2 \right] \tag{8}$$

where V_t^{target} is in our case estimated using Generalized Advantage Estimation (GAE).

5 Performance Evaluation

The performance of the trained agent was evaluated on two years of unseen historical data. This validation set allowed us to assess the generalization ability of the agent, ensuring that it could make effective decisions when confronted with new data that was not part of the training set.

During the evaluation process, we performed hyperparameter tuning using this validation data. The final hyperparameters chosen were those that yielded the best performance in terms of cumulative rewards, which were calculated over the course of a single episode, in compliance with the professor's request. Cumulative rewards were used as the key metric to evaluate the agent's performance, as this provides a clear indication of the overall effectiveness of the policy over time. This approach allows for assessing the long-term performance of the agent, rather than focusing on immediate or short-term outcomes.

5.1 Benchmarks

Since the environment simulates a real-world scenario and lacks established benchmarks for this task, we evaluated the quality of our policy by comparing its cumulative reward against heuristic policies. In Table 1 we can see the cumulative rewards achieved by different heuristic policies on both the training and validation sets. For more details, refer to the GitHub file *agent.py*.

Policy	Training Set	Validation Set
Always Buy Max	-13,304,051	-8,373,839
Random	-7,417,251	-4,563,585
Do Nothing	-7,544,268	-4,600,066
Maintain Daily Requirement	-6,229,228	-3,781,382

Table 1: Comparison of Heuristic Policies (Cumulative reward)

5.2 Results

Table 2 presents the cumulative rewards achieved by different reinforcement learning algorithms on both the training and validation sets.

Algorithm	Training Set	Validation Set
Q-learning (baseline)	**-5,285,044**	**-3,604,366**
Q-learning (improved)	-5,441,744	-3,621,468
DDQN	-7,170,930	-4,390,602
PPO	-7,213,100	-4,628,398
REINFORCE	-7,398,224	-5,043,190

Table 2: Performance Comparison of RL Algorithms (Cumulative reward)

6 Challenges during RL training

One of the main challenges was preventing the agent from triggering the *force-buy mechanism*, which often led to poor rewards (as seen in the *"Do Nothing"* policy). Ultimately, our models consistently opted to buy large amounts of energy within the first 12 hours while avoiding selling, effectively bypassing higher prices later in the day. Several issues appeared during RL training, including learning instability. The agent struggled to balance energy purchasing, selling, and storage amid fluctuating electricity prices, often leading to oscillations or divergence (especially in Deep RL). Another challenge was efficiently discretizing the state space to retain relevant information while mitigating the curse of dimensionality. Additionally, hyperparameter tuning demanded careful attention, as even changes in the random seed had a significant impact on Q-value updates.

7 Conclusion

The agent successfully learned to make earlier purchasing decisions, improving its performance over the Heuristic policies. The baseline Q-learning algorithm outperformed Deep RL algorithms and yielded promising results, achieving approximately **29.95%** savings compared to the *"Do Nothing"* policy on the training set, and **21.65%** savings on the validation set. These findings demonstrate the effectiveness of reinforcement learning in optimizing purchasing strategies.

8 Code and Pre-trained model

The GitHub repository contains the code with pre-trained models that can be applied to new time series, a Python notebook for visualization, and the complete dataset. It also includes a README file with instructions on reproducing the model training and utilizing the pre-trained model.

9 *

References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Perry, M. (2010). *The Exponentially Weighted Moving Average*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

