University of Verona
Department of Computer Science
Master's degree in Computer Science and Engineering

A Reinforcement Learning approach to strategy optimization for Battery Energy Storage Systems

Supervisor:
Professor
Alberto Castellini, PhD

Co-Supervisor:
Professor
Alessandro Farinelli, PhD

Candidate:
Nicolò Squarzoni
VR465317

ACADEMIC YEAR 2023/2024

Contents

Chapter 1

Introduction

The Battery Energy Storage Systems (BESS) market, valued at $7.31 billion in 2022, is set for substantial growth. Projections indicate an increase from $8.95 billion in 2023 to $69.77 billion by 2032, with a compound annual growth rate (CAGR) of 25.62% during this period.

Global demand for BESS is rising as various sectors recognize the importance of reliable energy supply. These systems offer two key benefits: optimizing energy costs by managing peak electricity demand and providing crucial backup power during outages. This dual functionality enhances both cost-effectiveness and operational resilience, particularly for manufacturing facilities.

The market is categorized into three segments based on battery capacity: low, medium, and high-scale systems. High-scale BESS are particularly vital for large utility projects, grid-level applications, and major industrial complexes. They play a critical role in grid stabilization, large-scale renewable energy integration, and maintaining overall grid reliability. These systems serve as cornerstone solutions for significant energy infrastructure, supporting the widespread adoption of renewable energy and ensuring a robust, stable power grid.

This project focuses on high-scale grid connected BESS, deployed to increase grid stability and to reduce the energy cost for the communities. These types of systems require considerable investments, therefore it is necessary to always adopt the best possible operation strategy in order to guarantee a positive return of investment. The goal of the project is then to investigate the effectiveness of the state-of-the-art reinforcement learning algorithms

in optimizing the BESS operations, trying to maximize the return of the investment.

Chapter 2

The environment: BESS in the energy market

There are several and very precise regulations about how a BESS and other market participants can interact with the grid, and they are defined by the Transmission System Operator (TSO), the organization responsible for managing all the aspects related to energy generation and transmission.

Despite similar, every TSO has its own regulation and naming. For the project, it was decided to analyze and base the whole work on the NYISO regulations, the TSO of the state of New York in USA, because the majority of data necessary for the project are available on the NYISO website.

The NYISO market comprises of a number of closely linked processes to ensure efficiency and promote reliable operation. The major processes, illustrated in Figure 2.1, include:

- annual Installed Capacity requirement/market assessment
- week-ahead reliability reviews
- Day-Ahead Market (DAM) to define a detailed energy flow for the following day based on load forecast and generation availability
- Real-Time Commitment (RTC) to define a detailed energy flow for the next 15 minutes and secure the commitment of the generators that offered their energy
- Real-Time Dispatch (RTD) to communicate to the generators to produce the energy they committed for

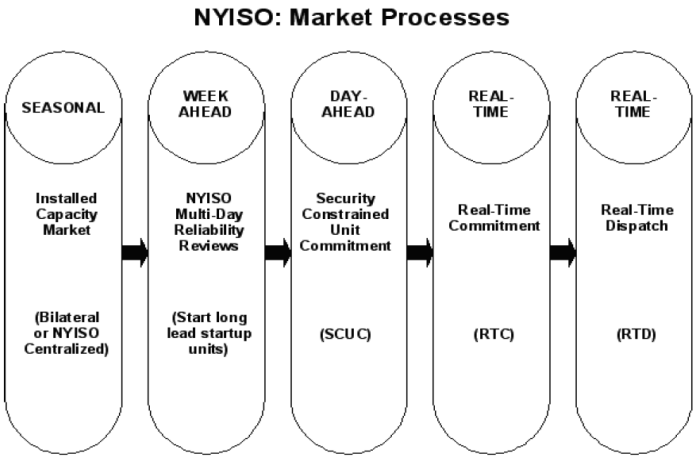


Figure 2.1: Figure 1: NYISO market process from NYISO Market Participants User Guide [16]

The way a BESS, as a market participant, interacts with the system is by means of two levels of bid: day ahead and real time.

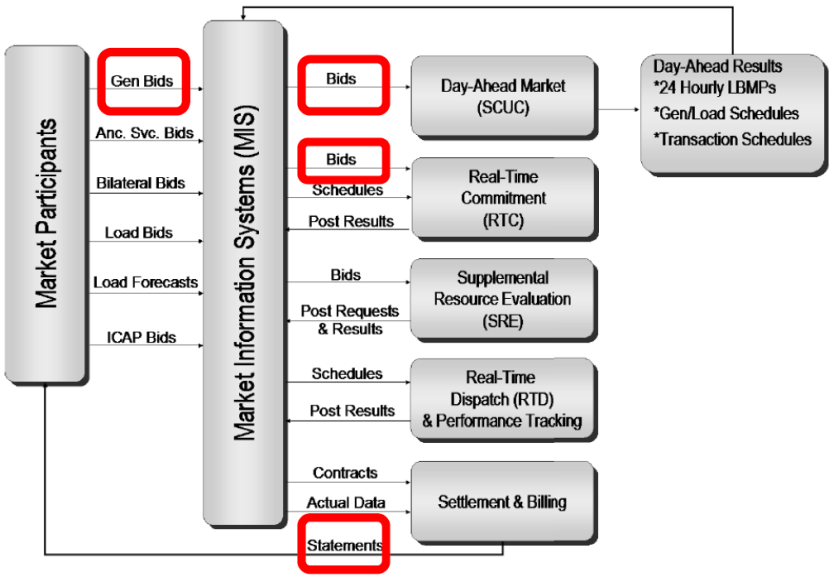


Figure 2.2: Figure 2: NYISO bidding process from NYISO Market Participants User Guide [16]

As shown in Figure 2.2 the Market Information System (MIS) processes the

bids together with other data and communicates to the market participants the accepted and declined bids, together with the real time dispatch which represents the request to produce the agreed energy. Figure 2.3 shows the timing details of both type of bids, processing and final dispatch.

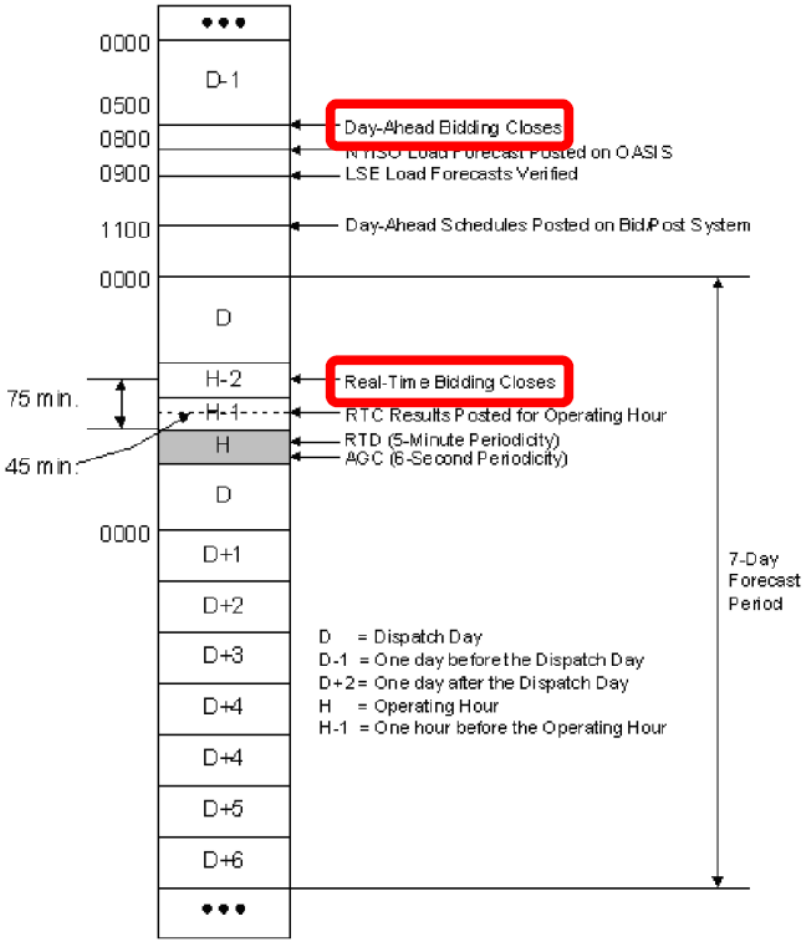


Figure 2.3: Figure 3: bidding – dispatch process timing from NYISO Market Participants User Guide [16]

It can be noticed that it is possible to submit Day Ahead bids until 5 am of the day before, and Real Time bids until 75 minutes before the operating hour. These are very critical constraints to consider during the design of the software that will submit the bids for the BESS.

Another important aspect to keep in consideration specifically for the BESS

is the diversity of services that such system can offer to the grid other than the standard wholesale (buy and sell), which include 10 minutes Spinning Reserve, 10 minutes Non-Synchronous Reserve, 30 minutes Operating Reserve, Regulation Capacity, Regulation Movement. They are called ancillary services, and for each service the price paid by the TSO is different and this represents a consistent complication of the operating environment.

Chapter 3

The Goal

Now that it is clear how the BESS interacts with the TSO it is possible to have a clearer definition of the project goal, which is the development of a software system that, based on the environment conditions, is able to suggest the best day ahead and real time bids in order to maximize the BESS return of investment. All this using state-of-the-art reinforcement learning algorithms.

Chapter 4

Related Work

Machine Learning is a subset of computer science, data science, and artificial intelligence that enables systems to learn and improve from data without explicit programming. Machine Learning models use algorithms and statistical models to perform tasks based on data patterns and inferences, continuously updating outputs as new data becomes available.

Machine learning algorithms fall into five broad categories: supervised learning, unsupervised learning, semi-supervised learning, self-supervised and reinforcement learning.

Supervised Learning: it is a type of machine learning where the model is trained on a labeled dataset. In this approach, the algorithm learns to map input data to known output labels.

Unsupervised Learning: these algorithms work with unlabeled data, attempting to find patterns or structures within the data without predefined categories.

Self-Supervised Learning: it is a subset of unsupervised learning where the model generates its own labels from the data.

Semi-Supervised Learning: it combines elements of both supervised and unsupervised learning, using a small amount of labeled data and a large amount of unlabeled data.

Reinforcement Learning: it represents a significant milestone in the field of artificial intelligence, with a rich history spanning over seven decades of rigorous academic pursuit. At its core, Reinforcement Learning is a sophisticated machine learning paradigm that enables algorithms to make informed decisions within a given environment. The learning process is fundamentally rooted in trial and error, allowing the algorithm to refine its decision-making capabilities through repeated interactions with its surroundings.

In the contemporary landscape, Reinforcement Learning has transcended its academic origins and found practical applications across diverse industries. Its impact is particularly notable in the field of business process optimization, advanced control systems, and sophisticated monitoring processes.

The versatility of Reinforcement Learning has positioned it as a powerful tool for tackling real-world challenges across various sectors. The development of Reinforcement Learning has been marked by three parallel and interconnected research threads:

- trial and error learning
- optimal control theory
- temporal difference methods

These parallel streams converged in the 1990s, giving birth to the unified field of Reinforcement Learning as we know it today. This convergence catalyzed remarkable achievements, most notably in the realm of game playing. Reinforcement Learning algorithms have demonstrated their capabilities by mastering complex games such as chess and Go.

The field of optimal control, which emerged in the 1950s, provided crucial mathematical foundations for what would later become reinforcement learning. Optimal control theory focuses on finding a controller that minimizes a measure of a dynamical system's behavior over time.

Richard Bellman made seminal contributions to this field; the Bellman equation, a fundamental optimal return function in Reinforcement Learning that expresses the relationship between the value of a state and the values of its successor states [1], and the formulation of Markovian Decision Processes (MDPs) (1957) [2], a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker.

After a period of relative quiet in the 1960s and 1970s, research in optimal

control experienced a resurgence in the 1980s. This revival saw exploration of various topics, including partially observable Markovian Decision Processes, approximation methods, asynchronous methods and modern treatments of dynamic programming. These developments laid crucial foundations for the integration of optimal control theory with other strands of RL research.

Temporal difference (TD) learning represents a novel approach to prediction problems in reinforcement learning. Unlike traditional methods that wait for a final outcome before updating predictions, Temporal difference methods update predictions based on other learned predictions, without waiting for the actual outcome. The conceptual roots of Temporal difference learning can be traced back to animal learning psychology, particularly the notion of secondary reinforcers. This concept suggests that stimuli associated with primary reinforcers (like food or pain) can themselves acquire reinforcing properties. The Q-Learning algorithm (Watkins, 1989 [3]) effectively united the three main threads of RL research, providing a method that combined aspects of trial-and-error learning, optimal control, and temporal difference methods, while Paul Werbos argued for the integration of trial-and-error learning with dynamic programming techniques (1987 [4]).

