Reinforcement Learning Project Report - Group 5

Julien P.Testu (2695398) Ege Toker Dinc (2819794) Matus Halak (2724858)

Group 5

January 2025

1 Introduction

Data centers are critical to modern computing but face energy management challenges due to their exceptionally high power demands and dynamic electricity prices. This project focuses on optimizing energy usage in a simulated data center with flexible purchasing and limited storage. The facility must meet a daily demand of 120 MWh, purchase electricity at a maximum rate of 10 MW each hour at dynamic prices, and manage storage to avoid waste or depletion. However, selling excess energy to the grid incurs a 20% loss. The goal is to minimize costs while ensuring a reliable energy supply, using only historical and current price data without knowledge of future prices.

We use reinforcement learning (RL) since the problem at hand can be classified as sequential decision-making under uncertainty, which falls into RL territory. By training an RL agent to dynamically adjust energy purchases and sales, we aim to develop a cost-effective strategy that adheres to operational constraints. We use varying approaches in order to showcase how they perform on the same task when similar amounts of resources are invested to develop a solution with each method. Our research aim is to investigate which approach among Tabular Q-Learning and Double Deep Q-Learning compares better to a selection of baselines, under the constraint that similar and limited time, compute, and human effort were allocated to developing a solution using both methods. We hope this work provides actionable and realistic insight when planning for energy cost optimization in dynamic real-world applications.

The report contains, respectively: the formalization of the problem summarized above, the methods chosen to tackle this optimization problem, the results we achieved along with their interpretation, and finally some concluding remarks that highlight the insights drawn from our experiments.

2 Problem Formalization

The energy management problem for the datacenter is modeled as a Markov Decision Process (MDP), which provides a mathematical framework for decision-making in stochastic environments. An MDP is defined by a tuple (S, A, P, R), where:

2.1 States (S)

The state space S represents the current situation of the datacenter at a given time step. A state $s \in S$ encodes the following key elements:

- **Current energy storage level** (in MWh): Indicates the amount of energy available in the storage system.
- **Electricity price** (in euros/MWh): Represents the current market price of electricity.
- **Time of day**: The current hour, integer in [1, 24].
- **Day index**: The number of days passed since the simulation started.

2.2 Actions (A)

The action space A defines the possible decisions the agent (control system) can take at each time step. Actions $a \in A$ include:

- **Energy purchase**: Decide how much energy to buy from the grid (up to 10 MW per time step).
- **Energy sale**: Decide how much energy to sell to the grid (up to 10 MW per time step).
- **No operation**: Choose not to buy or sell energy during the current time step.

2.3 Rewards (R)

The reward function $R(s, a)$ quantifies the agent's performance at each time step based on its chosen action a in state s. The reward is designed to:

- **Minimize costs**: Encourage purchasing electricity at lower prices and avoiding expensive time periods.
- **Maximize revenues**: Incentivize selling electricity during high-price periods.
- **Avoid penalties**: Penalize actions that lead to:
  - Falling below the daily energy requirement (120 MWh).
  - Exceeding the storage capacity or causing energy wastage (more than 50 MWh of surplus).

2.4 Transitions (P)

The transition dynamics $P(s'|s, a)$ define the probability of transitioning to a new state s' given the current state s and action a. The system evolves based on:

- **Energy storage changes**: Updated based on the amount of energy bought or sold.
- **Electricity price dynamics**: Future prices are unknown and provide new information at each step.
- **Time**: Time of day and day of year tie into the price fluctuation patterns and the daily storage reset.

3 Methods

Before going into detail on our choice of methods, some feature exploration is in order.