10 Appendix

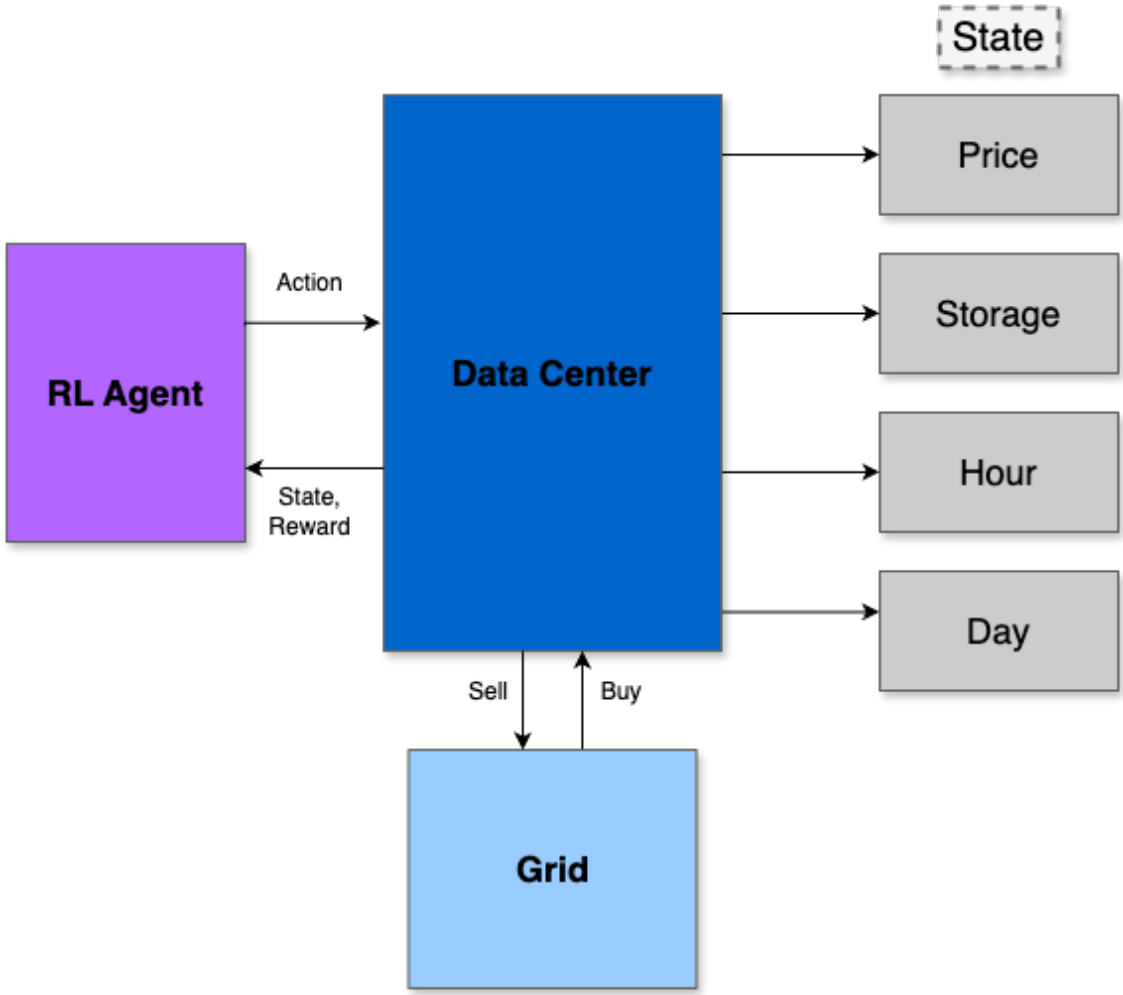

Figure 1: Environment Workflow

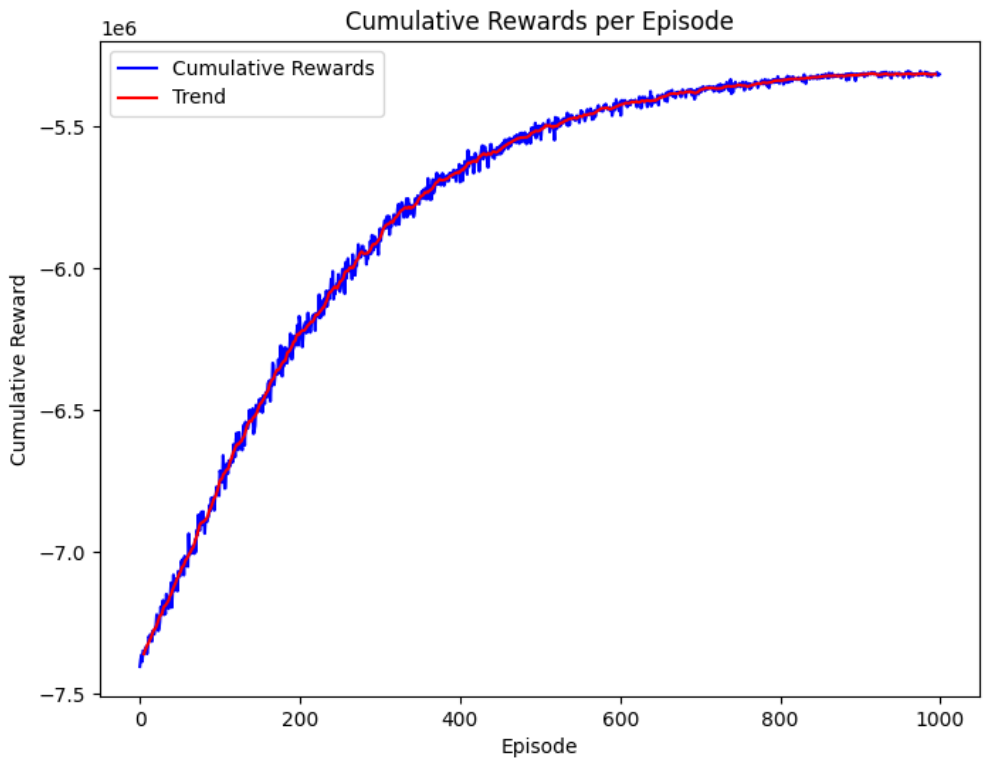

Figure 2: Cumulative Reward of the baseline agent during training