The next major challenge for Reinforcement Learning in game playing was the ancient game of Go, long considered one of the most difficult games for artificial intelligence due to its vast number of possible moves and complex strategies. Various attempts were made throughout the 2000s and early 2010s to develop Go-playing programs using RL techniques. In 2014, Google DeepMind patented an application of Q-learning to deep learning, titled ''Deep Reinforcement Learning" or '' Deep Q-Learning " that can play Atari 2600 games at expert human levels [5], and in 2015 a further improvement of Deep Q-Learning was published, called ''Double Deep Q-Learning " [6].

At the same time John Schulman introduced Trust Region Policy Optimization (TRPO) [7], which addressed the instability issue found in the previous algorithm, Deep Q-Learning, by using the trust region constraint to regulate the divergence between the old and new policy. However, TRPO is computationally complicated and inefficient due to its second-order optimization, leading to expensive and difficult implementation for large-scale problems.

Finally in 2017, John Schulman solved the complexity issue of TRPO by adopting first-order optimization in Proximal Policy Optimization (PPO) [8]. Schulman and his teams designed a clipping mechanism that forbids the new policy from deviating significantly from the old one when the likelihood

ratio between them is out of clipping range. Both DDQN and PPO represent nowadays the state-of-the-art of Reinforcement Learning and these two algorithms feature considerable differences, belonging to two distinctive categories, as shown in Figure 4.1 from OpenAI introduction guide [9].

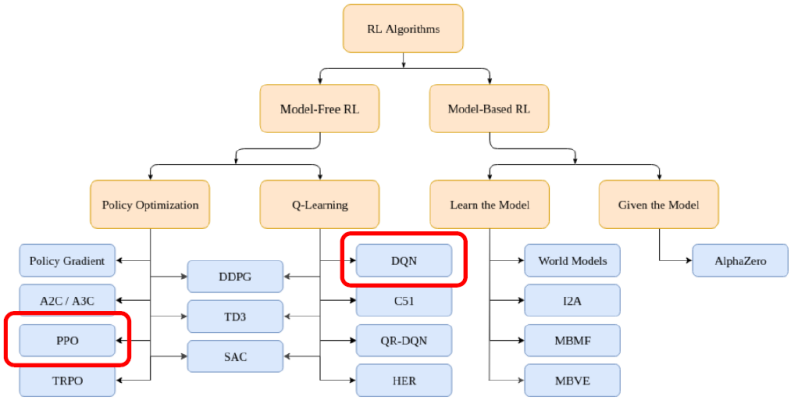


Figure 4.1: RL algorithms overview from OpenAI introduction guide

DDQN, which as mentioned is an improvement of DQN, is a model-free, value based, off policy algorithm, deriving from the Q-Learning algorithm. This means that it does not compute the optimal policy from an estimation of the environment dynamic, but from an estimation of the Q-function, which provides the ''quality" of each state-action pair. Such estimation is performed using a single neural network in case of DQN and two different neural networks in case of DDQN. After and during the training the agent, given a state, will choose the action with the highest Q-value computed by one of the two neural networks. It looks clear how DDQN represents and improvement of DQN, in particular thanks to the use of two separate neural networks, it reduces the possibility of computing an over-estimation of the Q-value during the training phase with just a small increase of computational complexity. Overall, the following can be stated in relation to the main features:

- **Stability**: can suffer from instability due to the correlation between consecutive experiences. Techniques like experience replay and target networks help mitigate this.
- **Sample efficiency**: relatively sample inefficient, as it requires a large amount of experience data to converge.

- **Implementation complexity**: simpler to implement compared to PPO, particularly in environments with discrete action spaces.
- **Use Cases**: Effective in discrete action spaces, like playing Atari games.

PPO, as mentioned, was developed by John Schulman in 2017, and became the default reinforcement learning algorithm at the American artificial intelligence company OpenAI. Many experts call PPO the state-of-the-art because it seems to represent a balance between performance and comprehension. Compared with other algorithms, the three main advantages of PPO are simplicity, stability, and sample efficiency. PPO is classified as a policy gradient algorithm for training an agent's policy network. Essentially, to train the right policy network, PPO takes a small policy update, so the agent can reliably reach the optimal solution. A too-big step may direct policy in the false direction, thus having little possibility of recovery; a too-small step lowers overall efficiency. Consequently, PPO implements a clip function that constrains the policy update of an agent from being too large or too small.

The architecture of PPO primarily refers to the neural network architecture used to represent the policy and value functions in the context of deep reinforcement learning. The policy network is a neural network that takes the current state of the environment as input and outputs a probability distribution over possible actions. This distribution represents the agent's policy, which specifies the probabilities of taking each action given the current state. The architecture of the policy network can vary depending on the complexity of the environment and the task at hand. Common choices include feedforward neural networks, recurrent neural networks (RNNs), or convolutional neural networks (CNNs), depending on whether the state space is structured (e.g., images) or sequential (e.g., time-series data).

In addition to the policy network, PPO often includes a value network. The value network estimates the expected cumulative reward (value) of being in a given state. This estimation helps reduce the variance of the policy gradient estimates and provides additional signal for learning. The architecture of the value network can be similar to the policy network, although it typically outputs a single value instead of a probability distribution. Overall, the following can be stated in relation to the main features:

- **Stability**: designed for stability with its clipped objective, which prevents large policy updates and allows for more reliable training.

- **Sample efficiency**: more sample efficient than DQN because it uses on-policy data effectively and updates the policy based on the collected experiences.
- **Implementation complexity** more complex than DQN. It is widely used due to its balance of performance and ease of use.
- **Use Cases**: commonly applied in a wide range of environments, including those with continuous action spaces, and is favored for its robustness.

Despite reinforcement learning algorithms are largely used in many applications, upon literature research it does not appear that a lot of work was carried out in relation to BESS operations optimization.

António Corte Real et Al [10], propose an architecture for the optimization of an electric battery operation in a real, online and data-driven environment that integrates state-of-the-art load forecasting combining CNN and LSTM neural networks to increase the robustness of decisions. Several Reinforcement Learning agents are trained with different algorithms (Double DQN, Dueling DQN, Rainbow and Proximal Policy Optimization) in order to minimize the cost of electricity purchase and to maximize photovoltaic self-consumption for a PV-Battery residential system. In this case the goal is to optimize the use of PV-Battery system.

Gwangwoo Han et al [11], propose an AI-based novel arbitrage architecture to maximize operating profit in the electricity market composed of a grid operator, an energy storage system, and customers. The algorithm consists of two parts: the first is recurrent neural network-based deep learning for overcoming the future uncertainties of electricity prices and load demands. The second is reinforcement learning to derive the optimal charging or discharging policy considering the grid peak states, the BESS operator profit, and customers load demand. This case is much closer to the environment that is taken in consideration for this project, but it is not considered the other services that the BESS can provide to the grid (ancillary services) neither the battery degradation.

Hyuna Kang et al [12], develop a reinforcement learning (RL)-based optimal scheduling model to better reflect the continuous behaviors of residential buildings connected to the grid and equipped with a BESS and PV system. The optimal scheduling models are developed using four algorithms from among the various RL techniques according to training methods. The results of the case study show that the developed RL-based optimal scheduling

model using Proximal Policy Optimization (PPO) can be applied to effectively operate the BESS with a PV system, considering possible uncertainties in the real world. Even in this case the goal is to optimize the use of PV-Battery system.

Zhiquiang Wang et al [13] propose to combine deep learning-based demand forecasting and resource planning techniques to address the challenges to optimize energy storage systems charging and discharging strategies in market-oriented trading environments within power grids. By improving the capabilities of deep learning models, the aim is to improve the accuracy of demand forecasts, enabling more informed decision-making regarding the optimal allocation of energy storage resources. In addition, deep learning-based resource planning algorithms can dynamically adjust charging and discharging strategies in response to real-time market signals, grid conditions, and operational requirements. This paper focuses on the optimal use of the BESS and other energy generators composing a microgrid, such as wind turbines, fossil fuels generator and photovoltaic panels, thus a broader application.

Sara Abedi et al [14] developed an intelligent and real-time battery energy storage control based on a reinforcement learning model focused on residential houses connected to the grid and equipped with solar photovoltaic panels and a battery energy storage system. A cyclic time-dependent Markov Process was uniquely designed to capture existing daily cyclic patterns in demand, electricity price, and solar energy. The Markov Process was successfully used in the Q-learning algorithm, resulting in more efficient battery energy control and saving electricity costs. Even in this case the goal is to optimize the use of PV-Battery system.

Compared to the previous work done, this project aims to develop a reinforcement learning based optimizer for just a BESS integrated in the power grid, managing the energy exchange and the services to provide in order to maximize the BESS return of investment, and taking into account all data available from the environment such as load, energy price, battery status, battery aging, etc.

Chapter 5

Proposed architecture

Based on the environment and the type of problem that needs to be solved, a two stage architecture was designed in the initial phase, which can be seen in Figure 5.1.

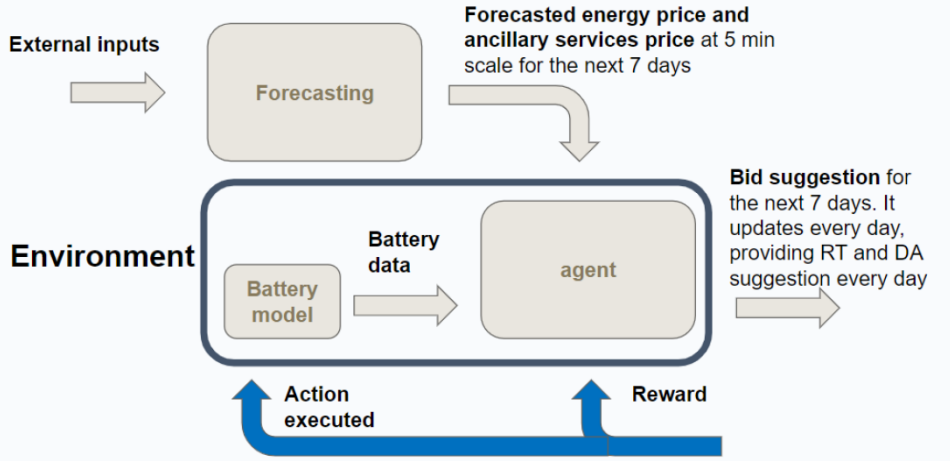


Figure 5.1: proposed architecture

The first stage forecasts the energy price and the ancillary services price for the next 7 days with a 5-minute resolution. The external inputs considered should include weather forecasts, generation forecasts, load forecasts and fuels price forecasts at least. The trained forecasting model provides the prices to an agent trained with reinforcement learning algorithms which provides a bid strategy suggestion for the next 7 days with a 5-minute resolution, taking into account also information about the battery status such as state

of charge, number of cycles and aging. It was established to provide a 7-day bidding strategy because the NYISO requires all the market participants to provide bids in such a period, considering financially binding only the real time and day ahead bids. It seemed very clear that this project required a considerable amount of time and resources in order to obtain some results, so before starting it was decided to minimize the effort on the forecasting model and mainly focus on agent training.

Chapter 6

Data collection

After defining the overall architecture, it was necessary to collect all data to train both the forecasting model and the agent.

As mentioned before it was decided to focus on the NYISO because its website provides a very broad range of high quality data. In relation to the forecasting model it was possible to freely download the energy price, load and generation mix for all 2023 and 2024 with a 5-minutes resolution. From an other website it was possible to download detailed weather forecasts data with 1-hour resolution for the same period of time. It was not possible to download the ancillary services and the fuels prices.

At this point the first issue came up, because without information about the ancillary services price it was not possible to provide forecasts and therefore to train the agent to consider such services when elaborating the bidding strategy. Considering the situation it was decided to simplify the environment, considering only the energy price, and therefore considering only the option to buy and sell energy in arbitrage mode during the bidding strategy.

This choice does not impact on the architecture shown in figure 4, but simply reduces the agent action space to three options: buy, sell, wait. Some Python code was written in order to download and process all the aforementioned data in the format needed for the trainings.

In relation to the battery model and the agent training there was no need to download and process any further specific data. The battery was modeled with a simple linear model, representing the charge-discharge dynamics when energy was bought or sold.

Chapter 7

Modelling BESS with Reinforcement Learning

For the forecasting stage, since it was established to minimize the effort, a library dedicated to time series forecasting called Autogluon was chosen. It includes a variety of different models to train and it is also able to combine different trained forecasting models to optimize the predictions. In the later chapters it will be shown which models performed better.

For the Reinforcement Learning stage it was decided to apply two of the best reinforcement learning algorithms currently available: Double Deep Q Network (DDQN) and Proximal Policy Optimization (PPO). In order to make the algorithms working it was also necessary to model the environment representing the BESS and its interaction with the grid.

As mentioned this second step was the real core of the project, and the first choice to make was the library to use for the implementation. The final decision was Stable Baselines3 (SB3), which includes a set of reliable implementations of Reinforcement Learning algorithms in PyTorch and it is a continuation of Stable Baselines3. The library provides a unified interface for various algorithms, such as PPO, A2C, SAC, TD3, and DQN, making it easy to train and compare different methods. It is designed to be user-friendly, with good documentation and examples for beginners.

