STOCHASTIC OPTIMIZATION AND LEARNING

Assignment 2 Optimal Learning

Group 10

Alice Lonza (S3514641)

Serena Lattanzi (S3515338)

04-06-2025

Use of Artificial Intelligence Statement

During the preparation of this work, we used ChatGPT for text enhancement, and GitHub Copilot to investigate the reasons behind the long algorithm running time of a specific code segment: the generation of sampled values with the truncated normal distribution. The function chosen turned out to be computationally very expensive and we were calling it several times. After using these tools, we thoroughly reviewed and edited the content as needed, taking full responsibility for the final outcome.

Contribution Statement

Both of us worked together on understanding and mathematically formulating the problems, analyzing the results and drawing the conclusions about them. Overall, we believe the time burden of this project was shared 50-50 between both teammates.

1. Evaluating one alternative

In order to estimate the expected daily profit of the energy storage system managed by a given rule with the set parameters $\gamma_1 = 23$ and $\gamma_2 = 26$, we simulated the decision model over 1,000,000 days of solar generation, load demand, and market prices. These variables were sampled from truncated normal distributions based on the given means and variances, using standardized bounds via the Python function *scipy.stats.truncnorm*.

To simulate the decision model over all the days, we tested three approaches with progressively improved performance. Initially, we generated random values for each variable within every time step of every day, leading to an estimated runtime of over 12 hours. The reason behind this is that the chosen function, *scipy.stats.truncnorm*, is very expensive computationally, and we were calling it several times. We improved the performance by generating arrays of 97 values per variable once per day, resulting in a runtime of approximately 8 hours. Finally, we decided to create a matrix of all required values for every day and time step in a separate function, *samples_matrix*, at the beginning of the simulation. This way, the function is called only once instead of during every iteration, reducing the total runtime to just 4 minutes.

To make the simulation reproducible, we used a random number generator with a fixed seed (*default_rng(seed=1) from NumPy*). This generator was used every time we created random values for generation, load, and price.

For each simulated day, the *simulate_one_day* function calculates the energy flows by distributing energy between solar generation, battery storage, load consumption, and market interactions, based on the decision rule applied every 15-minute interval. The function respects the battery constraints (such as capacity and maximum charging and discharging rates) and calculates the profit across 97 time steps. The total profit for each day is the sum of the profits over all the intervals.

Finally, the function *simulate_many_days* runs the one-day simulation for N independent days, collects all the profits, and computes the mean expected daily profit along with the sample variance.

After running the code, we obtained the following results:

Table 1: Results of simulating 1,000,000 days.

Expected daily profit	5,108.9632
Sample variance of profit	1,236,930.5686

The result of the expected daily profit is approximately 5,109 euros, indicating that the system can generate positive returns on average under the given decision rule parameters.

However, the sample variance of the daily profit is quite large, about 1,236,931. It indicates that daily profits fluctuate due to the uncertainties in solar generation, load demand, and market prices, which are susceptible to sudden changes.

2. Offline Learning: Searching for the best alternative

To identify the best decision rule, we compared 27 alternatives defined as (γ_1, γ_2), where $\gamma_1 \in \{22,23,24,25,26\}$, $\gamma_2 \in \{25,26,27,28,29,30\}$, and $\gamma_1 < \gamma_2$. We estimated the true expected profit and variance for each alternative by simulating the system over 1,000,000 days for each (γ_1, γ_2) pair, using the simulation model that we created in question 2.1 to compute the daily profits under each rule. Generation, load, and price variables were pre-sampled using truncated normal distributions based on the given average profiles and stored in MxNxT matrices. To ensure a reproducible comparison between policies, we used a fixed random seed (*default_rng(seed=1)*), as in the previous problem. Table 2 shows the results, which were stored in a CSV file (*true_qualities*):

Table 2: Expected profit and variance for each alternative. The highlighted alternative is the best found.