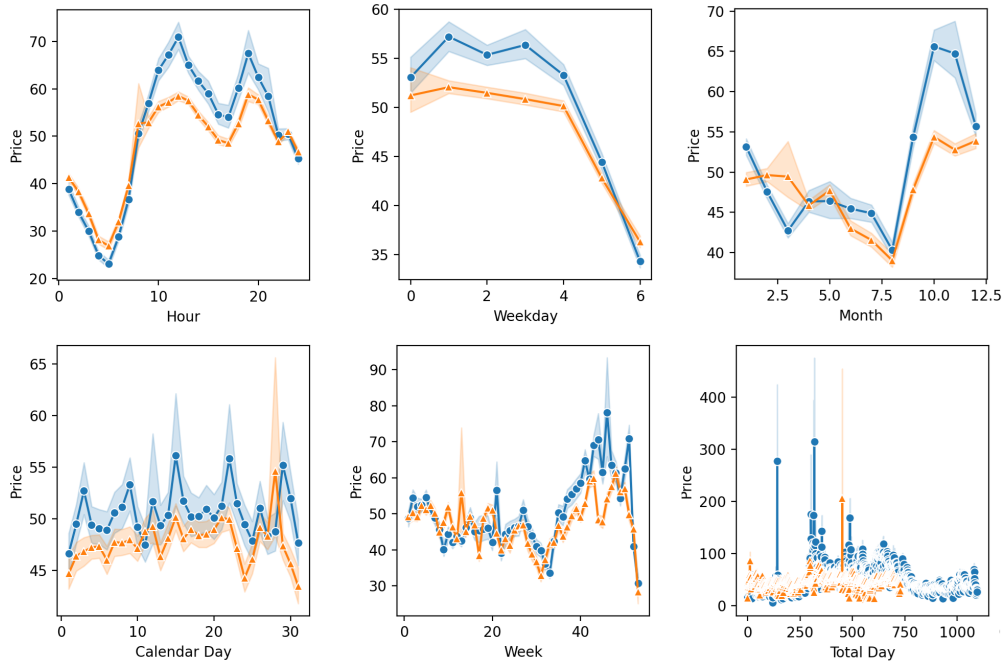


Figure 1: Price trends on different time scales. (Blue = train dataset, Orange = validate dataset)

We carried out an investigation of the potential patterns in the price fluctuations and used the extracted knowledge to inform our baselines as well as our state-space discretization and reward shaping. Price trends on both training and validation datasets can be seen in Figure 1, where we can observe the following:

- Price changes throughout the hours are quite emphasized, e.g. the morning dip, which will allow us to break the day up into a few smaller time intervals that could represent their salient trends.
- Days of the week show a sharp dip in price just as the week ends, indicating weekends will be prominent.
- The yearly price cycle is also prevalent as it jumps right when summer ends and maintains a high average through October and December.
- There are occasional spikes in certain days when the price is exceptionally high.

3.1 Baselines

3.1.1 Rule-Based Heuristic

The first three baselines candidates we employed are all hard-coded agents in the sense that they choose their actions based on explicit rules.

- **Random Agent**: Chooses each action randomly, uniform between [-1, 1] (which is multiplied by 10 for the MW equivalent)
- **Moving Average Agent**: Tracks a tuned interval of past prices to compare to the current price and choose action accordingly. This performed equally well as only buying in the night.
- **Smart Agent**: The best performing baseline followed a ruleset that involved a combination of moving average, buying in the night, and the insights extracted from the feature exploration such as the weekend and winter trends. This allowed the agent to take advantage of the weekly and seasonal patterns as well.

To provide contrast, we decided to include Random Agent and Smart Agent in our comparison study.

3.1.2 LSTM and NN

Another baseline implemented was a manually designed neural network optimized using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). The neural network consisted of two hidden layers and one output layer, all using the tanh activation function. The tanh function was chosen because the environment's action space is continuous within the range [-1,1], matching tanh's output range. Since CMA-ES is a minimization algorithm, the negative aggregated reward was returned as the optimization objective. Given that the dataset consists of temporal data, we also manually implemented LSTM formulas. This model followed a similar structure to the neural network, with a tanh-activated output layer. However, instead of simple feedforward layers, the forward pass incorporated LSTM formulas, including a hidden state that could learn long-term dependencies. The aim was to leverage sequential dependencies in the data to predict the optimal action in each state. While the neural network contained 190 weights, the LSTM required significantly more parameters, reaching 750 weights. As CMA-ES is an evolutionary algorithm, training was computationally expensive. Both models were trained for the same duration, but the LSTM completed only half the iterations of the neural network due to its higher computational cost.

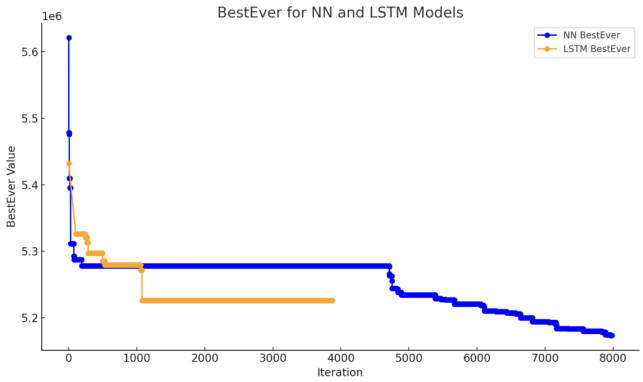


Figure 2: Best ever fitness of LSTM vs NN

In terms of performance, the neural network converged faster within the first 50 epochs, reaching a fitness plateau just above -5.3e-6 until epoch 5000, indicating difficulty in finding better solutions. Eventually, CMA-ES identified a promising direction, improving fitness to -5.15e-6, but the process was halted due to time constraints. Similarly, the LSTM plateaued at epoch 500, unexpectedly dropping to -5.21e-6 at epoch 1000, and remained stable until training was stopped at epoch 4000 due to time limitations.