It also offers built-in support for vectorized environments, allowing for efficient parallel training, it integrates very well with OpenAI Gym environments and provides tools for custom environment creation. It includes features for logging, saving, and loading models, as well as callback functions

for custom behavior during training. Stable Baselines3 emphasizes reproducibility, provides pretrained models for benchmarking, it is actively maintained and has a considerable community of users and contributors in the RL field.

Thanks to Stable Baselines3 it was possible to rapidly create a scalable customized environment representing the most important features of the battery dynamics and the interaction between the battery and the grid based on the energy price. It was then possible to run built in implementations of DQN and PPO in the created environment, performing all the necessary tests and fine tunings.

Chapter 8

Development and testing

Recalling the architecture shown in figure 4, the development was carried out in two main steps using Python as programming language with some of the most popular libraries used in AI development, such as NumPy, Pandas and PyTorch. Since it was necessary a considerable computational capacity, but there was no budget available, it was decided to use the free version of Google Colab which allows the use of CPUs or T4 GPUs.

8.1 Forecasting

The first step was focused on the development and testing of a forecasting model to predict the energy and ancillary services price. As mentioned before it was possible to obtain good quality data only about the energy price and not about the ancillary services price, therefore the work was focused only on the creation of a forecasting model for the energy price. It was decided to not dedicate too much time to this step, therefore to create the forecasting model it was chosen Autogluon, a library with pre-made models and training algorithms. Figure 8.1 shows the forecasting model architecture that was conceived.

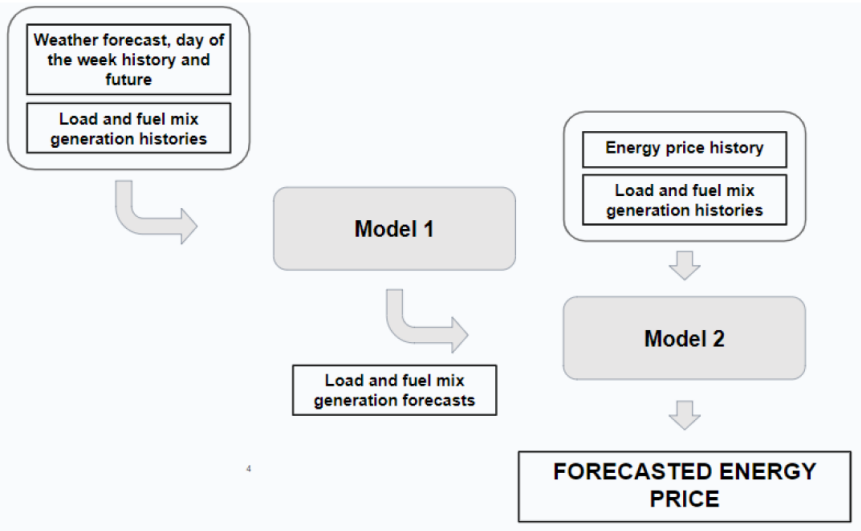


Figure 8.1: Forecasting architecture

8.1.1 Model 1: forecasting covariates

The first model was trained to forecast the load and the fuel mix generation given their history and given the past and future weather conditions plus the identification of the day of the week. The choice was made because weather conditions and day of the week are between the primary covariates influencing the target variables. There are other important covariates that have a consistent influence, such as the fuel prices but it was not possible to find complete and accurate data.

So the fuel mix and load history were downloaded from the NYISO website, for all 2023 and with 5-minutes resolution. The weather conditions were downloaded from VisualCrossing website [15], specialized in weather data, and it was possible to obtain a very wide range of data for all 2023 with a 1-hour resolution:

- Temperature
- Perceived temperature
- Dew point
- Humidity
- Precipitations
- Precipitation probability

- Precipitation type
- Snow
- Snow depth
- Wind gust
- Wind Speed
- Wind direction
- Sea level pressure
- Cloud cover
- Visibility
- Solar radiation
- Solar energy
- UV index

To perform the models training only a subset of such data were considered. Another aspect to mention is the need to perform some further data processing, such as up sampling in order to train the models with a unified dataset of time series, all with a 5-minutes resolution.

Autogluon provides very straightforward function to train the models, but it has specific requirements about the data structure to provide, which needs to be imported as a class from the library and it is called `TimesSeriesDataFrame`. It is possible to create it from a Pandas DataFrame, so all the necessary data were imported and processed using Pandas functions and then converted in the aforementioned data structure.

In order to train new models it is necessary to first instantiate a dedicated class called `TimeSeriesPredictor`. The list of arguments is quite long and allows a considerable flexibility. For the purpose of this project it was enough to include the following arguments:

- **Prediction length**, to establish how many steps in the future the model will have to predict
- **Path**, to define where to save the trained model
- **Target**, to define the specific time series in the training set to predict. This is a critical feature because one trained model can only predict one specific time series after the training. This means that in this case

to have the prediction of load and all the generation mix it is necessary to train a model for each specific time series.

- **Known covariates**, to define all the other time series in the training set to use to obtain correlations and other insights to optimize the prediction.
- **Frequency**, to define the prediction time resolution

Once the class is instantiated the training can be performed with a dedicated method called `fit`. Even in this case the list of arguments is quite long, allowing a considerable flexibility, and for the purpose of this project it was enough to include the following arguments:

- **Train data**, the training dataset in the form of a `TimesSeriesDataFrame`
- **Preset**, to define the accuracy of the trained model prediction. It is possible to choose between `fast_training`, `medium_quality`, `high_quality` and `best_quality` according to the desired tradeoff between training speed, memory requirements and prediction accuracy.
- **Time limit**, to define a time out for the training

The library training method is quite extensive and automatically performs the training of multiple models, from the simplest in `fast_training` mode to the most complex in `best_quality` mode. It eventually defines the final trained model according to some internal test and can also create a weighted ensemble of the best performing trained models.

For the purpose of this project and due to the limited computation capabilities it was not possible to select a higher preset than `medium_quality` with a prediction length of 600 steps.

Nine models in total were trained, one for each of the following time series:

- Load
- Natural gas generation
- Dual fuel generation
- Wind generation
- Nuclear generation
- Other fossil fuels generation
- Other renewables generation

- Hydro power generation
- Total generation

Given the fact that the temporal behavior is very different between the listed variables it is reasonable to expect that different models better predict different variables and in all cases the best performance was given by a weighted ensemble of different trained models.

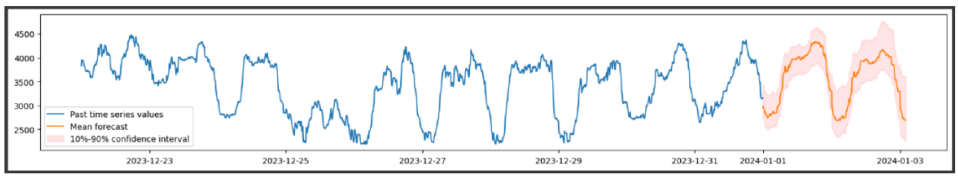
As an example in Figure 8.2 it is reported a frame of the Load training log, where it is possible to notice the composition of the weighted ensemble model and the list of all models trained.

```
Fitting simple weighted ensemble.
        Ensemble weights: {'DirectTabular': 0.01, 'ETS': 0.05, 'Naive': 0.26, 'Theta': 0.68}
        -0.0032       = Validation score (-WQL)
        0.86    s     = Training runtime
        1.63    s     = Validation (prediction) runtime
Training complete. Models trained: ['Naive', 'SeasonalNaive', 'RecursiveTabular', 'DirectTabular', 'ETS', 'Theta', 'TemporalFusionTransformer', 'WeightedEnsemble']
Total runtime: 244.45 s
Best model: WeightedEnsemble
```

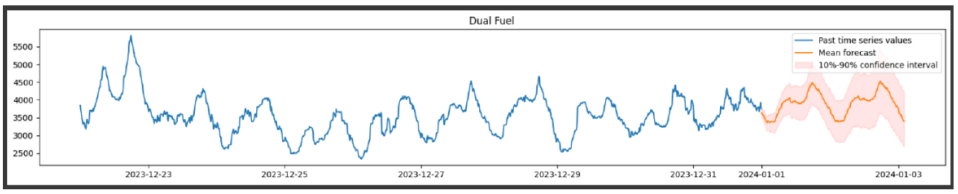
Figure 8.2: Training log from Autogluon

After the training it is possible to test the models using the method predict, which requires a `TimeSeriesDataFrame` with past time series of target and covariates used by the method to start the prediction, and another `TimeSeriesDataFrame` of the covariates future values that will help to make the prediction more accurate.

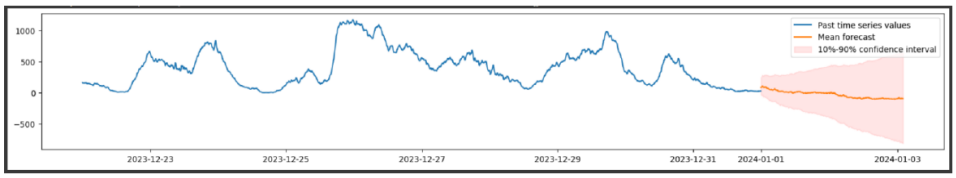
The graphs shown in Figure 8.3, 8.4 represent a comparison between a 2-days prediction and the real records for all the afore listed variables, after the whole training dataset. The prediction is represented by an average and a 10% - 90% confidence interval.



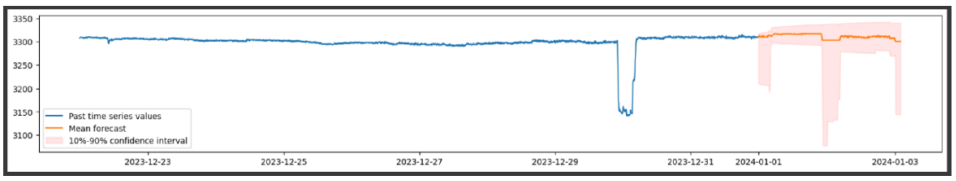
(a) forecasting models testing of load and energy mix generation



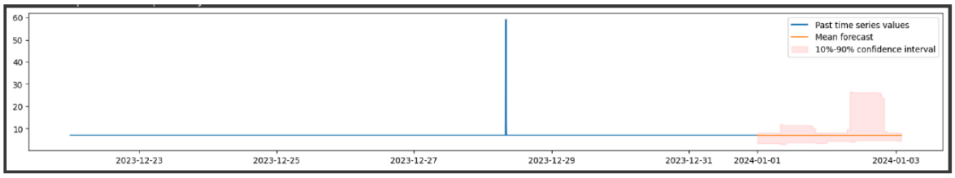
(b) forecasting models testing of load and energy mix generation



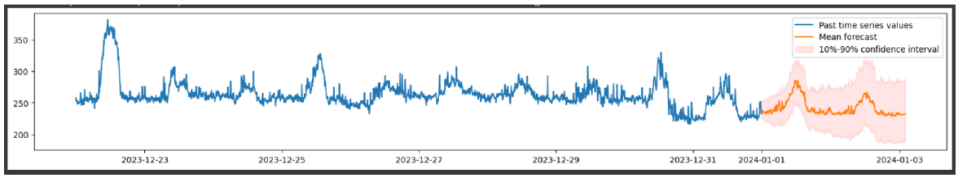
(c) forecasting models testing of load and energy mix generation



(d) forecasting models testing of load and energy mix generation

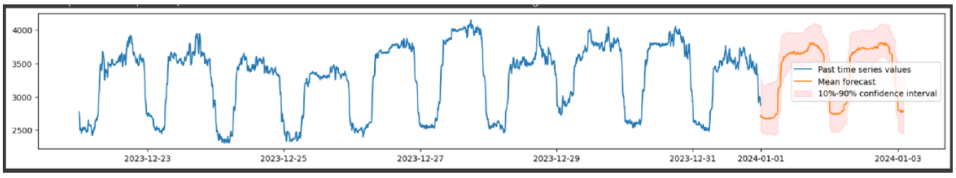


(e) forecasting models testing of load and energy mix generation

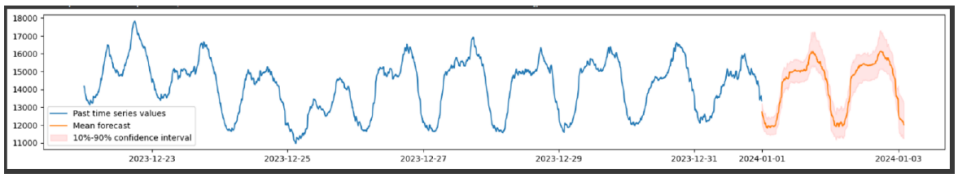


(f) forecasting models testing of load and energy mix generation

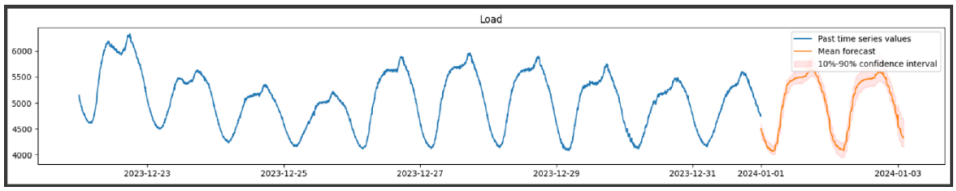
Figure 8.3: forecasting models testing of load and energy mix generation



(a) forecasting models testing of load and energy mix generation



(b) forecasting models testing of load and energy mix generation



(c) forecasting models testing of load and energy mix generation

Figure 8.4: forecasting models testing of load and energy mix generation

As mentioned before it is possible to notice that the variables with a more stable seasonality are predicted with more accuracy. Furthermore it is possible to observe that especially the wind energy and the other renewable energy variables, which are highly correlated to the weather conditions, have a lower accuracy: one possible explanation could be that the weather data have a 1-hour time resolution, compared to the 5-minutes resolution of the target time series to predict.