γ_1	γ_2	mean	variance
22	25	4085.646508	782575.694
22	26	4423.213298	949978.539
22	27	4863.297824	1165279.347
22	28	5307.363365	1369630.708
22	29	5543.334544	1568524.791
22	30	5563.151514	1748020.662
23	25	4687.518576	1040115.665
23	26	5108.963179	1236930.569
23	27	5538.141950	1437236.005
23	28	5749.803371	1646595.089
23	29	5710.581439	1827831.950
23	30	5592.671331	1908531.447
24	25	5315.538995	1300789.964
24	26	5708.391359	1478489.783
24	27	5904.366185	1676714.245
24	28	5840.423009	1842323.765
24	29	5682.914122	1907884.445
24	30	5541.937544	1925566.264
25	26	5985.269349	1675605.010
25	27	5945.624732	1825515.680
25	28	5784.336745	1888218.980
25	29	5617.801348	1905207.305
25	30	5484.649545	1910752.580
26	27	5874.478257	1861785.346
26	28	5709.089569	1882835.950
26	29	5555.368711	1890564.973
26	30	5432.949506	1893950.193

These results represent the true qualities of each decision rule. As can be seen on the table, the best alternative is (γ_1, γ_2) = (25, 26), which gave a mean profit of $5985. We then used these good

estimates of the true qualities as reference values for the offline learning simulation. Our aim was to evaluate how effectively each policy learns to identify high-quality alternatives over time, through 100 experiments of 500 days each. For doing that, we applied 4 different sampling policies that selected which alternative to evaluate at each iteration:

1. Exploration

It selects randomly one of the alternatives every day, regardless of the current beliefs, so it ensures that all alternatives are sampled equally, but without exploiting any knowledge about which is the best. The code used for this policy was:

```
choice = np.random.randint(K)
```

2. Exploitation

It always selects the alternative with the highest estimated profit (mean μ) at the current iteration, so it maximizes reward on current beliefs, but it can miss some better alternatives if the initial estimates are poor. It is possible that multiple alternatives share the highest mean, so to avoid making a biased choice by always selecting the first alternative in the array with the highest mean, we designed the algorithm to choose randomly among the best options. The code used for this policy was:

```
max_μ = np.max(μ)
candidates = np.flatnonzero(μ == max_μ)  # Indices of alternatives with max mean
choice = rng.choice(candidates)
```

3. ε-greedy

It balances exploration and exploitation by selecting a random alternative with a probability that decreases over time. At iteration n, the policy chooses a random alternative with probability $\varepsilon_n = \frac{0.95}{n+1}$, and otherwise selects the alternative with the highest current estimates mean. This approach allows for more exploration in the first iterations and gradually shifts toward exploitation. The code used for this policy was:

```
c = 0.95
ε = c / (n + 1)  # +1 to avoid dividing by zero
if rng.random() < ε:
  choice = rng.integers(K)  # Explore
else:
  max_μ = np.max(μ)
  candidates = np.flatnonzero(μ == max_μ) # Exploit
  choice = rng.choice(candidates)
```

4. Knowledge gradient (KG)

At each day, this policy selects the alternative that is expected to provide the greatest improvement in decision quality after one additional observation. For each alternative, a Knowledge Gradient (KG) is computed based on the difference between its current estimated profit and the next best alternative, adjusted by how uncertain we are about the estimate. Then, the algorithm selects the alternative with the highest KG value.
This policy balances exploration and exploitation by considering both current reward and the value of learning. The code used for this policy was:

```
β = 1 / var
β_w = 1 / var_w
var_tilde = (1 / β) - (1 / (β + β_w))
σ_tilde = np.sqrt(var_tilde)

μ_matrix = np.tile(μ, (K, 1))  # Matrix of means
np.fill_diagonal(μ_matrix, -np.inf) # Fill diagonal with -inf to ignore self-comparison
 μ_star = μ_matrix.max(axis=1)  # vector of max for j ≠ i

ζ = -np.abs((μ - μ_star) / σ_tilde)
f_ζ = ζ * norm.cdf(ζ) + norm.pdf(ζ)
ν_kg = σ_tilde * f_ζ

candidates = np.flatnonzero(ν_kg == np.max(ν_kg))
choice = rng.choice(candidates) # Randomly select if multiple candidates have the same score
```

Once the choice is made, the selected alternative is used to simulate one day of operation, returning the corresponding daily profit. Then, the belief about this alternative is updated using a Bayesian update: the mean μ is updated as a weighted average between the previous estimate and the observed profit, with weights based on their variances.

$$\mu_{new} = \frac{\mu_{old} * \sigma_w^2 + profit * \sigma_{old}^2}{\sigma_{old}^2 + \sigma_w^2}$$

The variance is updated to reflect the reduced uncertainty after gaining new information, so the more we sample an alternative, the more confident we become about its true value.

$$\sigma_{new}^2 = \frac{\sigma_{old}^2 * \sigma_w^2}{\sigma_{old}^2 + \sigma_w^2}$$

After updating the beliefs, the algorithm identifies the alternative with the highest current estimated mean and records its true profit from the table (*true_qualities*). This value represents the true quality of the alternative that the model believed to be best at that point in time. We stored these values at each iteration and for each experiment. At the end, we computed the average across all 100

experiments, resulting in a learning curve over 500 days. Figure 1 illustrates how quickly each policy converges to high-performing alternatives.