In conclusion, both models performed similarly when trained for the same duration. However, considering the number of iterations, the LSTM outperformed the neural network as it achieved comparable fitness in half the iterations. This suggests that LSTM leveraged temporal dependencies to refine its predictions. However, due to its longer computation time per iteration, further training would be required to fully exploit its potential.

3.2 Tabular Q-Learning

For a detailed definition of Q-learning please see A.1 in appendix.

Q-learning is an off-policy learning algorithm, meaning it updates its Q-values using the maximum estimated future reward, irrespective of the policy used for exploration. Given sufficient exploration and an appropriate learning rate

schedule, Q-learning is guaranteed to converge to the optimal policy. However, tabular Q-learning explicitly stores Q-values for all state-action pairs, meaning in large or continuous state-action spaces, maintaining a Q-table becomes computationally infeasible. To navigate this, one must keep the number of dimensions small enough to conduct experiments. While attempting to strike this balance we tried many discretizations and feature combinations, and we present the best combination we achieved in the allotted time.

Our Q-learning implementation utilizes an ϵ-greedy exploration strategy with a decaying ϵ parameter and a decaying learning rate. This ensures initial exploration and gradual exploitation as training progresses. Learning rate decay: starts at 0.1, ends at 0.001 over 200 iterations. Convergence is typically observed once the learning rate is sufficiently small. ϵ decay: starts at 1 (pure exploration), ends at 0.001 (exploitation dominant) over 100 iterations. This schedule is sufficient given the small state space. With more features, extended training and exploration phases were necessary.

The state space is discretized into three features:

- **Hour of the day** (5 bins): $< 9, < 14, < 18, < 22, > 22$ corresponding to (morning, noon, afternoon, evening, night).
- **Storage levels** (5 bins): $< 40, < 80, < 120, < 150, > 150$.
- **Price relation to moving average** (3 bins): "below", "around", or "above" a moving weekly average. Thresholds are set by steps of size 1.5 times standard deviation of the moving interval.

Additional features, such as binary weekend (0/1) and seasonal indicators (0/1), were tested but did not improve performance and were therefore omitted.

The action space consists of three discrete actions: $-1, 0, 1$, corresponding to full sell, hold, and full buy, respectively. This aligns with the storage discretization, which is not fine-grained enough to support fractional actions.

Therefore, in total, our Q-Table consists of 225 discrete state-action pairs. Experimentation with larger actions spaces with more features or finer discretization did not improve performance.

During experimentation, initializing the Q-table with predefined values to guide search was tested. This aimed to steer the agent away from states where forced purchases were required to meet energy demands. However, this approach proved unnecessary, and for simplicity, it was discarded. Some unvisited states retain zero Q-values, but these should not be misinterpreted as optimal states.

The training procedure includes:

- Training for 300 episodes, with convergence typically reached after 200 episodes.
- Discount factor $\gamma = 0.99$, found to be optimal. Lower values significantly degraded performance, while higher values had negligible effects.
- Regular evaluation every 50 episodes on both training and validation datasets.
- Maintaining performance metrics as total yearly cost, facilitating comparison across datasets.

Reward shaping was implemented to guide learning by incorporating additional incentives such as encouraging purchases when storage is low (≤ 120) and sales when storage is high (≥ 130) using the reward shaping function defined in A.2.

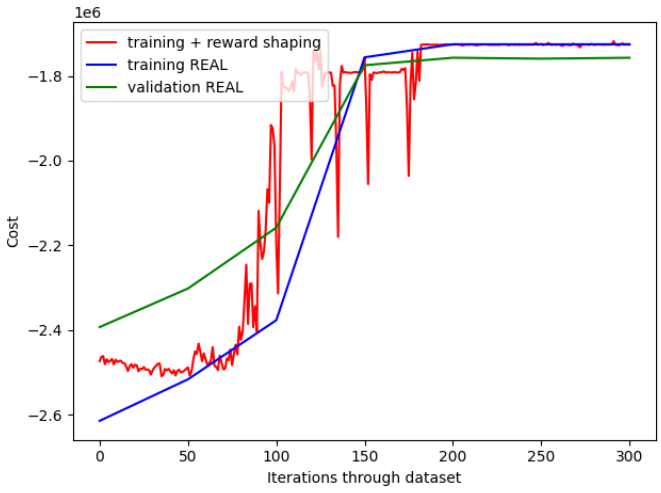


Figure 3: Actual and shaped rewards converging through iterations.