8.1.2 Model 2: forecasting energy price

Once the first step models were trained, tested and saved, the focus moved to the second model, with reference to the architecture in figure 6. The training and testing functions are the same used in the first models, and the difference lays just in the dataset used.

For training the model, the energy price, energy generation and load histories dataset was used for the whole 2023, with a 5-minutes time resolution. The prediction length in the `TimeSeriesPredictor` was set to 600 steps and few experiments were carried out trying different prediction quality presets,

selecting only specific models to train and reducing the number of covariates time series, removing the ones that were less relevant and obtained less accurate models.

To test the trained models, together with the history dataset it was necessary to add the future time series of the covariates, which were provided by the models trained during the step 1.

The plot in Figure 8.5 shows a comparison between the prediction obtained from the best trained model and the real values of the energy price.

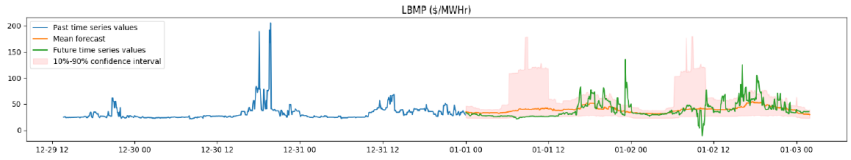


Figure 8.5: Energy price forecasting model testing and comparison with real price

As mentioned the training dataset is for the whole 2023, so the 600 steps prediction with a 5-minutes resolution covers about the first two days of January 2024.

It is possible to notice that the mean value of the prediction is quite aligned with the price trend, but despite the variance increases correctly where the price volatility increases, the mean predicted value does not really align with the several peaks.

Even considering that forecasting the energy price with high accuracy is still a great challenge today, the accuracy level obtained with the best model shown in figure 9 cannot be acceptable to obtain very good results in this project. Furthermore the initial goal was to obtain a 7-days ahead price prediction, necessary to develop the 7-days ahead bidding strategy, but the memory space available in Google Colab did not allow to reach much more than 2 days maintaining an acceptable level of accuracy.

Nevertheless it was decided to move on and mainly dedicate time and resources to the Reinforcement Learning step, leaving this step as an open problem to improve in the future and accepting the low accuracy of the best trained model.

8.2 Reinforcement Learning

As mentioned for the Reinforcement Learning stage it was decided to use Stable Baselines 3 (SB3) library.

8.2.1 Development

Customized Environment

The first step for this part was to create a customized and scalable environment, that allowed to start modelling the problem with just the essential features and then progressively add more complexity.

According to the customized environment template in SB3, first it is necessary to define the new `CustomEnv` class that inherits from `gym.Env`. Inheriting from `gym.Env` ensures that the customized environment follows the OpenAI Gym interface, which is required for compatibility with Stable Baselines 3.

Inside the new class it is necessary to define the constructor `def __init__(self)`. In this method the parent class constructor (`gym.Env`) is called to ensure that any setup done by the parent class is properly executed, and both action space and state space are defined. It is also possible to initialize other attributes that are necessary to complete the environments.

To initialize the customized environment it is necessary to provide a DataFrame containing the time series of the energy price which will be used to include in the observation space the current energy price, together with the battery state of charge. As mentioned the first environment was designed to minimize the complexity, thus the action space is reduced to three values representing buy, sell, no action.

Other attributes have been added to complete the environment:

- `self.current_step = 0`, to identify the current step during the simulations
- `self.max_steps = len(self.df) - 1`
- `self.wallet = 400`, to define the account where the money from the energy trading is stored, in dollars
- `self.net_wallet = 400`, to define the account that considers only the financial gain after the amortization cost of the battery. Although

there are several other costs to take in consideration this represents the most relevant and it was decided to start with a simple environment.

- `self.amort = (self.capacity*139000)/5000)` defines the amortization cost per battery cycle. In other words this number provides the portion of battery cost divided by its life expressed in cycles. Such numbers are set based on market averages: the average cost of a battery is about 139000 $ for each MWh of capacity, and the average life of a battery is about 5000 cycles. This parameter, multiplied by the fraction of cycle increment occurred after taking the action at the current step allows to obtain the fraction of battery cost for each step taken. It is important to observe that normally the cycle increment, which corresponds to the battery life reduction is not fixed and depends on how the battery is used and other conditions, but to keep the model simple it was chosen to reduce the battery life only proportionally to the energy charged or discharged.
- `self.capacity = 129`, to define the battery capacity in MWh. This value represents the size of the BESS and it was chosen a typical size of a large scale installation
- `self.discharge = 50`, to define the battery discharge rate in MWh. It represents the battery energy discharged in 1 hour of operation.
- `self.soc = 50`, to define the state of charge in %.
- `self.delta_soc = 0`, to define the state of charge variation, necessary to track the number of cycles.
- `self.charge_bonus = 0.1`, to define a parameter for the reward function
- `self.battery_penalty = -4`, to define a parameter for the reward function
- `self.delta_soc_penalty = -0.05`, to define a parameter for the reward function
- `self.max_reward = np.max(self.df)*(5/60)*self.discharge`, to standardize the reward as suggested by SB3 guide in order to optimize the training.

After the constructor it is mandatory to have a `step()` method, which defines the environment dynamic, so everything that happens after an action is executed.

This method increases the step variable and then depending on the action selected by the agent updates the necessary attributes and assigns the reward. If the action is either buy or sell the wallet and the state of charge are updated considering the current energy price and the fixed discharge rate defined in the attributes.

Also in both cases it is included a condition that prevents the battery to be charged over 85% and discharged over 15%, including a negative reward to train the agent to avoid such actions in those cases. In this way it is possible to model a physical limitation that prevents the battery to be damaged and to train the agent to avoid damaging the battery.

The reward is calculated differently between buy and sell, and in both cases it is normalized. In the first case the reward is proportional to the money gained with the sales on top of a corrective coefficient necessary to optimize the training. In the sell case the reward includes the money spent for the purchase and a positive fixed bonus to reward the agent when it decides to charge. This is necessary otherwise with a buy action the reward would always be negative and the agent would never choose this action; with the extra bonus the reward would turn positive if the buy price is low, which is exactly when the agent should choose to buy.

Even the net wallet is calculated differently between buy and sell; in the first case a fraction of the battery cost proportional to the fraction of cycle increase due to the action taken is summed to the price paid to buy the energy, while in the second case the battery cost fraction is subtracted to the money gained selling the energy.

The neutral action does not produce any effect and leaves the reward to 0.

The last part of the method increases the battery cycle counter, used as a KPI and to compute the wallet number of cycles ratio, another important KPI to measure agent effectiveness.

All the variables stored in the `info` dictionary are necessary to track both training and testing process and adjust the hyperparameters.

Another mandatory method is the `reset()`. This is necessary to restore the initial conditions when a new episode is started.

Training function

After developing the environment it was necessary to develop a training function to train the agent. This function has the typical argument neces-

sary to train a model, such as the environment object, the time steps, the learning rate and the seed. Furthermore, the type of Reinforcement Learning algorithm and the policy network parameters are added to make the function use more flexible.

The seed is set for the Numpy random generator, used by SB3 library and the environment object is wrapped into the `Monitor` object from the OpenAI Gym library to create a training log with main details.

Then the model is created depending on the selected algorithm, the method `learn` executes the training and a print function, developed for this project, plots on a Plotly graph the relevant variables of a training.

In order to extract such variables and plot them it was necessary to create a callback class. It inherits from `BaseCallback` superclass, which belongs to SB3 library. The constructor method defines two attributes lists, `self.training_log` and `self.env_log`, that store all the desired values during each step.

Testing Function

After the training function the testing function was developed. The arguments are quite typical and are the trained model, the environment with test data and the number of steps to test.

First the environment is reset since a new episode needs to start. Then a series of lists are initialized to record the values of the relevant variables necessary to evaluate the testing, which is implemented with a for loop where the `predict` method is called on the trained model given the current environment state. This method returns the action selected by the agent and the method `step` is called on the environment to progress to the following state according to the action taken and to obtain the values of all the variables that need to be recorded. A print function developed for this purpose plots all the recorded variables on a Plotly graph.

The last two relevant utility functions that have been developed are to save the training and testing results. All the classes and functions so far presented are called in a main function that is used to run the different experiments. First the training and testing environment objects are instantiated from the developed class: the difference between the two is the dataset provided. In the first case the dataset is the energy price for the whole 2023 in the city

of New York with a 5-minutes resolution, while in the second case the time period is January 2024.

Then some parameters of the policy network are defined, considering that for all the experiments the chosen architecture was a Multi-Layer Perceptron. In particular it is defined the activation function and the size of the policy network in terms of number of layers and number of neurons for each layer. Some lists are initialized to store the relevant variables.

According to training best practices it is important to repeat training and related testing with the same environment and parameters using a multitude of different seeds since the initialization of the policy networks parameters can influence the final result. For this reason, a for loop is created to repeat the test for seven different seeds. This loop includes all the functions that were previously resented and two hyperparameters are set: the training length (105000), corresponding to the whole 2023 of energy price in 5-minutes steps and the learning rate (0,0003). After presenting all the classes and functions developed in this second step of the project the following part will explain all the most relevant tests that were carried out.

8.2.2 Testing Env5: basic environment

Random agent

The first test was carried out using the basic environment previously presented and a random agent, meaning an agent that selects the action to take on a random basis. This test was carried out for two main reasons: first to make sure that all the classes and functions developed did not have bugs, and second to establish a base benchmark to compare the results of all the following experiments.

In order to guarantee clarity and comparability between all the different experiments it was decided to show, for each experiment, the mean value and the related standard deviation of all the seeds of three KPIs: Gross Wallet, Net Wallet and Gross Wallet - Number of cycles ratio.

The first KPI shows how much money the agent allowed to gain during the testing time frame. The Net Wallet shows how much money the agent allowed to gain considering the amortization costs of the battery caused by its life span reduction during the operation in the testing time frame. This KPI is very important because, due to the high battery cost and the limited battery life, the ideal trading strategy should consider making an energy transaction only when the financial gain is relevant, and it should make

the battery last longer. The Wallet - Number of cycles ratio shows similar information, but on a relative basis, allowing to have a better understanding of the agent strategy effectiveness considering the battery amortization costs.

Furthermore, it was decided to show the State Of Charge behavior against the energy price in order to directly evaluate, on a qualitative basis, the agent trading strategy.

As previously mentioned, the first test was carried out using the basic environment, identified as ''env5" and a random agent. In order to ease the series of experiments comprehension a summary table with the ''env5" main features is shown in Table 8.1.

Feature name	**Value**
Variables in the observation space	Current energy price, battery SOC
Action space	Discrete, 3 values: buy, sell, neutral. A fixed amount of energy is exchanged each step.
Buy reward function	Normalized. Proportional to the gross wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the gross wallet gain
Battery overcharge and undercharge penalty	-0.1

Table 8.1: ''env5" main features

The three KPIs are shown in Figure 8.6, 8.7, 8.8.