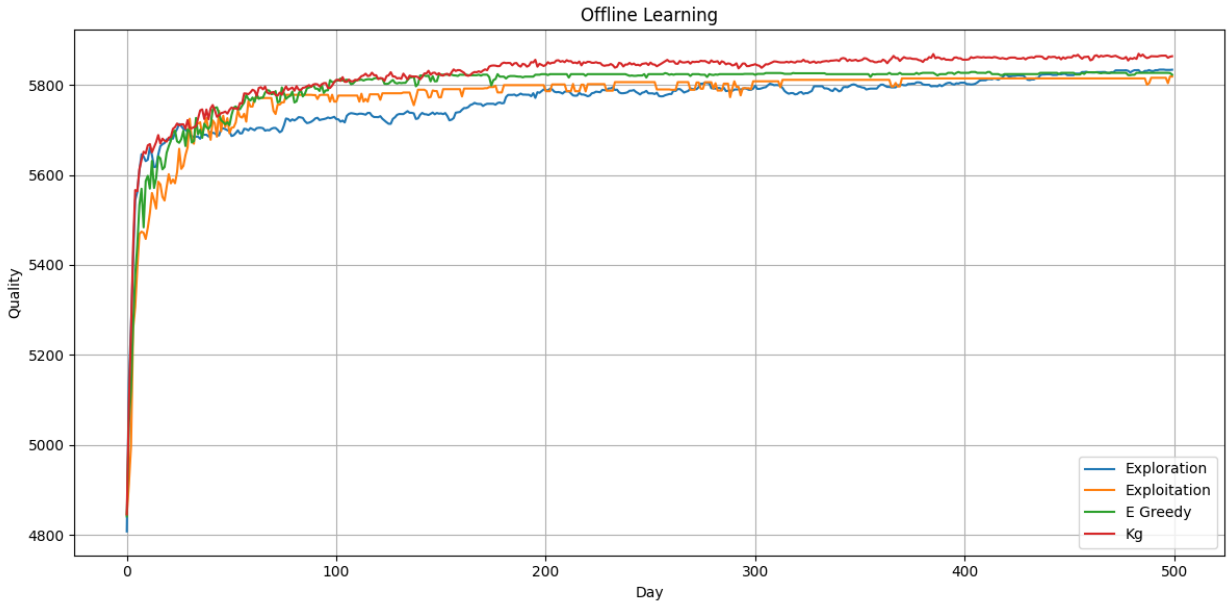

Fig. 1: Offline results of policies simulation. Created on Visual Studio Code

We can see that during the first iterations, all the policies behave similarly, learning very fast, but they grow differently as the days pass.

As expected, the knowledge gradient policy achieved the best performance overall: it converges quickly towards high-quality alternatives and remains the best throughout the days.

The initial exploration allows the ε-greedy policy to avoid local optima, and its gradual shift toward exploitation ensures the convergence. However, its performance stabilizes earlier and converges to a lower final quality compared to pure exploration. We were surprised to see this result in the last iterations and believe this is due to the early exploration of ε-greedy being unlucky, so it commits to suboptimal choices when turning into exploitation.

The exploration policy is slower to improve initially, but by sampling all alternatives uniformly, it discovers high-quality options that ε-greedy may overlook as it begins to exploit too soon.

Finally, the exploitation improves rapidly at the start but quickly stabilizes, due to its lack of exploration. It prevents further improvement because it likely gets stuck in suboptimal alternatives.

Next, because running time is also an important feature of optimization learning, we have compared the running time that each policy requires to run the 100 experiments of 500 days each. The results can be seen in Table 3.

Table 3: Running time of each policy.

Policy	Run time (s)
Exploration	12
Exploitation	13
ε-Greedy	13
Knowledge Gradient	21

It is important to note that the KG policy not only outperforms the other strategies in terms of learning effectiveness but also keeps a reasonable computational time. While it is almost double the time of the rest, it remains within the same order of magnitude. This makes it a very powerful and practical approach.

3. Online Learning: Optimizing profits while operating the system

In contrast with the offline case, where we could evaluate all alternatives every day and choose the one that looked the best, in the online scenario, we assumed that we didn't know the true quality of each option in advance. We had to choose one alternative each day, measure the profit, and update our belief about how good it is.