Figure 3 shows that shaped rewards and actual rewards tend to converge at around 200 iterations through the dataset.

Throughout numerous independent runs, several near-optimal agents emerged, that replicably achieved approximately 1.73 million/year cost, among which we observed distinct strategies. Some agents adopted a highly conservative strategy, minimizing sales and only buying at optimal times, while others engaged in more frequent trading with occasional suboptimal purchases. Despite these behavioral differences, the overall performance remained consistent across training and validation datasets.

In the plots here and in Section 4, we present our agent that performed the best on the 5-fold sequential cross-validation including validation dataset and that exhibited a slightly more risk-seeking strategy, achieving near-optimal performance comparable to the Smart Agent heuristic. Interestingly, agents that never sold energy also attained similar performance, but we decided to pick an agent that will showcase this approach's learning capabilities.

3.3 Double Deep Q-Learning (DDQN)

Deep Q-Learning (DQN) is an extension of Q-learning that utilizes deep neural networks to approximate the action-value function $Q(s, a)$, making it feasible for high-dimensional state spaces where tabular methods become impractical. Instead of maintaining a Q-table, a neural network with parameters θ is trained to predict Q-values, and the loss function is defined as:

$$L(\theta) = E\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

where θ^- represents the parameters of a target network that stabilizes training by being periodically updated from θ. The network is trained using mini-batch gradient descent on transitions (s, a, r, s') sampled from an experience replay buffer, which stores past experiences to break temporal correlations and improve learning stability.

DQN employs several key techniques to enhance performance and stability. **Experience Replay** stores past experiences and samples random batches to decorrelate updates. We enhanced this implementation by prioritizing the experiences with high temporal difference errors. This aims to sample more insightful experiences to learn from. **Target Network** maintains a separate network which periodically mirrors the main network, and it is tasked with estimating the target Q-values so that the estimations are locally consistent. Hence, utilizing a second network in this setting receives the name 'Double DQN'. Our **Exploration Strategy** uses ϵ-greedy exploration with a decaying ϵ to balance exploration and exploitation.

Despite its success in solving high-dimensional reinforcement learning problems, DQN can suffer from instability and inefficiency in continuous action spaces. Therefore, we opted to discretize its action space in a more granular manner, such that the agent could buy energy in multiples of 2.5 MW.

The architecture of our DDQN approach involves three hidden layers of size 256, 64, 32 respectively. We experimented with tanh but switched to ReLU for the hidden activations after facing vanishing gradients in our training. However, exploding gradients were detected after switching to ReLU, which prompted us to add gradient clipping. As for weight initializations, orthogonal initialization was found to perform better than random and Xavier initializations.

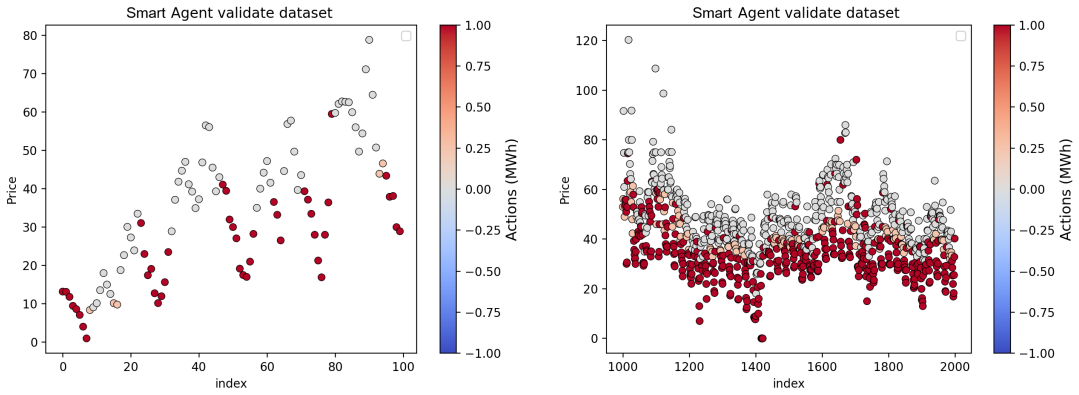
Reward shaping (See A.3 for details) has also been applied to encourage better storage management and incentivize buying as the rate and amount of buying actions were much lower than the baseline methods. We also attempted to leverage reward shaping to incentivize selling when the price is high and not selling when low. However, this had limited effect on overcoming the issue. Further experimentation with reward shaping would be required, and there is substantial room for improvement.