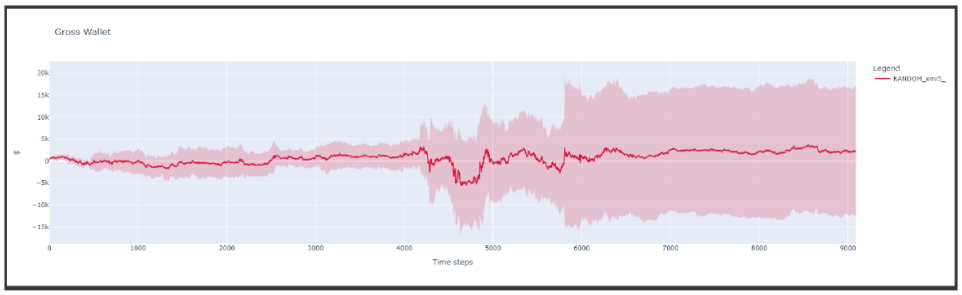


Figure 8.6: Gross Wallet KPI

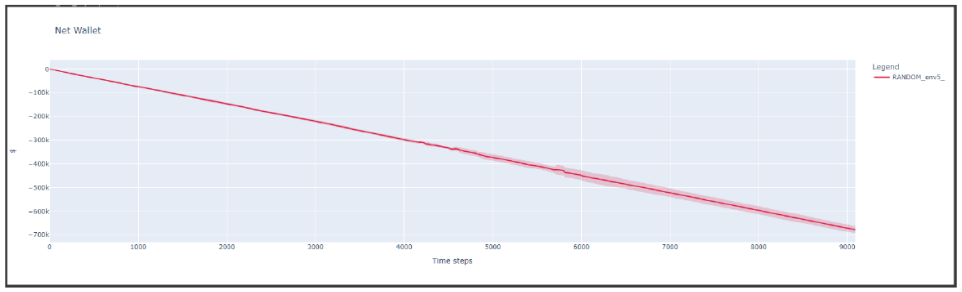


Figure 8.7: Net Wallet KPI

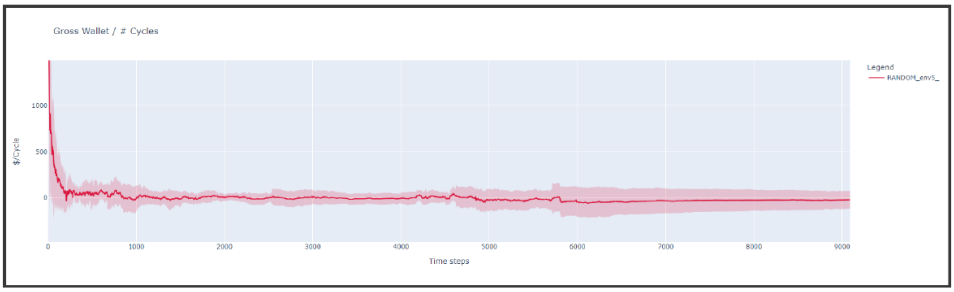


Figure 8.8: Gross Wallet – Number of cycles ratio KPI

It can be noticed that both Gross Wallet and Gross Wallet – Number of cycles ratio have an average value around zero, which is legit considering that the probability distribution chosen to pick the action is a uniform distribution. Even the Net wallet trend looks legit, since for each action taken, due to the set battery size and cost, approximately 116 $ are subtracted to the wallet; thus, the energy sale is profitable only if the energy price is above 28 $/MWh. For this experiment the State of Charge versus energy price trend is omitted since the agent was operating randomly.

PPO and DQN

The following experiment was to test both PPO and DQN in the basic environment ''env5". This test was carried out to verify how effective such algorithm could be in a similar environment, and the expectation was at least to have consistently higher values for the Gross Wallet and Gross Wallet – Number of cycles ratio.

In Figure 8.9, 8.10, 8.11 the three KPIs are shown, and the three batches of experiments random agent, PPO and DQN are compared and Table 8.2 shows a summary of the results.

Agent	**Gross Wallet $**	**Net Wallet $**	**Gross Wallet/# of cycles $/#C**
Random ''env5"	-4000 ± 17800	-677500 ± 15000	-22 ± 96
DQN ''env5"	77650 ± 52000	-614000 ± 246000	470 ± 430
PPO ''env5"	65000 ± 29000	-735000 ± 454000	290 ± 140

Table 8.2: ''env5" testing summary

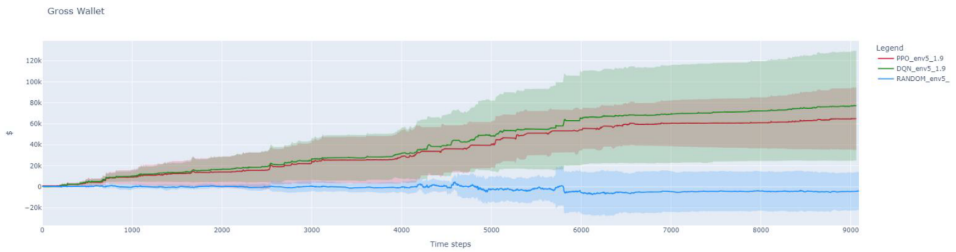


Figure 8.9: Gross Wallet KPI

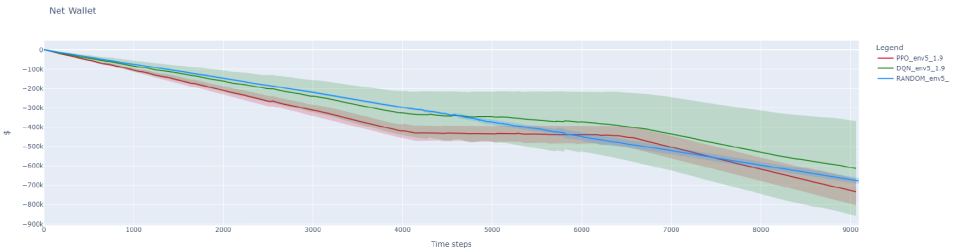


Figure 8.10: Net Wallet KPI

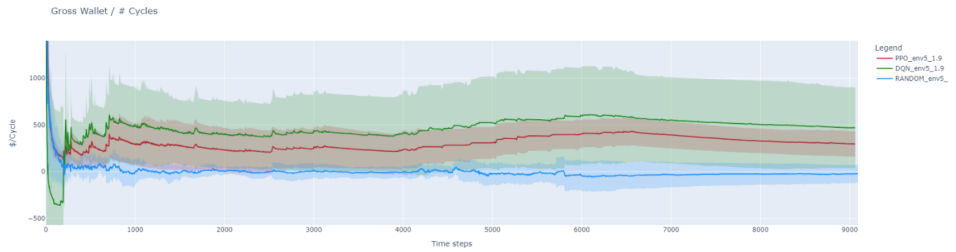


Figure 8.11: Gross Wallet – Number of cycles ratio KPI

First of all it is very interesting to notice, as expected, that compared to the random agent, the other two trained with PPO and DQN allow a considerable gain in the Gross Wallet and the Gross Wallet – Number of cycles ratio; this proofs the efficacy of both algorithms in optimizing the Gross Wallet gain, on which the reward function is based. Differently it can be noticed that the Net Wallet behavior did not change compared to the performance of the random agent and this is totally legit, since the reward function does not keep in consideration the battery amortization cost.

Another important observation is about the performance similarity of both DQN and PPO on the average values, with DQN performing slightly better, but on the other hand less stable on different seeds training compared to

PPO, observing the variance. Such results did not come right on the first experiments, and it was necessary to fine tune the reward function parameters and the policy networks size multiple times.

Figure 8.12 shows the State Of Charge behavior versus the energy price on a test with the best agent trained with DQN. The action taken plot is also included. Only a portion of the training time frame is shown, but it is possible to observe moments with low, medium and high price volatility.

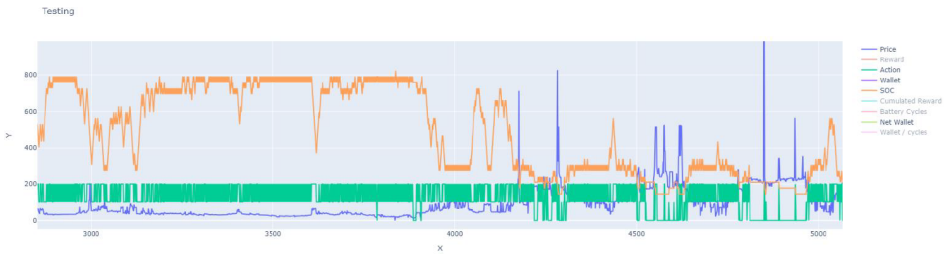


Figure 8.12: SOC vs energy price of best model trained with DQN

During low volatility time the battery tends to be fully and over charged, discharging as soon as the energy price increases. During medium a high volatility the battery tends to stay low, apparently waiting for the price to lower more to the low volatility range. This prevents to fully exploit the high price peaks when they occur. Considering that the agent is basing its decision only on the current energy price and State of Charge the results seem to be very good, but there is definitely room for improvements.

Figure 8.13 shows the State Of Charge behavior versus the energy price on a test with the best agent trained with PPO. The action taken plot is also included. Only a portion of the training time frame is shown, but it is possible to observe moments with low, medium and high price volatility.

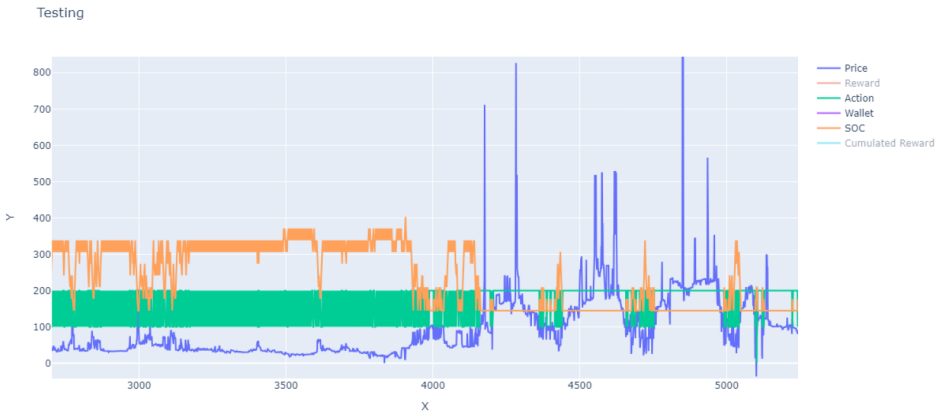


Figure 8.13: SOC vs energy price of best model trained with PPO

In this case the agent maintains the State Of Charge at mid-level during the low volatility, discharging when the price increases. During medium and high volatility the battery tends to be undercharged, waiting as in case of DQN for the price to lower to the low volatility range. Thus, the behaviors are similar, with DQN performing a bit better on the medium – high volatility part, which justifies a slightly better performance looking at the KPIs.

8.2.3 Testing Env6: basic environment with 2h true price ahead in the observations

Considering the positive results obtained knowing just the current energy price in the environment it was decided to increase the complexity allowing the agent to se also the energy price of the future 20 steps after the current one. This experiment was carried out with the expectation to improve the test results knowing the energy price in the future when choosing the action to take.

In order to maintain simplicity in this first experiment with future price values it was decided to have an oracle as a forecasting model, provided only the true future price values. The new environment class called ‘’env6” was modified from ‘’env5” and Table 8.3 shows a summary of the updated features.

Feature name	Value
Variables in the observation space	Current energy price, true energy price of next 20 steps, battery SOC
Action space	Discrete, 3 values: buy, sell, neutral. A fixed amount of energy is exchanged each step.
Buy reward function	Normalized. Proportional to the gross wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the gross wallet gain
Battery overcharge and undercharge penalty	-0.1

Table 8.3: ''env6" main features

In this environment both DQN and PPO have been tested and the three KPIs are shown in Figure 8.14, 8.15, 8.16, while Table 8.4 shows a summary of the results.

Agent	Gross Wallet $	Net Wallet $	Gross Wallet/# of cycles $/#C
DQN ''env5"	77650 ± 52000	-614000 ± 246000	470 ± 430
PPO ''env5"	65000 ± 29000	-735000 ± 454000	290 ± 140
DQN ''env6"	63700 ± 39000	-312500 ± 260000	3033 ± 5700
PPO ''env6"	258000 ± 69000	-524000 ± 244000	2934 ± 4400

Table 8.4: ''env6" testing summary

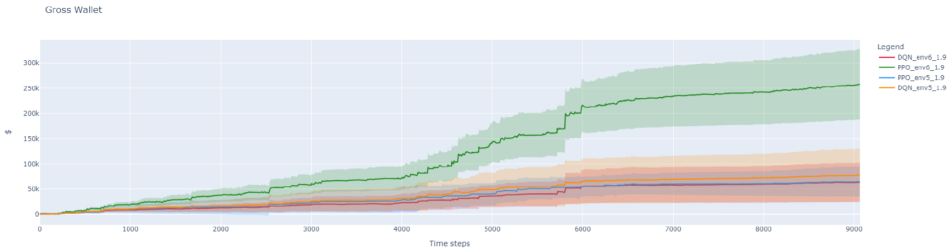


Figure 8.14: Gross Wallet KPI

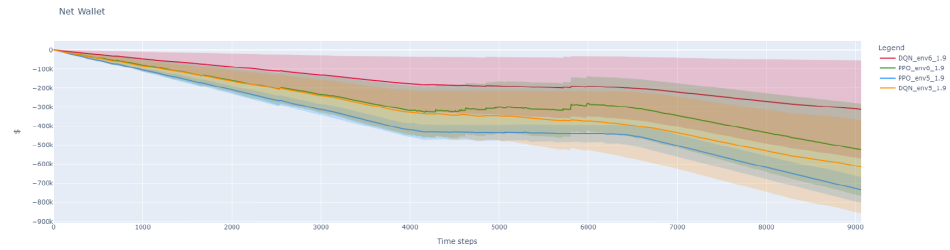


Figure 8.15: Net Wallet KPI

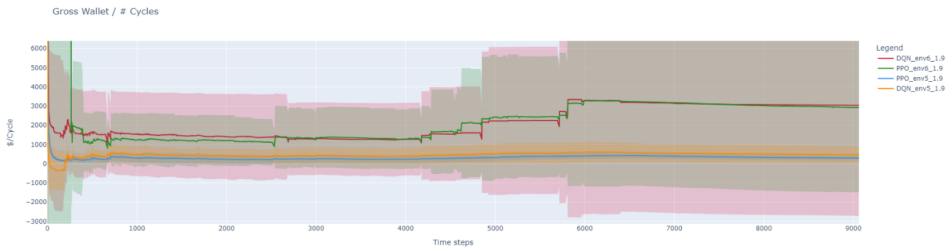


Figure 8.16: Gross Wallet – Number of cycles ratio KPI

These results look remarkably interesting. Observing the Gross Wallet it seems that PPO was able to exploit the information about the future energy price while DQN not, performing even slightly worse than the training in the ''env5". Indeed, observing the other two KPIs it is possible to conclude that even DQN was able to exploit the information about the future energy price taking actions more effectively, thus providing higher wallet gains.