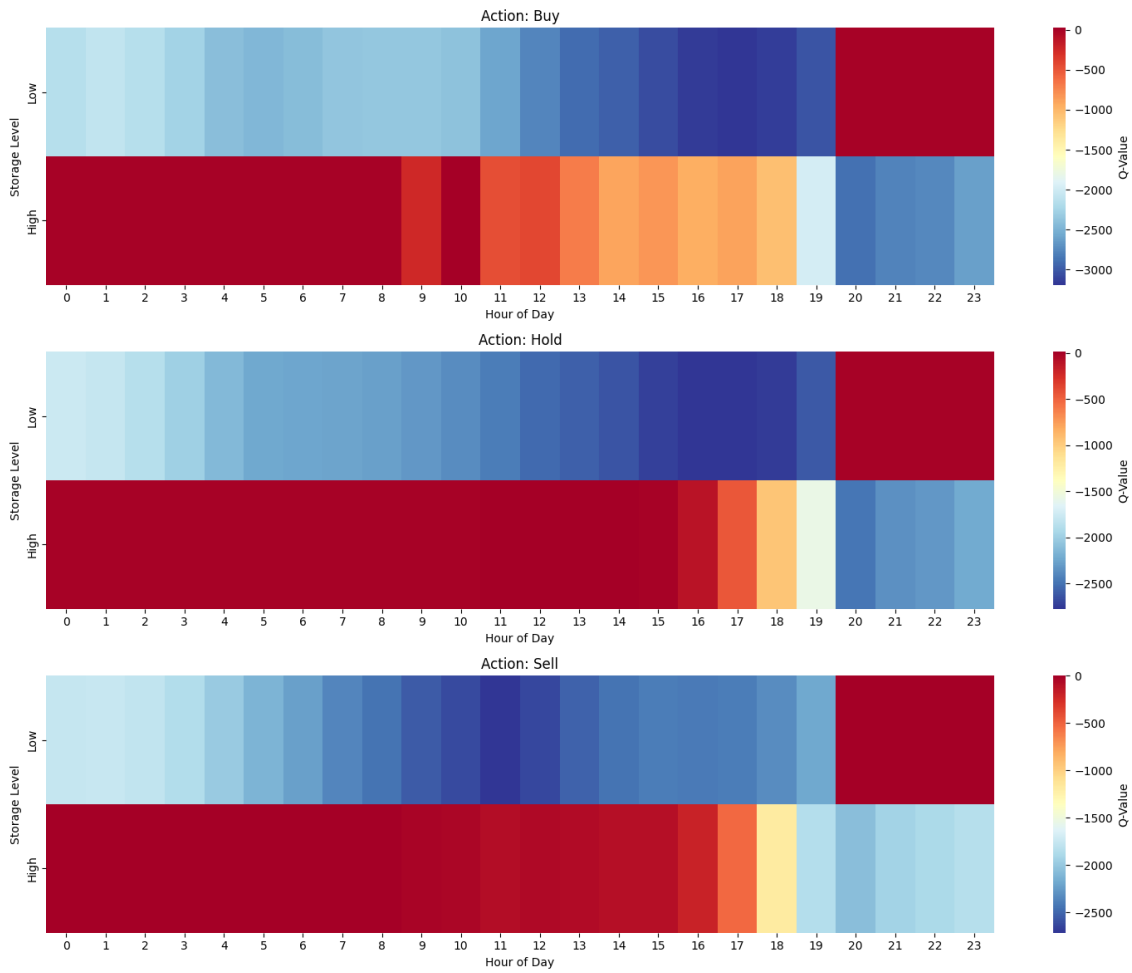

Figure 3: Q-table of the baseline agent

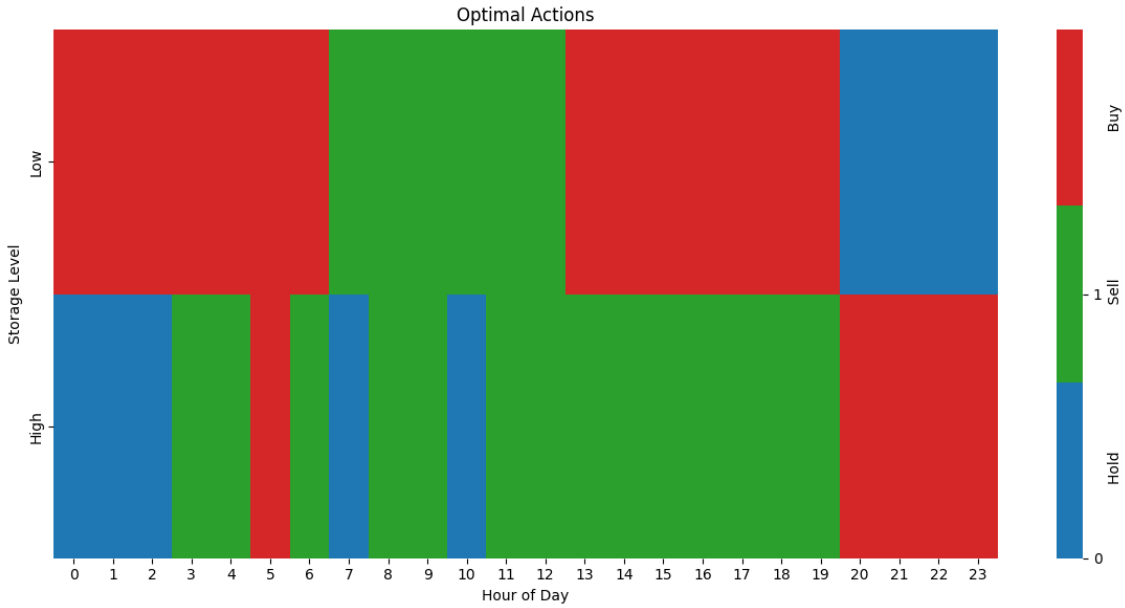

Figure 4: Policy of the baseline agent