In its final state, the number of possible hyperparameter combinations for the DDQN were quite large (see A.4 for the full list). Here we were bound by our computational resources for tuning. For a selection of the parameters, we used trial and error to pick promising values. For the rest, we followed a manually randomized approach where the ranges for each parameter were predetermined and each run sampled its parameters from their respective range. Under time constraints, 81 different combinations were trained for 20 episodes and the best performer on validation was chosen to be presented.

4 Results

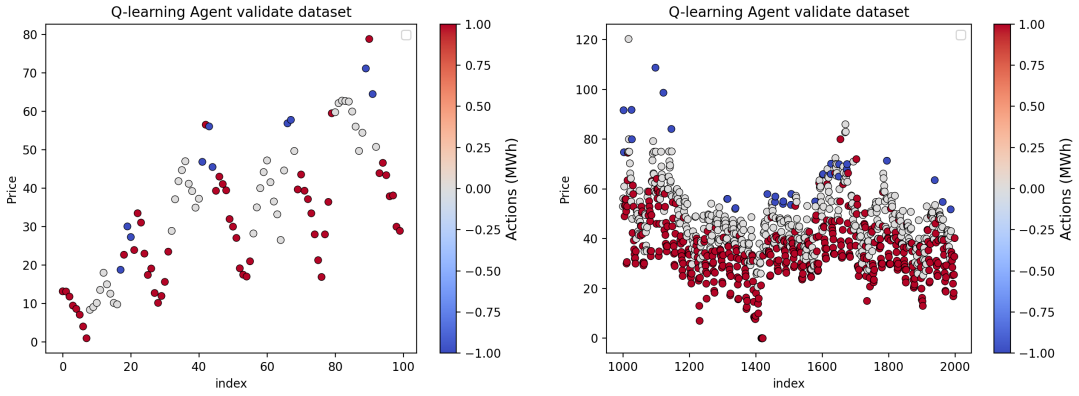
Figures 4 to 6 are 3 pairs of plots representing the actions taken by the Smart Agent, Q-learning Agent and DDQN Agent during a certain interval: 4-day interval on the left and 41-day interval on the right.

The Smart Agent's plots show that it correctly applies the rule of buying when the price is relatively cheap compared to the moving average. In the 4-day interval plot (Figure 4a), we see that the price increases over the 4 days but show regular drops, reflecing the night-time price periods in which the agent is consistently buying. On the contrary, when the price is above the moving average, the agent's actions becomes grey which means it does not buy nor sell. When we look at the 41-day interval (Figure 4b), we clearly see that most of the red actions are located when the price is below the moving average and grey actions when the price is above the moving average.



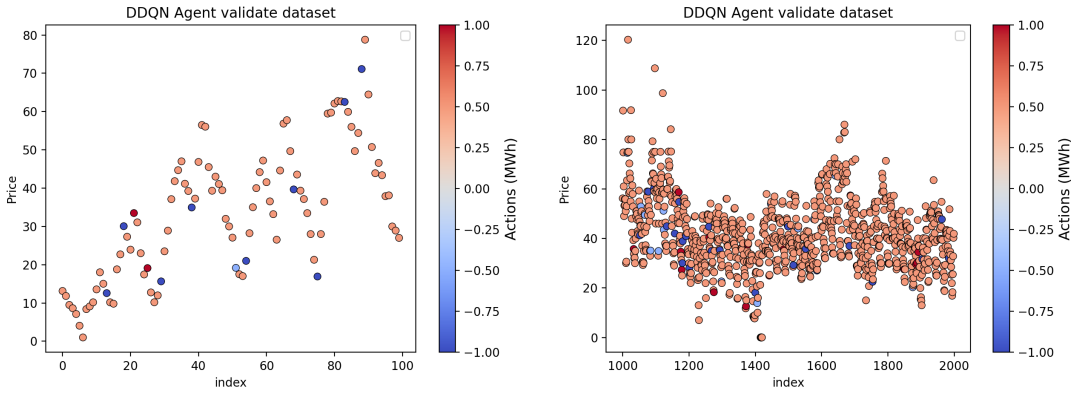
(a) 4-day interval (b) 41-day interval

Figure 4: Actions of the Smart agent through 100 steps (left) and 1000 steps (right). Red = Buy, Blue = Sell