Other important insights can be obtained analyzing the State Of Charge versus Energy price plots of the best agents testing trained with both PPO and DQN, in Figure 8.17, 8.18 respectively, which also include the action taken plot.

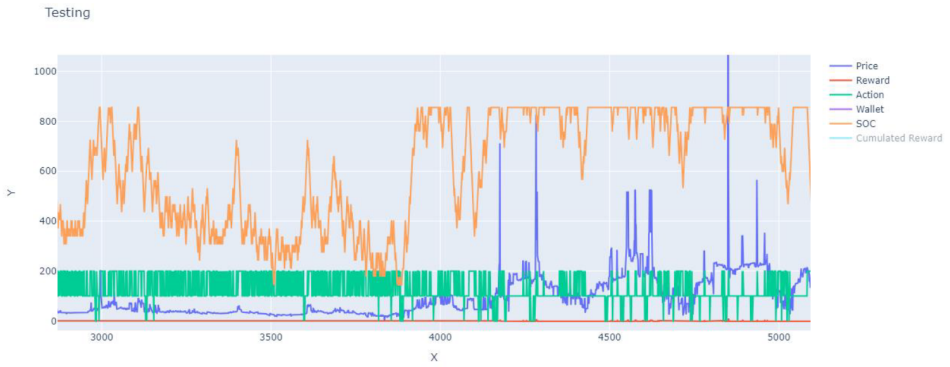


Figure 8.17: SOC vs energy price of best model trained with PPO

Figure 8.18: SOC vs energy price of best model trained with DQN

As in the previous similar plots the snap incudes three different energy price volatility levels: low, medium and high. In the low volatility time frame the State Of Charge behavior is similar, with the PPO agent making wider variations that allow to exchange more energy and thus gain more money. This justifies the better Gross Wallet performance during low price volatility.

The strategies completely change at the beginning of the medium volatility time frame where both agents have the battery fully charged. The PPO agent chooses to maintain the battery highly and some times over charged, discharging only when the energy price has significant peaks. The DQN agent chooses to fully discharge the battery as soon as the energy price is higher but not on the highest peaks, and then it maintains the battery at the minimum, often trying to over discharge it. Then it tends to charge the battery right before relevant price peaks only with the necessary amount of energy needed to sell during the peak. This strategy results less effective than the PPO agent one because the buy price before a peak is not necessarily the best but since the battery is low it is the only option to make a gain on the peak.

Overall neither the two strategies are optimal because they could exploit much more the price peaks with deeper charge – discharge patterns, but clearly the PPO agent performs better allowing a much bigger gain. One of the reason of such difference could be the use of DQN instead of DDQN, which as mentioned before should guarantee better results.

Another interesting observation on both the State Of Charge patterns is the ripple due to small charge – discharge cycles that occur mainly during low price volatility time frames. Figure 8.19 shows a plot zoom.

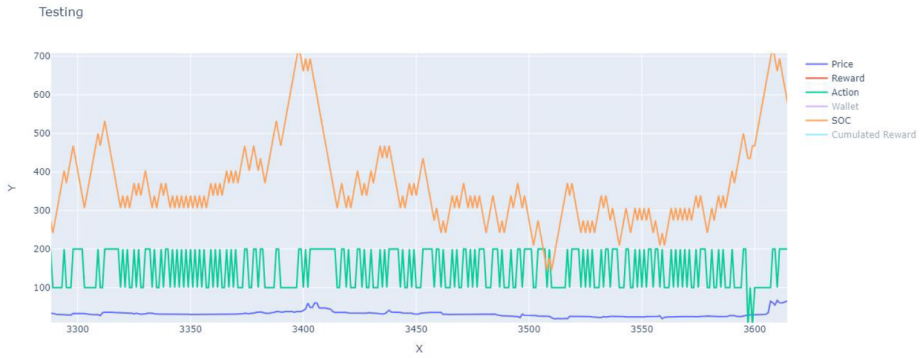


Figure 8.19: ripple in SOC from PPO testing

It is hard to explain this behavior but it is clear that it does not bring any positive effect on the KPIs, especially on the Net Wallet and the Gross Wallet – Number of Cycles Ratio. It is reasonable to conclude that this ripple effect, which is present event in the ''env5" tests, is one of the main causes of the negative trend of the Net Wallet.

8.2.4 Testing Env8: environment with continuous action sapace and 2h true price ahead in the observations

The following test aimed to both solve the ripple effect and to further increase the environment complexity, leaving the agent the choice of how much energy to exchange in one single step. Thus ''env6" class was modified into the new environment, ''env8".

The first two images show respectively for PPO and DQN how the action space was changed to allow to choose how much energy to exchange. Two different versions needed to be developed because DQN can't work with continuous action space, therefore a discrete action space with a high number of choices was created. The concept is the same in both cases, where the action value determines the discharge rate, spanning from 0 (no action) to 1 (discharge rate at 1C, e.g. discharge a full battery in 1 hour). It reasonable to expect a further increase of the KPIs values since the battery can be charged/discharged more at each step.

The State Of Charge ripple was mitigated, simply adding a small negative reward in case the State Of Charge derivatives in the current and previous step change sign. Table 8.5 provides the updated main features of ''env8".

Feature name	Value
Variables in the observation space	Current energy price, true energy price of next 20 steps, battery SOC
Action space	Continuous or Discrete with 200 values so that variable amount of energy depending on the action value is exchanged each step..
Buy reward function	Normalized. Proportional to the gross wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the gross wallet gain
Battery overcharge and undercharge penalty	-0.1
SOC ripple correction reward	Depending on the SOC derivative sign

Table 8.5: ''env8" main features

The three KPIs related to the experiment carried out with the ''env8" are shown in Figure 8.20, 8.21, 8.22, and Table 8.6 reports a summary of the results.

Agent	Gross Wallet $	Net Wallet $	Gross Wallet/# of cycles $/#C
DQN ''env8"	124000 ± 49000	-427500 ±100000	825 ± 290
PPO ''env8"	525000 ± 40000	-210000 ± 50000	990 ± 700

Table 8.6: ''env8" testing summary

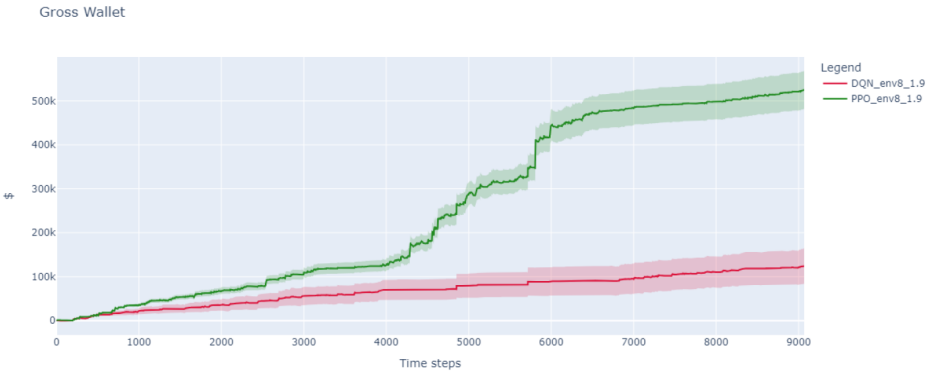


Figure 8.20: Gross Wallet KPI

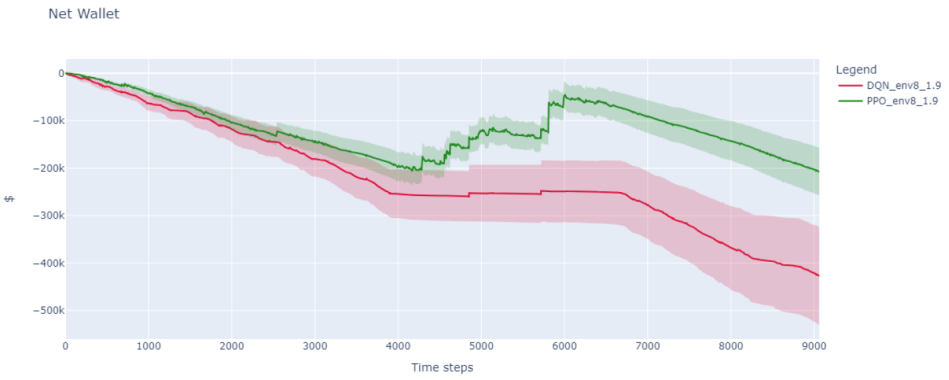


Figure 8.21: Net Wallet KPI

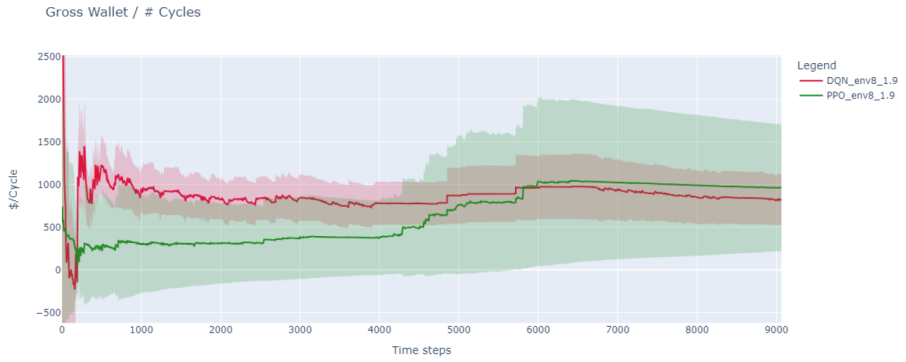


Figure 8.22: Gross Wallet – Number of cycles ratio KPI

The first aspect to observe from these plots is the clear performance superiority of the PPO agent. Moreover, it is possible to observe, as expected, that giving the possibility to the agent to decide how much energy to exchange in a step increases the Gross Wallet KPI in both PPO and DQN agents experiments. This result could be expected mainly because in the previous environment the amount of energy to exchange was fixed and related to 0.4C discharge rate (in 1 hour a full battery looses 40% of is charge), while in ''env8" the amount of energy to exchange could be twice the fixed amount, therefore allowing to double the wallet gain at each step.

Another interesting observation is about the increase of stability over the experiments with different seeds, with both PPO and DQN agents.

In order to complete the analysis of the experiments with this environment the State Of Charge versus energy price plots are shown in Figure 8.23, 8.24

respectively for PPO and DQN best agents testing.