Just like in the offline setting, we simulated 100 experiments of 500 days each, using the same random seeds, truncated normal distributions, and initial values. The main difference is how we evaluated the performance: in the online case, we tracked the quality of the alternatives selected each day, whereas in the offline case, we recorded the true profit of the best belief, even if it hadn't been sampled yet.

The structure of the policy calculations is the same as in the offline case for exploration, exploitation, and ε-greedy; however, the Knowledge Gradient (KG) policy is adapted, since here we multiplied the KG bonus by the number of remaining days (N−n), giving more importance to learning early on. This is because new observations are more useful if they happen early, when there is still time to use what we learn. By multiplying the KG score by the number of days left, we give more importance to exploration at the beginning and focus more on good options later. As time goes on, the value of exploration decreases, and the KG policy naturally focuses more on exploitation.

The updated code used for this policy was:

```
β = 1 / var
β_w = 1 / var_w
var_tilde = (1 / β) - (1 / (β + β_w))
σ_tilde = np.sqrt(var_tilde)

μ_matrix = np.tile(μ, (K, 1))  # Matrix of means
np.fill_diagonal(μ_matrix, -np.inf) # Fill diagonal with -inf to ignore self-comparison
μ_star = μ_matrix.max(axis=1)  # vector of max for j ≠ i
```

ζ = -np.abs((μ - μ_star) / σ_tilde)
*f_ζ = ζ * norm.cdf(ζ) + norm.pdf(ζ)*
*ν_kg = σ_tilde * f_ζ*

*score = μ + (N - n) * ν_kg*
candidates = np.flatnonzero(score == np.max(score))
choice = rng.choice(candidates)

Figure 2 illustrates the results after the online policies simulation.

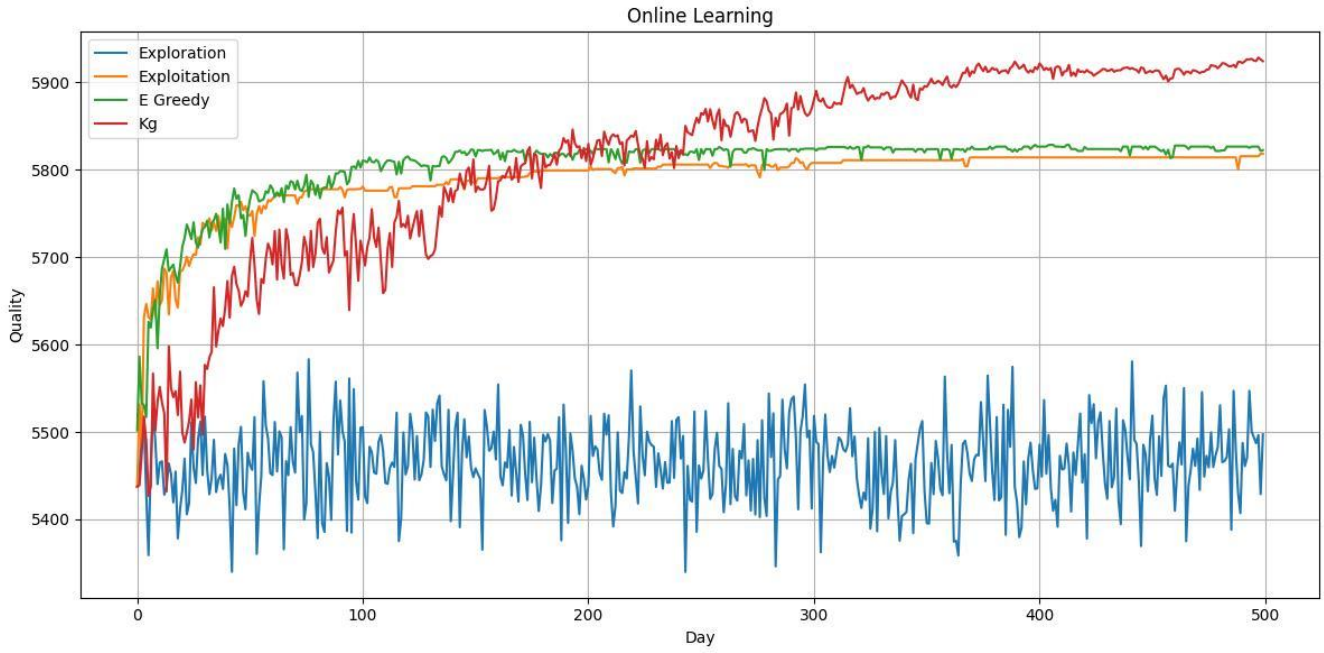

Fig. 2: Online results of policies simulation. Created on Visual Studio Code

In the online setting, the iterations start from the initial beliefs instead of from the true qualities. The biggest difference compared to the offline tests is in the exploration policy. In this case, the line has a high variance and does not show improvements. While useful for offline learning, exploration is highly inefficient in online settings, where actions directly affect rewards.

