ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

eeh power systems laboratory

Computing Dispatch Plan for Active Distribution Grids using the Soft Actor Critic Algorithm

Leonardo Pagani

July 2022

EEH - Power Systems Laboratory
Swiss Federal Institute of Technology (ETH Zurich)
Physikstrasse 3
8092 Zurich
Switzerland
Tutor: Prof. Dr. Gabriela Hug
Supervisor: Dr. Eleni Stai

Abstract

In this thesis, we evaluate the performances of the Soft Actor Critic algorithm for computing a dispatch plan in a medium size radial distribution grid. Using a reinforcement learning algorithm to solve this task has already been proved to be a valid alternative to the standard optimization method involving the computation of the AC Optimal Power Flow. In particular, it has been shown that the DDPG (Deep Deterministic Policy Gradient) algorithm in the actor critic framework is able to compute a reliable and efficient day-ahead dispatch plan. However, the DDPG algorithm suffers from some instability issues. When training an agent multiple times, with the same data and the same parameters, not all agents reaches the same quality. The goal of this thesis is to show that the state-of-the-art Soft Actor Critic Algorithm can actually achieve a more stable performances in terms of the computation of the dispatch plan. Since the standard SAC algorithm is designed for an online interaction with the environment, we need to design a tailored methodology to compute a day-ahead dispatch plan. In this work, we compute a dispatch plan for a real electrical network, equipped with PV generation plants and with a battery. The computed dispatch plan is then validated using a real-time controller which tries to follow the plan as closely as possible. Finally, the obtained results are compared to those achieved with the DDPG reinforcement learning algorithm and with CoDistFlow, an optimization based scheme method.

Contents

1 Introduction

The deployment of renewable energy has increased significantly in recent times and experts suggest that this growth is not going to stop in the immediate future [1]. The most recent political measures suggest that green renewable energy production is projected to increase in the following years. Furthermore, after the outbreak of the Russia-Ukraine conflict, every country is trying to achieve energy independence, by making investments in the renewable energy sector [2]. Every study suggests that, in the long term, the production of energy by renewable sources will be predominant, as governments are trying to accelerate the transition to green energy [3]. It is clear that electrical operators have to adopt to this major shift. The transition to green energy brings structural changes in electrical grids with a tendency to shift towards distributed generation and decentralized plants [4]. The main challenge arises from the fact that RESs(renewable energy sources), such as PV production and wind energy from turbines, are characterized by an intrinsic uncertainty that makes it difficult to predict the availability and the disposal of energy. Moreover, renewable energy plants are not dispatchable, which means that energy production cannot be increased or decreased at will, depending on the demand on the grid. More energy is available, more energy is produced with the plant output that can only be regulated to a certain extent [5]. When the wind stops blowing or the sun stops shining, no power is produced. Conversely, in favourable weather conditions, wind and PV energy sources may produce more power than is needed: they have to be shut down - renewable power production has to be curtailed - and the opportunity to turn that wind and solar irradiance into useful electric power is lost.

Battery energy storage systems (BESSs) are being considered as a countermeasure for this issue. In [6] and [7], it has been shown that introducing batteries in the electrical grid yields great benefits not only in terms of saved energy, but also in terms of grid stability. Batteries are able to absorb the uncertainty of RESs. Specifically, the batteries can charge to consume energy when there is over-generation and discharge to provide extra energy when the demand exceeds the supply. In order to exploit the full potential of batteries, we can compute optimal consumption/generation plans for aggregated sets of prosumers, which are denoted as dispatch plans and are usually computed one day ahead their application. The idea is to use batteries as a storage device to make up for energy fluctuations. Energy is stored in the battery when the production overcomes the current demand, whereas the stored energy is consumed when current renewable production is not sufficient to meet the demand of the network. During the day that the dispatch plan is applied, the batteries will make sure that a given cluster of prosumers follows the prosumption given by the plan. Computing an accurate day-ahead dispatch plan is crucial not only for the stability and balance of the network but also to reduce the costs of grid operators. Any foreseeable energy surplases or shortages can be sold or purchased in the day-ahead market, which results in reduced cost when compared to energy transactions in the intra-day market [8].