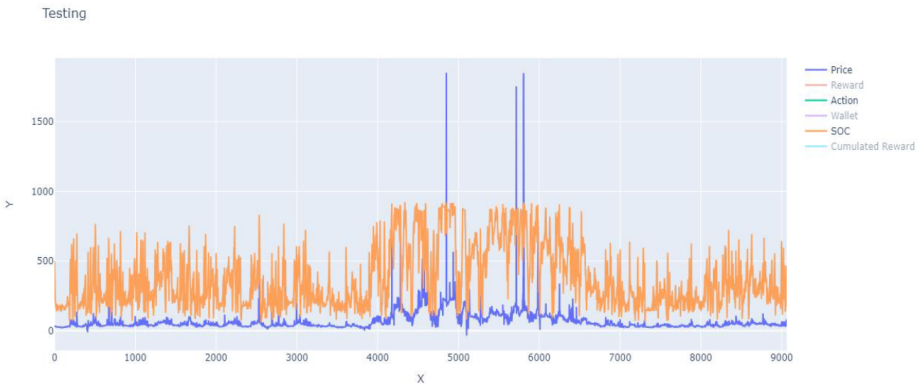


Figure 8.23: SOC vs energy price of best model trained with PPO

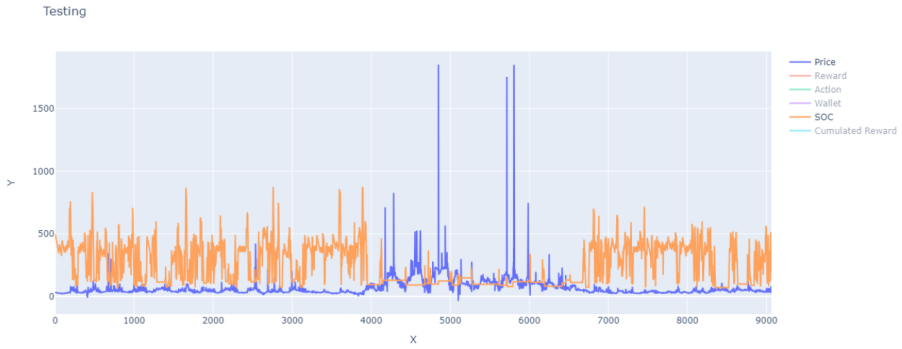


Figure 8.24: SOC vs energy price of best model trained with DQN

From this images it appears clear why the PPO agent is performing much better compared to the DQN one. The latter, compared to the same plot shown for the experiment in ''env6" was able to increase the charge-discharge dept only in low and medium price volatility time frames, keeping the State Of Charge to the minimum during the high volatility time frames. This prevents the DQN agent to exploit the time frame where it is possible to obtain the highest gain for both gross and net wallet.

In total opposition the PPO agent highly takes advantage of the entire high volatility time frame, maximizing the Gross Wallet gain. Therefore it can be concluded that in ''env8", which represents a fairly complete model of

the BESS – grid interaction, PPO overperforms DQN, providing a highly effective trading strategy that maximizes the Gross Wallet.

8.2.5 Testing Env10: environment with battery costs included

After reaching such a positive result with ''env8" and PPO, it was decided to further increase the environment complexity introducing some operational costs into the reward function. So far the Net Wallet was only calculated as a KPI, but it did not affect at all the reward function, thus the agents never took in consideration that for every action they make, on top of the energy cash flow, other costs add up. The different type of costs was already mentioned in previous section, and the important aspect to consider now is that in the reality, since every buy /sell action has a cost, it is not always profitable to purchase or sell energy, and the KPI that matter the most for the BESS owner is the Net Wallet, which represents the net financial gain after the costs of operations. Therefore the following experiment aimed to train the agent to optimize the trading strategy keeping in consideration the most relevant operational costs.

In order to keep the environment fairly simple, only the battery amortization cost was considered in updating the reward function, with the same logic used to compute the Net Wallet. In this experiment it is expected to observe a drop in the Gross Wallet values, an increase of the Gross Wallet – Number of Cycles ratio and positive values of the Net Wallet.

The updated reward function is the result of a long fine tuning in terms of coefficients and logic. Due to time constraints only the version for PPO was developed, therefore the entire experiment was carried out only with PPO. Since the action space is continuous and between -1 and 1, it was necessary to define a continuous interval (between -0.1 and 0.1) for the neutral action.

In both cases the reward was modified including the Net Wallet increment instead of the Gross Wallet increment. A summary of the updated environment (env10) is shown in Table 8.7.

Feature name	**Value**
Variables in the observation space	Current energy price, true energy price of next 20 steps, battery SOC
Action space	Continuous, so that variable amount of energy depending on the action value is exchanged each step..
Buy reward function	Normalized. Proportional to the net wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the net wallet gain
Battery overcharge and undercharge penalty	-0.1
SOC ripple correction reward	Depending on the SOC derivative sign

Table 8.7: ''env10" main features

Figure 8.25, 8.26, 8.27 show the KPIs of all the agents testing and compares the result with the testing in ''env8", while Table 8.8 provides a summary of such results.

Agent	Gross Wallet $	Net Wallet $	Gross Wallet/# of cycles $/#C
PPO ''env8"	525000 ± 40000	-210000 ± 50000	990 ± 700
PPO ''env10"	68500 ± 35000	46500 ± 25000	16900 ± 9000

Table 8.8: ''env10" testing summary

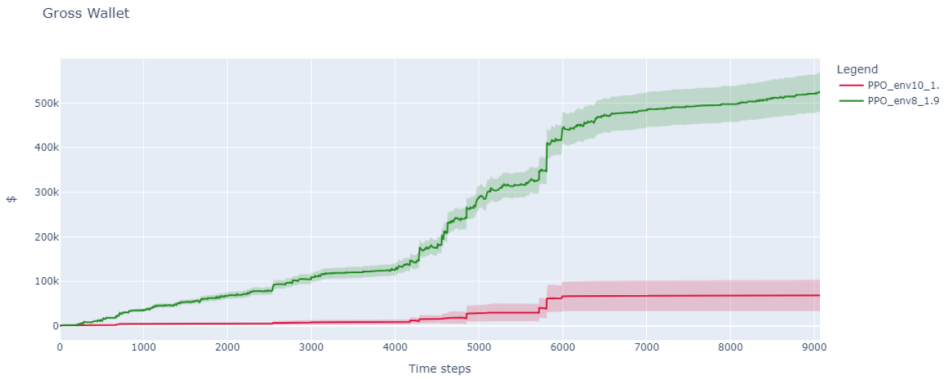


Figure 8.25: Gross Wallet KPI

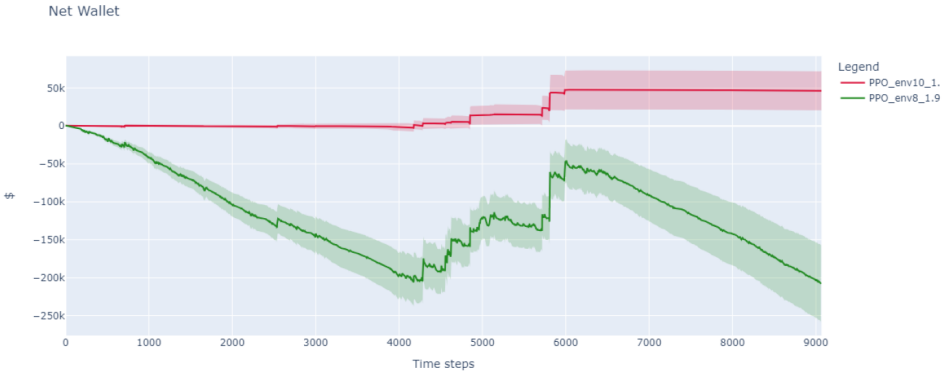


Figure 8.26: Net Wallet KPI

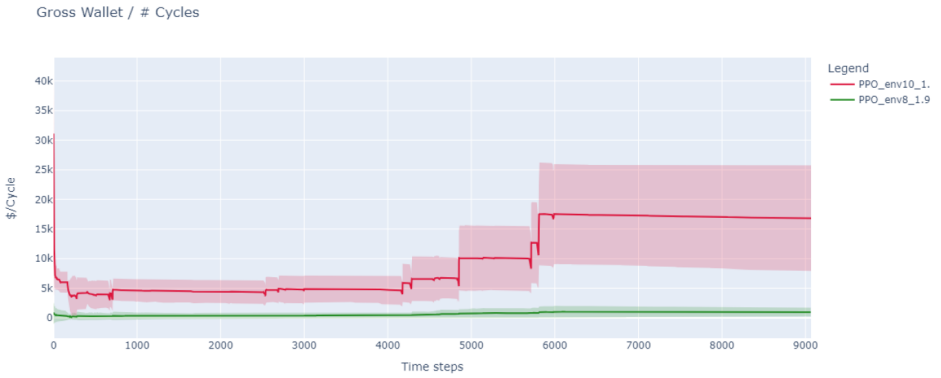


Figure 8.27: Gross Wallet – Number of cycles ratio KPI

Despite the high drop in the Gross Wallet, which can be expected since the reward function is now based on the net wallet increment and as a consequence the agent drastically reduces the buy/sell action to only the most profitable ones, the Net Wallet is highly positive for the first time, and the Gross Wallet – Number of Cycles ratio is 15 times higher. These results confirm again the high effectiveness of PPO in training agents that optimize the BESS operations to maximize the return of investments. Figure 8.28 shows the State Of Charge versus energy price plots for the best agent testing.

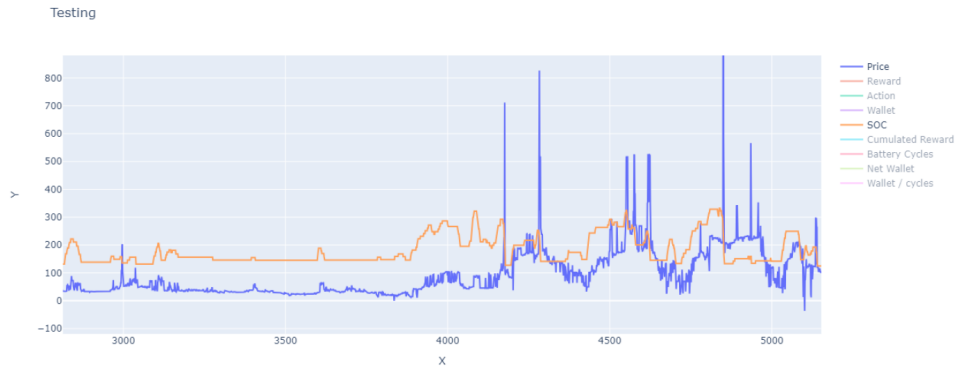


Figure 8.28: SOC vs energy price of best model trained with PPO

The State Of Charge observable behavior is now closer to an optimal trading strategy, where the battery is charged before the energy price increases, and it is discharged during the most relevant peaks. The results so far obtained were very promising, but all the conducted tests were based on an unrealistic

assumption: the energy price forecast is 100% accurate. Such an assumption was used to simplify the environment, but in order to make a more realistic test it was necessary to replace the real future price observation with forecast made by a dedicated model.

8.2.6 Testing Env12: environment with 2h forecasted price ahead in the observations

The following test carried out had therefore this purpose and used the forecasting model trained in the first step of the project. Due to time limitation the conducted test used the price forecasts only during the agent testing, while during the agent training the future price values corresponded to the real price. Despite this situation can be totally real, since the trainings are always carried out on historical data, that the agent is trained to trust the price forecast, but during the test such forecast are not necessarily reliable.

In this scenario might not perform very well, and the comparison with another test where even the price forecasts during training are not accurate might be one of the several option for future work. It is reasonable to expect an overall performance drop of all KPIs with this experiment. Table 8.9 shows the main features of the modified environment, ''env12".

Feature name	Value
Variables in the observation space	Current energy price, forecasted energy price of next 20 steps from trained model, battery SOC
Action space	Continuous, so that variable amount of energy depending on the action value is exchanged each step..
Buy reward function	Normalized. Proportional to the net wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the net wallet gain
Battery overcharge and undercharge penalty	-0.1
SOC ripple correction reward	Depending on the SOC derivative sign

Table 8.9: ''env12" main features

During the this test the training was carried out using ''env10", and the testing using ''env12. Figure 8.29, 8.30, 8.31 show the KPIs of all the agents testing and compares the result with the testing in ''env10", while Table 8.10 shows a summary of such results.