(a) 4-day interval (b) 41-day interval

Figure 5: Actions of the Q agent through 100 steps (left) and 1000 steps (right). Red = Buy, Blue = Sell



(a) 4-day interval (b) 41-day interval

Figure 6: Actions of the DDQN agent through 100 steps (left) and 1000 steps (right). Red = Buy, Blue = Sell

Next we have the Q-learning Agent (Figure 5). We can see a similar behavior as we've seen for the Smart Agent except that now the Q-learning Agent also sells when it detects that the price increases (blue points). In the same 4-day interval, we see blue points when the prices increase; However, the agent sometimes makes the mistake of buying when the price is actually above the moving average visible at index 40 and 90 while the Smart Agent would hold. Those types of errors revealed themselves to be somewhat costly as it affected the overall reward. The Q-learning Agent, while close, couldn't outperform the Smart Agent's overall reward. In the 41-day interval, we see similar trends as in the Smart Agent except the sell actions where most of the time occur while the price is extremely above average. We also notice that this agent buys most of the time which seems to be the most rewarded policy.

Figure 6 concerns the DDQN Agent. Most of the chosen actions were 0.25 which in contrary to the two previous agents, DDQN performs a partial buy at each time step. The bias towards partial buying could be resolved via further reward shaping. Also, this model had a sub-optimal hyper-parameter tuning process due to the limited resources and it reflects on this plot as we see that the agent sells when the price is relatively low and sometimes buys at what seems to be random price points.

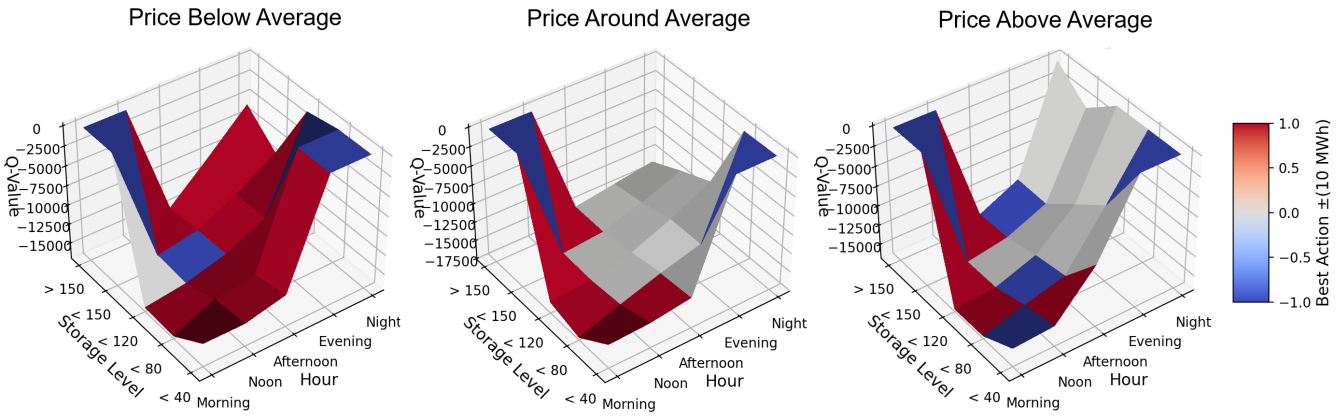


Figure 7: Q-values plotted in 3D when the price is below, around, and above the weekly moving average (left to right).

The 3D figures above reflects the 5-dimensional Q-table of our most promising RL agent (Tabular Q-learning Agent) and its Q-values in relation to the discretized storage level, time of the day, price level and actions. The left-most plot illustrates when the price is below the moving average, where we can clearly see that the the optimal action most of the time is to buy, which makes sense since the energy is cheaper than usual. In the central plot, we see that at average prices, the agent only buys at early times of day and avoids actions at other times. This overly conservative behavior likely hinders performance. In the right-most plot, we see that the buying behavior is the same at above-average prices, except that the agent will also try to sell sometimes if there is surplus energy or enough time left in the day. Overall, the agent correctly learned to display more liberal buying behavior when energy prices are cheap, and more conservative buying behavior once prices got more expensive, although the buying behavior was quite limited. This price-sensitive behavior emerged purely from experience and was not hard-coded or reward-shaped in any way, which shows that the agent really learned. The ever-blue corners in the plots represent the unvisited cells in the Q-table (impossible states, Q-value 0 like at initialization) and they should be disregarded.