Figure 5: Daily action pattern of the baseline agent on validation set

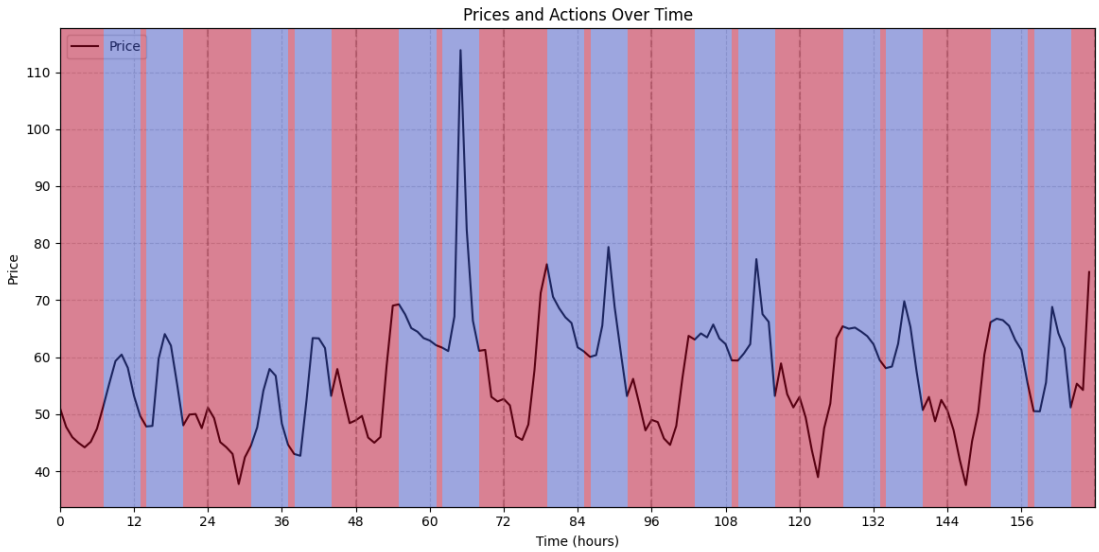

Figure 6: Weekly action pattern of the baseline Q-agent on validation set

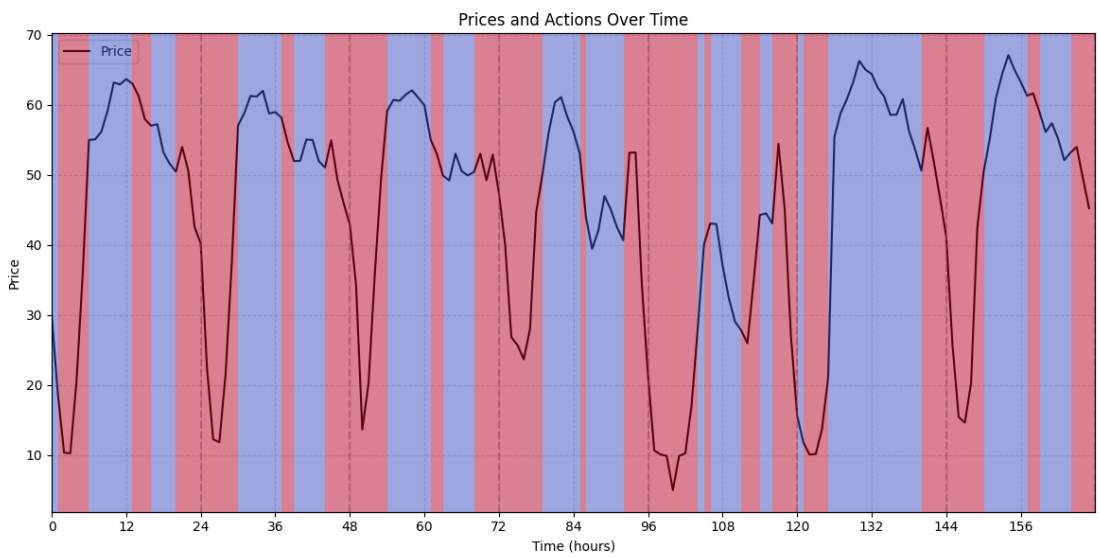