Computing a dispatch plan for a distribution grid with RESs and batteries is a well known problem. Multiple solutions can be found in the literature ([19],[20]). However, computing a dispatch plan with stochastic resources and storage devices is, in general, a difficult problem, as it involves the solution of an AC Optimal Power Flow to satisfy all the constraints. This problem involves the non-linear power flow equations, thus it is a non-convex and hard to solve problem. Several methods proposing relaxation and approximation techniques have been proposed and analyzed. In [21], the CoDistflow Algorithm is introduced to solve the non convex scenario-based AC OPF with intermittent renewable resources and battery storage. An optimized dispatch plan for an entire distribution network is obtained while taking into account grid and battery losses as well as possible realizations of the load and of the stochastic energy resources. In [22], the CoDistFlow algorithm is improved for redispatching, with the plan being updated intra-day to match the most recent loads and PV forecasts. These methods achieve an optimal computation of the dispatch plan, but are computationally expensive. While this approach might still be suitable to small local grids, the calculations with a scaled up grid could take a lot of time. This might be problematic, especially if we want to carry-out an intra-day optimization of the dispatch plan.
An alternative way to solve the problem of computing a day-ahead dispatch plan is to make use of state of the art reinforcement learning algorithms. In the past years, machine learning techniques have gained importance. Reinforcement Learning (RL) algorithms, in particular, have been proposed to solve various control problems, including the calculation of schedules in the energy sector. The availability of data productions and loads of certain network can be used to train a RL agent to learn the computation of a dispatch plan for certain load and source conditions. This new horizon to be explored is possible thanks to the success and the development of deep learning, which has led to the extension of the Reinforcement Learning framework to continuous learning space. The state-of-the-art RL actor critic algorithms are able to learn both the actions and the value functions of a continuous environment [14], which is exactly what is needed in our dispatch plan learning framework. In [24] the Soft Actor Critic algorithm is used to learn the optimal charging policy of an electric vehicle. In [25], SAC is used to adaptively select and route information over multiple paths in mmWave networks to achieve a desired source-destination rate. Deep Reinforcement Learning algorithms have already been experimented in many other scheduling problems. Clearly, training a RL agent generally requires a considerable amount of time, especially with the simulation environment being a real distribution network with all its voltages and currents. However, a RL agent can be trained offline in advance, with the dispatch plan predictions only computed day-ahead. Using a RL agent to perform a similar task is an innovative way to get around the intrinsic complexity of the optimization problem. The optimal dispatch plan is actually computed by avoiding any complicated approximations and simulations, but only by giving the proper feedback to the neural networks of the model. This framework extremely simplifies the problem but may be vulnerable to imprecision in the physical modelling of the network.

The choice of the reward is critical for the success of the network. In [26], the DDPG algorithm, introduced in [23], is able to correctly obtain an efficient dispatch plan despite suffering from little stability and inconsistency issue. Our goal in this thesis is to apply the Soft Actor Critic algorithm, proposed by [15]. In this thesis, a SAC algorithm is developed to train an agent to compute a proper dispatch plan for any given scenario. The obtained results will then be compared with those obtained with the DDPG algorithm.

This project is structured as follows: *Chapter 2* provides an introduction on Markov Decision Processes and Reinforcement Learning. We will provide more details on the Soft-Actor-Critic (SAC) algorithm, as introduced by [15]. *Chapter 3* derives the problem formulation, formalizing the MDP and the optimization problem. The algorithmic solution to the problem is formalized in *Chapter 4*, where we delve deeper into the algorithms at training and inference time. In *Chapter 5*, we specify the structure of the grid used for our experiments and we formalize our hypothesis on the battery. Finally, *Chapter 6* contains an analysis of the results as well as a comparison to the performances of DDPG algorithm.

2 Reinforcement Learning

In this section we will provide a brief introduction on the general reinforcement learning framework. We will start from the Markov Decision Processes which serve as a substratum to reinforcement learning and we will then proceed to explore the most advanced procedure to solve the reinforcement learning problem.