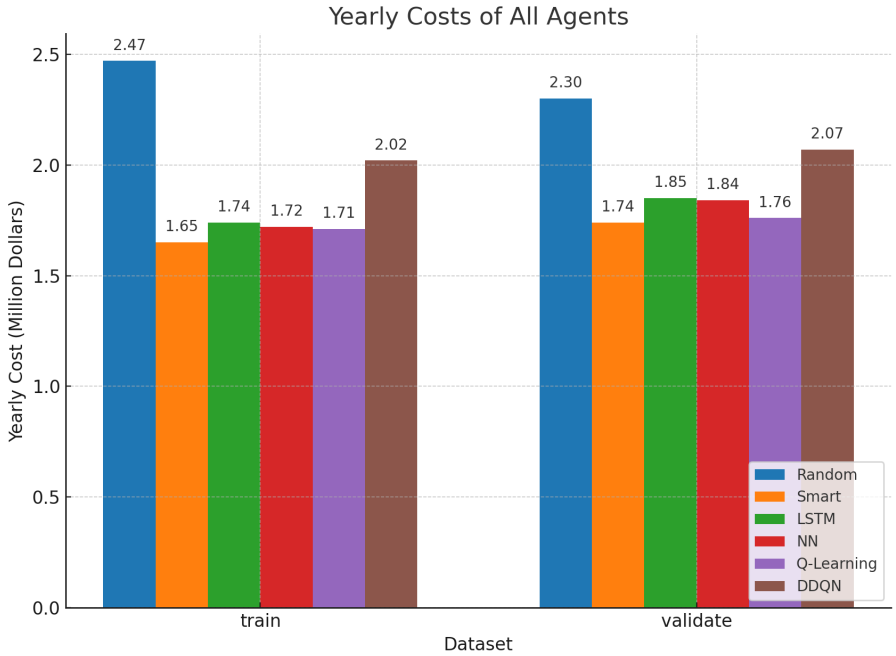


Figure 8: Average costs per year incurred by the best performing agent of each approach.

The bar plot above reflects all of our model's performance on the train and validate set. Every model has better results than the random agent. And all were around the -1.7e-6 yearly reward mark except the DDQN which would require further fine tuning and reward shaping in order to obtain significantly better results.

5 Discussion

Regarding our Tabular Q-learning agent, we tried many different training parameters, discretizations and reward shaping strategies and failed to achieve better performance than with our presented agent. A systematic grid-search or Bayesian optimization approach to hyper-parameter tuning could yield better results. While we also experimented with different features, and did not see improvements, perhaps an even more intensive investigation and feature engineering could place the agent in a state space more conducive to learning.

Since the dataset is of temporal nature, the linear layers that composed the network in our DDQN agent could have been replaced with LSTM layers (implemented as one of our baselines). This could have potentially helped the model learn dependencies within the data and boost its performance. Some changes would need to be made to preserve the temporal nature of the data, such as modifying the replay buffer so that it provides sequential experience instead of random samples.

A potential change in the CMA baseline models could be adding binned actions: instead of applying tanh on a single output, apply softmax activation on e.g. 3 outputs corresponding to the actions -1, 0, 1 and train the model to predict the probabilities of those actions given a state to receive a higher reward.

Due to the time constraints, the DDQN wasn't fine-tuned to the maximum. States (input) normalization was done and some reward normalization was achieved however in order to obtain a more robust model, those steps need to be re-visited and carefully re-designed. Reward shaping was also incorporated and showed some promise but extensive trials are needed to achieve an agent with good convergence. There was also a problem in the gradient descent optimizers where the gradient norm would become either extremely small (vanishing gradients), in response we changed the tanh activation to ReLU activation function after which we applied gradient clipping. Unfortunately, the training was still slow.

6 Conclusion

To conclude, under the time constraint our best agent ended up being the Smart Agent which was able to obtain the best score on the train and validation datasets. Followed closely by our Q-learning Agent, our winning RL approach in this study. We note, however, that the Q-learning algorithm produced agents with the most robust performance, nearly equal between the train and validate dataset. Combined with the fact that the Q-Learning agent also displayed selling behavior, we hypothesize that it would be the agent to use for unseen datasets.

Under similar computation and time constraints, DDQN proved to be the worst approach by its resource-intensive and complex nature. Rule-based approach was by far the most 'economical' in terms of resource per score. And although Q-learning agent was achieving similar performance, it could be further engineered to surpass the heuristic using relatively low resources.