Figure 7: Weekly action pattern of the improved Q-agent on validation set

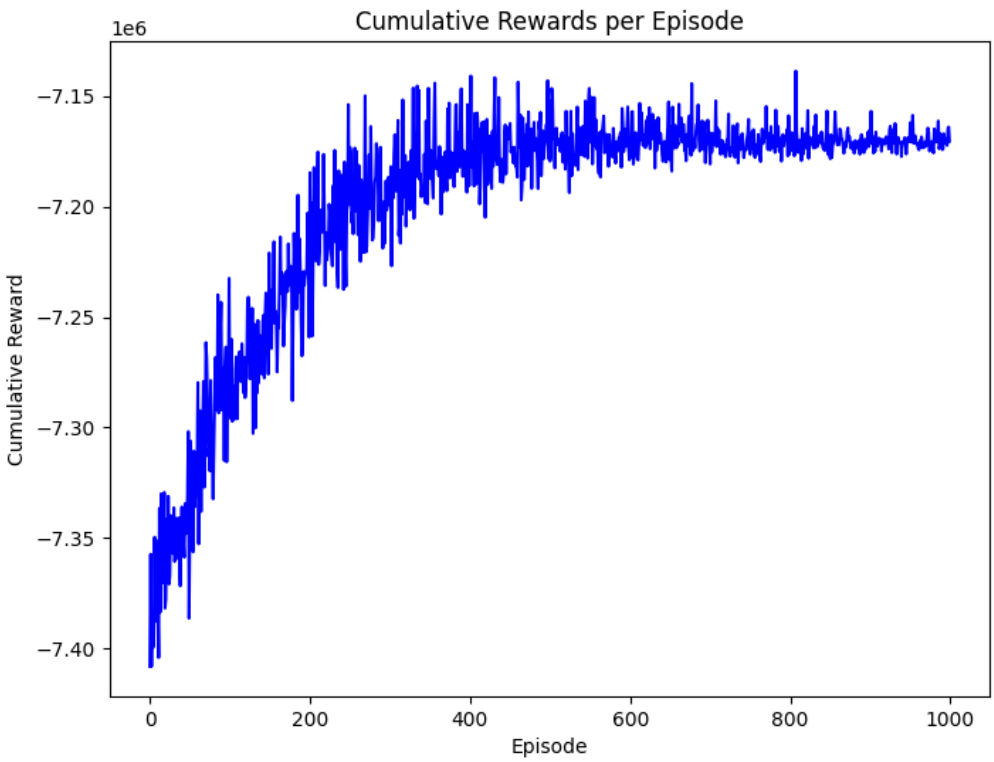

Figure 8: Cumulative Reward of the DDQN agent during training