2.1 An introduction to Reinforcement Learning

Reinforcement learning consists in training an agent by gathering feedbacks through the interaction with the environment. Reinforcement learning differs from *supervised learning* as it does not learn from a set of labeled examples provided by a knowledgeable supervisor. It is also different from *unsupervised learning*, which is typically about finding structure hidden in collections of unlabeled data. Instead, an RL agent is trained through a trial and error process, similar to a human [9]. A RL agent, by receiving feedbacks from the environment, is able to learn the optimal actions to take. Clearly, such an agent must be able to sense the state of the environment to some extent and must be able to take actions that affect the state. The agent also must have an objective relating to the state of the environment. Reinforcement learning algorithms have made huge progress in recent years, especially since the development of deep learning based actor critic algorithms. The most noticeable application of RL are in control and decision making problems. In 2017, the Deepmind RL based program *Alphazero* achieved an unprecedenedt level of play in the board games of chess, shogi and go, convincingly defeating world champions in each case [10].

2.2 Agent Environment Interface

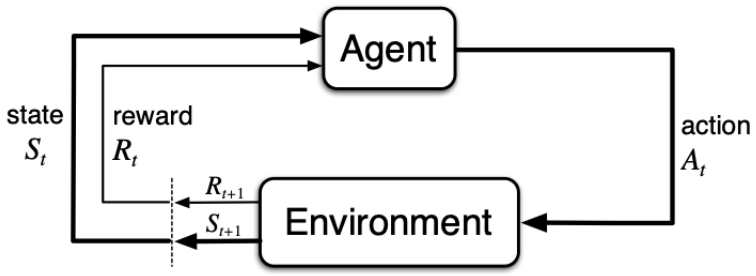

Figure 1: Interaction between agent and environment in Reinforcement Learning

While in standard supervised learning a model is trained on a provided labeled dataset, RL is a peculiar machine learning framework in which an agent is trained by interacting with the outside environment. As outlined in the previous paragraph, the reinforcement learning problem is meant to be a straightforward

framing of the problem of learning from interaction to achieve a goal. In every reinforcement learning problem we can therefore distinguish between an *agent* and an *environment* [9]. Specifically, the decision-maker is called the *agent*. As illustrated in *Figure 1*, at timestep t, the agent picks an action A_t, based on the current state of the *environment* S_t. For example, in chess, the action taken is a move made by the learner, while the state is the current position on the board. As the agent takes action A_t, the environment moves to a subsequent state S_{t+1}, while the learner receives a feedback reward R_t. Again, in chess, after a move is played, a new position (state) is reached. A generic chess position can for example be evaluated by certain parameters (material point values, king safety, pawn structure, activity). According to this evaluation a reward R_t can be fed to the agent to provide him feedback on his precedent behaviour.

2.3 Markov Decision Process

Let us now formalize our reasoning by understanding the terminology and the substratum of reinforcement learning. In this case we will simplify our study by analyzing *finite Markov Decision Processes*, where the state space and the action space are finite and countable. It is worth noting that this simplification better allows to grasp the core of a reinforcement learning task. As we will see, a more challenging task for RL is to deal with continuous action and state spaces. Let us first define S as the state space of the environment and A as the action space space available to the learner. As outlined before, at any time t, the agent might choose an action $a_t \in A$. The environment then moves to state s_{t+1}, while the agent receives reward r_t. It follows that every Markov Decision Process is characterized by the tuple $(S,A,\ p,r)$, where p indicates the so called *transition probabilities*. In general the dynamics of an MDP are stochastic and not deterministic. That means that, for any state s and action a taken at the current step, the next state of the environment is described by a probability distribution. $P(s_{t+1} \mid s_t, a_t)$ denotes the probability of the environment ending up in a certain future state s_{t+1}, given that it was in state s_t and that the agent took action a_t.

A crucial property defines MDPs. A state $s \in S$ in every Markov Decision Process depends only on the previous state and not on the past states. In formulas:

$$P(s_{t+1} \mid s_t) = P(s_{t+1} \mid s_1, \ldots, s_t)$$

If the system is Markovian, the history is all encompassed in the current state. This one step is a game-changer for computational efficiency. The Markov Property underpins the existence and success of all modern reinforcement learning algorithms [11].