Our Reinforcement Learning methods showed promise for this energy cost optimization problem but it requires careful tuning, while simple rule-based heuristics can often outperform RL when domain knowledge is applied effectively. For a real-world implementation for this problem, an hybrid approach combining heuristics (domain knowledge) and reinforcement learning would likely yield best results.

A Appendix

A.1 Q-Learning Definition

Q-learning is a model-free reinforcement learning algorithm that enables an agent to learn an optimal action-selection policy by interacting with an environment. It is based on estimating action-value functions, which represent the expected cumulative reward when taking a specific action in a given state and subsequently following the optimal policy. The action-value function, denoted as $Q(s, a)$, is iteratively updated to approximate the optimal action-value function $Q^*(s, a)$.

The optimal Q-function satisfies the Bellman equation:

$$Q^*(s, a) = E\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right]$$

where s and s' represent the current and next states, respectively, a and a' denote the current and next actions, r is the immediate reward received, and $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards. The update rule for Q-learning is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$$

where α is the learning rate, controlling how much new information influences the existing Q-values. The Q-learning algorithm follows an iterative procedure: (1) initialize the Q-table with arbitrary values, (2) observe the current state s, (3) select an action a using an exploration strategy such as ϵ-greedy, (4) execute a, observe the reward r and next state s', (5) update $Q(s, a)$ using the Q-learning rule, and (6) repeat the process until convergence.

A.2 Reward Shaping in Tabular Q-Learning

$$\text{if } s_{\text{storage}} \leq 120: \quad \begin{cases} \text{Buy } (a = 1) & +\frac{1}{3}|r| \\ \text{Sell } (a = -1) & -\frac{1}{2}|r| \\ \text{Hold } (a = 0) & -\frac{1}{5}|r| \end{cases}$$

$$\text{if } s_{\text{storage}} \geq 130: \quad \begin{cases} \text{Buy } (a = 1) & -\frac{1}{2}|r| \\ \text{Sell } (a = -1) & +\frac{1}{3}|r| \\ \text{Hold } (a = 0) & -\frac{1}{3}|r| \end{cases}$$

This reward shaping term (RS) is integrated into the Temporal Difference (TD) Q-value update:

$$\begin{aligned} \text{Target} &= r + RS + \gamma \max_{a'} Q(s', a') \\ \text{TD Error} &= \text{Target} - Q(s, a) \\ Q(s, a) &\leftarrow Q(s, a) + \alpha \cdot \text{TD Error} \end{aligned}$$

A.3 Reward Shaping in DDQN

To guide the learning process of the DDQN agent, a shaped reward function is applied to encourage desirable behaviors. The reward function is defined as follows:

- If the hour reaches 24, the reward is penalized based on the remaining storage level:

$$r \leftarrow r - a \cdot \text{storage}$$

- If the storage depletion rate is too high and the action is to buy or hold ($a \geq 0$), a large penalty is applied:

$$\text{if } \frac{\text{storage} - 50}{24 - \text{hour}} > 10 \text{ and } a \geq 0, \quad r \leftarrow r - 100x$$

- Conversely, if the storage depletion rate is too high and the agent chooses to sell ($a < 0$), a large reward is given:

$$\text{if } \frac{\text{storage} - 50}{24 - \text{hour}} > 10 \text{ and } a < 0, \quad r \leftarrow r + 100x$$

- If the price is high ($p > 0.4$) and the agent chooses to sell ($a < 0$), an additional reward is provided:

$$\text{if } p > 0.4 \text{ and } a < 0, \quad r \leftarrow r + 100x \cdot p$$

- A small penalty is applied for taking any selling action:

$$\text{if } a < 0, \quad r \leftarrow r - |a|x$$

- A small reward is provided for taking any buying action:

$$\text{if } a \geq 0, \quad r \leftarrow r + |a|x$$

Here, x is a tunable scaling factor that determines the strength of the shaping rewards and penalties. This function encourages the agent to avoid excessive depletion of storage while optimizing for higher prices when selling energy.

A.4 DDQN Parameter List

Parameter	**Value**
Hidden layers	[256, 64, 32] neurons
Activation function	ReLU
Weight initialization	Orthogonal initialization
Discount rate (γ)	0.9164
Batch size	32
Replay buffer size	300,000
Learning rate	0.000235
Target network update frequency	256 steps
Epsilon start	0.191
Epsilon end	0.01
Epsilon decay	2238.61
Alpha (prioritization exponent)	0.9
Beta (importance sampling exponent)	0.8
Minimum replay size	3,000
Shaped reward weight	0.519
Random seed	33
Gradient clipping norm	1.0