The behavior of the exploitation and ε-greedy policies aligns with what we expected, and they are also similar to the offline computations, having an initial fast improvement and converging quickly to values around 5800. In this regard, both policies are useful for offline and online settings. In the case of ε-greedy, the initial exploration allows to avoid local optima, but it stabilizes quickly. Since the exploitation policy does not explore at all, it locks into suboptimal alternatives. It performs better than exploration but lower than the adaptive policies.

Next, for the Knowledge Gradient policy we observe initial variability but consistent growth. Again, it outperforms the other policies, but now by a greater difference than in the previous experiment, which makes it a good approach for online learning. We also saw that the line was not stabilized yet

after 500 days, which made us curious about how much it could improve the profit over a longer horizon. For this reason, we decided to run the program for 1000 days. Figure 3 shows these results.

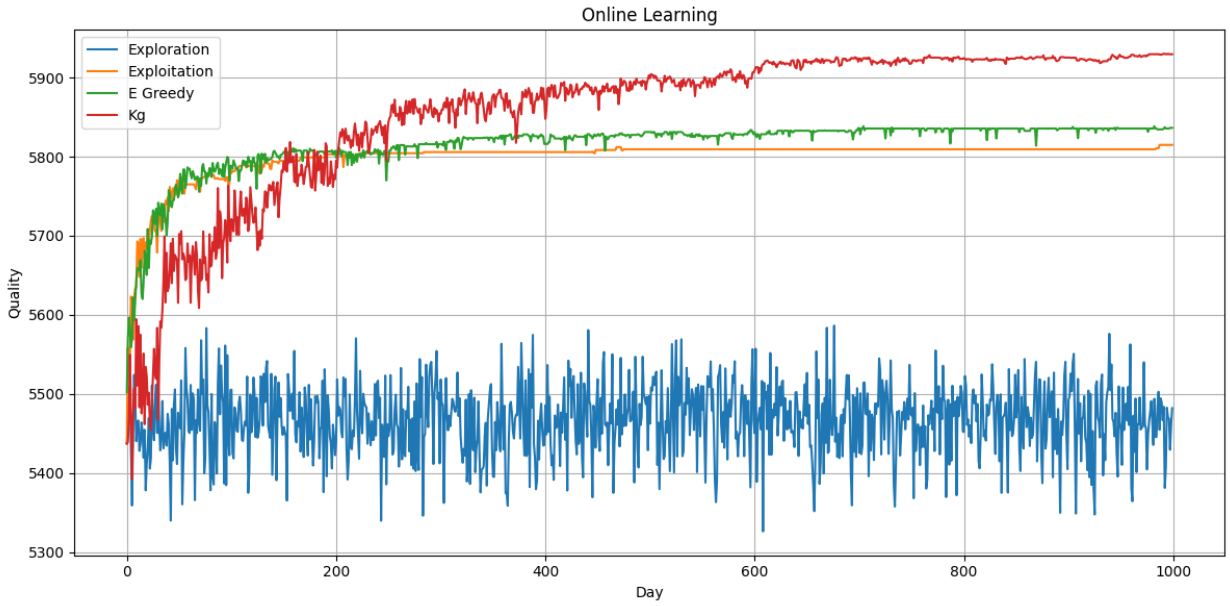

Fig. 3: Online Learning experiment: extended horizon

We can see that the quality continues to improve slightly after day 500, stabilizing around day 600. Although performance increases, the additional gains are limited. This confirms that running the simulation for 500 days captures the majority of the learning and provides a solid basis for evaluating the policy.

Finally, we also compared the computational time of each policy. As observed in the offline setting, the KG policy requires more time to run. However, in the online setting, the difference in execution time is smaller, further highlighting its effectiveness. The results can be seen in Table 4.

Table 4: Running time of the online policies

Policy	Run time (s)
Exploration	16
Exploitation	17
ε-Greedy	17
Knowledge Gradient	26