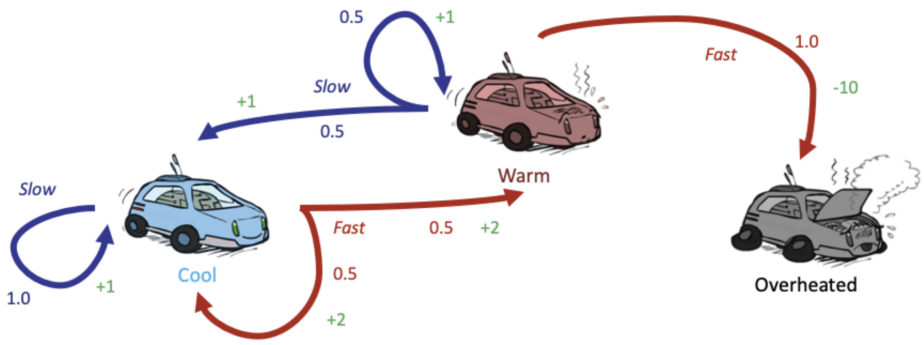

Figure 2: An example of a Markov Decision Process. The state space S = (Cool, Warm, Overheated) and the action space A = (Slow,Fast). Transition probabilities and rewards are also specified.

2.4 Policy, Value Function and Q-values

A policy π is a function mapping each state to an action (deterministic policy), or to a probability distribution over the actions (randomized policy). The policy describes the strategy based on which the algorithm determines an action to take in a certain state s [9]. Solving a Markov Decision Process means to determine the optimal policy function in terms of future discounted reward. An additional concept to add is that of *discounting*. The discount rate γ determines the present value of future rewards. In particular, our objective at time t is to maximize the following quantity G_t: Here γ lies in the interval [0,1] and repre-

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \quad = \quad \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

sents the fact that a future reward is not worth as much as an immediate reward. If $\gamma = 0$, the agent is *greedy* and looks to maximize only the immediate reward. In order to find an optimal policy, we can define a *value function* $v_\pi(s)$ over the state space S which basically tells us how good a certain state is in terms of future discounted reward. Since the reward depends on the actions it will take in the future, the value function is defined with respect to a certain policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right]$$

The expected value here captures the possibility that the policy is randomized over the actions. Similarly, we call *action-value function* q_π (s,a), the value obtained by taking action a in state s and then following policy π.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t{=}s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t{=}s, A_t{=}a\right]$$

Action-value functions are also called *q-values*. The value functions and the q-values for a certain policy π can be estimated from experience with *Monte Carlo Methods* that involve averaging over many random samples [9].

2.5 Q-learning

Let us first discuss a simple model free algorithm, that can be used to solve finite MDPs. The *Q-learning* algorithm assigns a value to each possible pair of state $s \in S$ and action $a \in A$. Specifically we will refer to this value as *state-action value*, or simply *Q-value*.

Pseudocode of Q-learning algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

At every step, the q-value function is updated by a moving average, characterized by the parameter α ($\alpha = 0.5$ is a typical value). Q-learning is an off policy algorithm as it estimates the discounted reward for state-action pairs assuming a greedy policy were followed despite the fact that it is not necessarily following a greedy policy. It has been shown [12] that Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely.

Q-learning is a *tabular* reinforcement learning algorithm, as it requires to store the q-values in an array. That makes Q-learning effective whenever we deal with

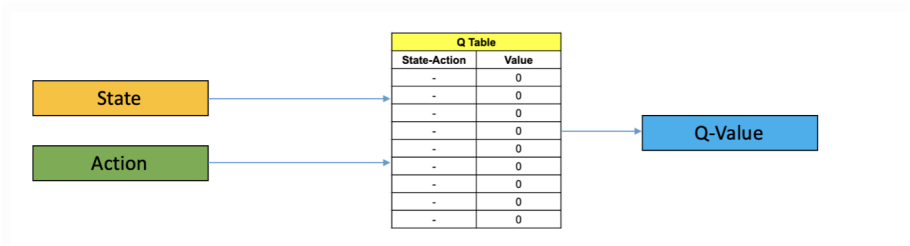

Figure 3: Representation of tabular Q-learning algorithm. The algorithm requires to store all the q-values in a table

small state and action spaces and therefore with a limited number of q-values. Storing and optimizing on a different q-value would be computationally expensive with large state and action spaces and practically impossible when S and A are continuous.