Agent	Gross Wallet $	Net Wallet $	Gross Wallet/# of cycles $/#C
PPO ''env10"	68500 ± 35000	46500 ± 25000	16900 ± 9000
PPO ''env10" forecast ("env12")	38000 ± 28000	21650 ± 21000	11780 ± 14000

Table 8.10: ''env12" testing summary

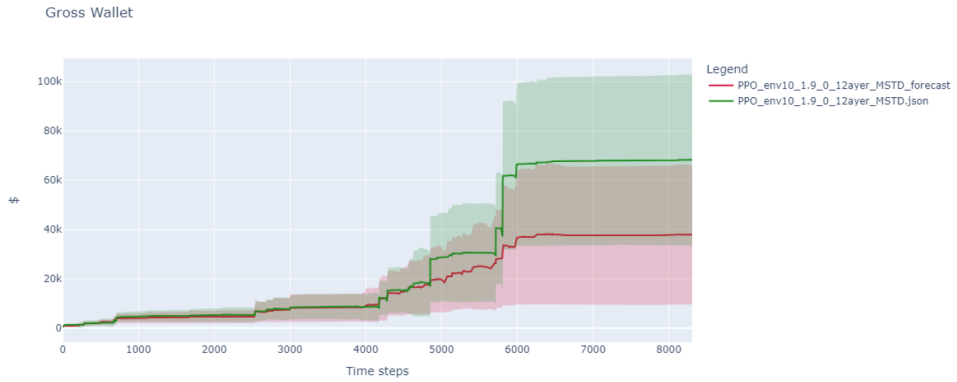


Figure 8.29: Gross Wallet KPI

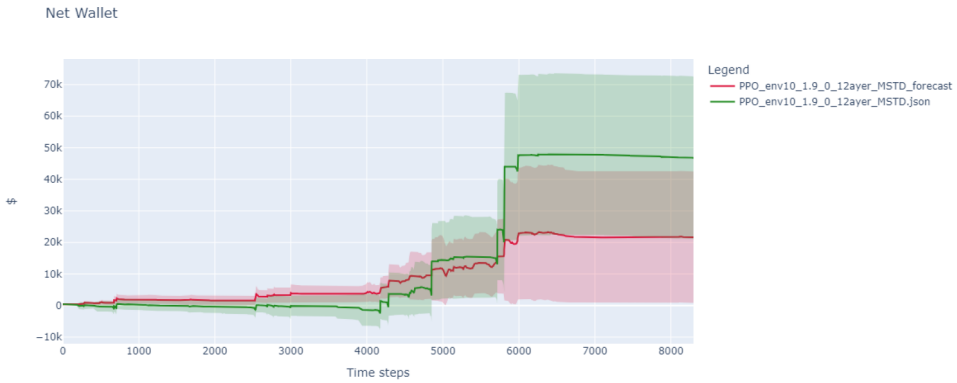


Figure 8.30: Net Wallet KPI

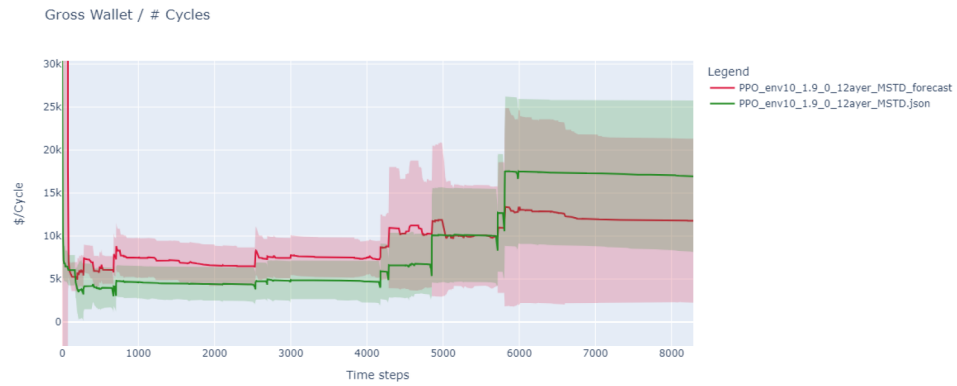


Figure 8.31: Gross Wallet – Number of cycles ratio KPI

As expected a consistent performance drop can be observed, clearly due to

the inability of the forecasting model to predict the price peaks. This consequentially reflects on the inability of the agent to effectively take advantage of the real peaks to maximize the wallet gain.

8.2.7 Testing Env13: environment with only current price and use of a LSTM as policy network

As mentioned this result was expected and it was then decided to explore the use of more complex policy networks, currently used by researchers to process time series. In particular it was decided to explore the use of LSTM and Transformer models. Unfortunately, due to time limitations and due to the unavailability of a transformer implementation with PPO in SB3, it was possible to carry out a test only using LSTM instead of MLP.

In terms of implementation, thanks to SB3 flexibility, it was only necessary to define a different model from the library, and to have in the observation space the energy price of the current step. A new environment, ''env13" was created for this test, and Table 8.11 provides the most relevant features.

Feature name	**Value**
Variables in the observation space	Current energy price, battery SOC
Action space	Continuous, so that variable amount of energy depending on the action value is exchanged each step..
Buy reward function	Normalized. Proportional to the net wallet loss minus a charge bonus
Sell reward function	Normalized. Proportional to the net wallet gain
Battery overcharge and undercharge penalty	-0.1
SOC ripple correction reward	Depending on the SOC derivative sign

Table 8.11: ''env13" main features

Figure 8.32, 8.33, 8.34 show the KPIs of all the agents testing and compares the result with the testing in ''env10", both with real future price values and forecasted values, while Table 8.12 shows a summary of such results.

Agent	**Gross Wallet $**	**Net Wallet $**	**Gross Wallet/# of cycles $/#C**
PPO ''env10"	68500 ± 35000	46500 ± 25000	16900 ± 9000
PPO ''env10" forecast ("env12")	38000 ± 28000	21650 ± 21000	11780 ± 14000
PPO ''env10"	30400 ± 37000	15070 ± 20000	18000 ± 15000

Table 8.12: ''env13" testing summary

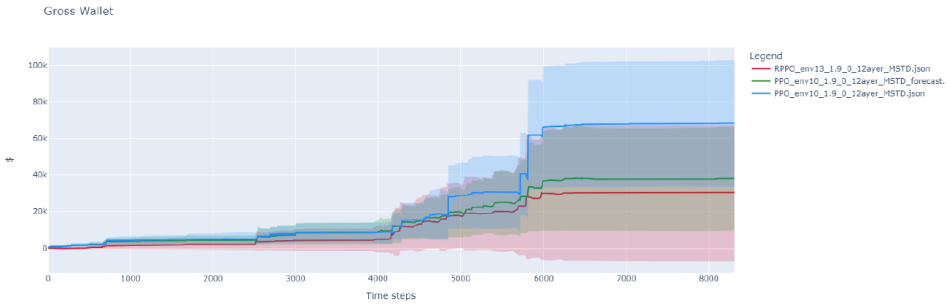


Figure 8.32: Gross Wallet KPI

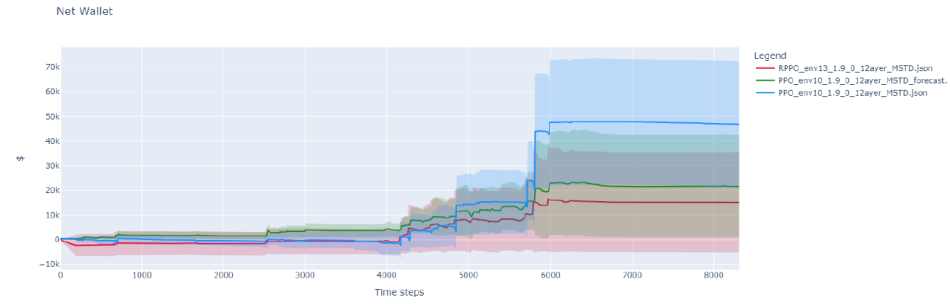


Figure 8.33: Net Wallet KPI

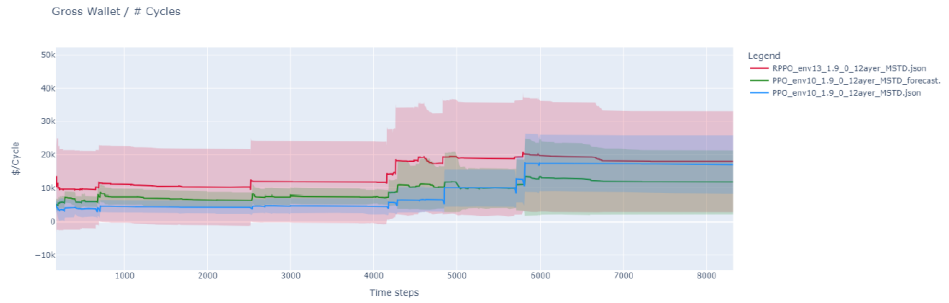


Figure 8.34: Gross Wallet – Number of cycles ratio KPI

It is possible to notice from both Gross and Net wallet KPIs that this agent performs slightly worse than the one tested with forecasted energy price. The Gross Wallet – Number of cycles ratio instead is better than the previous agents. This leads to assume that the new agent is performing a very limited amount of good quality buy-sell actions. Another important aspect to observe is the high variance of all KPIs in the last model, which means

that on different seeds the different trained agents performed significantly different.

Figure 8.35 shows the State Of Charge trend versus the energy price in the testing of the best performing agent. Figure

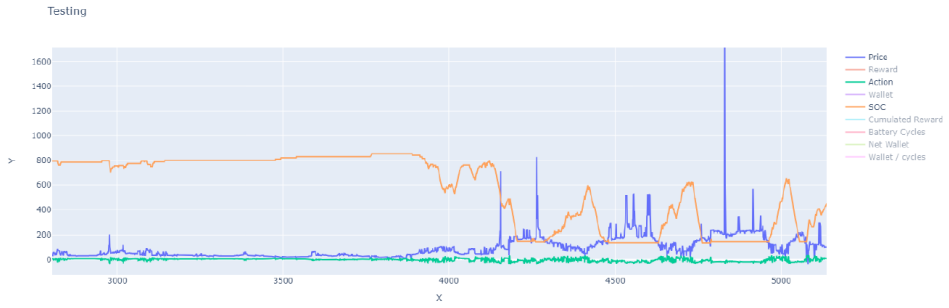


Figure 8.35: SOC vs energy price of best model trained with PPO

This plot is very helpful in understanding that actually the best model in terms of Net Wallet and Gross wallet had only partially an optimal trading strategy; the battery is in fact correctly charged when the price is low, but it is not discharged on the peaks, consistently reducing the potential gain.

This result leads to a first conclusion that even a more complex policy network model like an LSTM is not able to provide an optimal trading strategy having in the observation space just the current energy price and the State of Charge. Nevertheless the there was not enough time to perform an appropriate hyper-parameters fine tuning so it is necessary to run further experiments in orders to confirm the aforementioned first conclusion.

Chapter 9

Conclusions and future work

The research conducted during this project can confirm that PPO stands out as state-of-the-art reinforcement algorithm and that it has a very good potential in being applied to optimize the BESS operations strategy, while DQN does not perform well when the environment complexity increases. Nevertheless much further work it is necessary to proof that PPO can be used to train policy agents that are going to successfully manage the BESS operations strategy in a real environment, since the experiments carried out so far consistently simplified the environment and the action space.

First of all it will be necessary to improve the energy price forecasting model in order to give the agent better quality information to make decisions.

Then it will be necessary to gather all the necessary data to represent the ancillary services prices trend and the complex interaction between the BESS and the grid when an ancillary service transaction occurs. In order to do so the support of professional in the field will be necessary since such data are not publicly available.

Another area of further research, that was just slightly investigated in this project, is the application of complex policy network models such as LSTM and transformers instead of basic MLP to try to improve performances taking advantage of the time series. In this case it will be necessary to spend resources in developing an implementation of PPO compatible with LSTM and transformers, other than sourcing the necessary computational capacity.

It will be also necessary to spend some time fine tuning the hyper-parameters in such experiments In conclusion it can be established that with the right resources it will be possible to unleash all the potential of PPO in training

effective policy agents, and it will be possible to verify if such agents will be performing better than other technologies in optimizing the BESS operations strategy.

Bibliography

[1] Bellman, R. 1957a, Dynamic Programming, ISBN: 069107951x, Princeton University Press.

[2] Bellman, R. 1957b, 'A Markovian Decision Process', Journal of Mathematics and Mechanics, vol. 6, no. 5, pp. 679–84, ISSN: 00959057.

[3] Watkins, C. 1989, Learning from Delayed Rewards, PhD thesis, https://www.researchgate.net/publication/33784417_Learning_From_Delayed_Rewards>.

[4] Werbos, P. 1987, 'Building and Understanding Adaptive Systems: A Statistical/Numerical Approach to Factory Automation and Brain Research', IEEE Transactions on Systems, Man, and Cybernetics, vol. 17, no. 1, pp. 7–20, ISSN: 0018–9472, DOI: 10.1109/TSMC.1987.289329, <https://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-003.pdf>.

[5] "Methods and Apparatus for Reinforcement Learning, US Patent #20150100530A1" , US Patent Office, 9 April 2015.

[6] van Hasselt, Hado; Guez, Arthur; Silver, David (2015). "Deep reinforcement learning with double Q-learning" (PDF). AAAI Conference on Artificial Intelligence: 2094–2100. arXiv:1509.06461

[7] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015). Trust Region Policy Optimization. ArXiv. /abs/1502.05477

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv.org, https://arxiv.org/abs/1707.06347 , arXiv:1707.06347

[9] https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[10] Optimization of a photovoltaic-battery system using deep reinforcement learning and load forecasting – Antonio Corte Real, G.Pontes Luz , J.M.C. Sousa, M.C. Brito , S.M. Vieira

[11] Deep-learning-and reinforcement-learning-based profitable strategy of a grid-level energy storage system for the smart grid - Han G., Lee S., Lee J., Lee K., Bae J.

[12] Reinforcement learning-based optimal scheduling model of battery energy storage system at the building level - Hyuna Kang, Seunghoon Jung, Hakpyeong Kim, Jaewon Jeoung, Taehoon Hong

[13] Demand Forecasting and Resource Scheduling of Independent Energy Storage Market in Power Grid with Deep Learning - Zhiqiang Wang Jin Wang Yueli Zhou Kexin Liu Zheng Weng

[14] Battery energy storage control using a reinforcement learning approach with cyclic time-dependent Markov process – Sara Abedi, Sang Won Yoon, Soongeol Kwon

[15] https://www.visualcrossing.com/weather-data

[16] https://www.nyiso.com/documents/20142/3625950/mpug.pdf