2.6 Function Approximation Reinforcement Learning

We have so far assumed that our estimates of value functions are represented as a table with one entry for each state or for each state–action pair. This is a particularly clear and instructive case, but of course it is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to fill them accurately. In other words, the key issue is that of generalization. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function, q-values) and attempts to generalize from them to construct an approximation of the entire function. Historically, linear function approximators have been used to solve this task. Nowadays, state-of-the-art algorithms use neural networks to construct value function/ q-values approximators (Deep Q-learninig [13]).

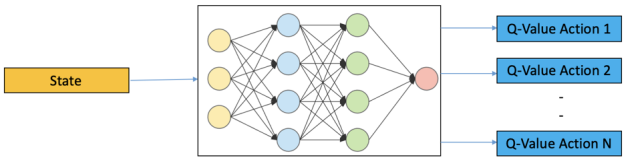

Figure 4: In Deep Q-learning a neural network learns state-action values

2.7 Actor-Critic Algorithms

Until now, we have been exploring methods that learns state-action pairs and use them directly to compute the optimal policy π. With *Actor-Critic* algorithms we try to represent the policy directly with its own weights [9]. Actor-Critic methods feature separate structures: the *actor* is used to select the actions - optimizing the actor means to optimize the policy π. The *critic* evaluates the actions taken by the actor by learning value functions of q-functions. The *critic* usually takes as input the action taken from the actor a_t, as well as the current state s_t to learn the q-value q(s_t,a_t) On the other hand, the actor input is given only by the current environment state s_t as it aims at learning $\pi(. \mid s_t)$ The idea of this approach is to learn simultaneously a policy π and an approximation of the Q-value. The actor improves its policy through the score received by the critic. The critic itself improves its approximation through the reward the agent gains from the environment [14].

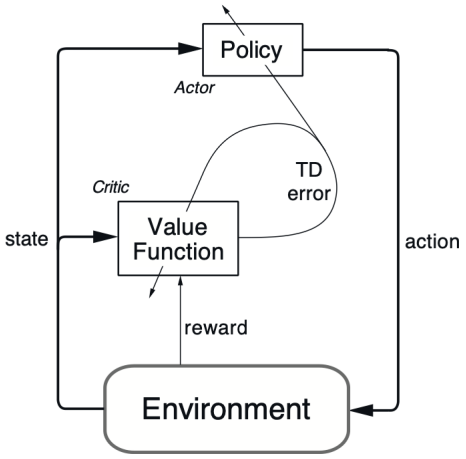

Figure 5: The actor-critic architecture

2.8 SAC algorithm

In this section we will go into the details of the Soft-Actor-Critic algorithm, as introduced by [15]. Soft Actor Critic can generally be used for continuous state and action spaces. The main distinguishing idea behind SAC is that it uses a modified RL objective function. It not only tries to maximize the lifetime expected reward but it also seeks to maximize the entropy of the policy. Let π denote our our policy, then our objective function can be expressed as:

$$\sum_{t=0}^{T} \mathbb{E}[(r(s_t, a_t)) + \alpha H(\pi(. \mid s_t)]$$

Here of course π denotes a stochastic policy depending on the current state s_t. It is therefore possible to calculate an entropy for the policy distribution. The temperature parameter α determines the relative importance of the entropy regularization with respect to the maximization of the reward, thus controlling the stochasticity of the optimal policy. The SAC algorithm objective function is a tradeoff between exploration and exploitation of the policy. This tradeoff is controlled by the parameter α, with higher α corresponding to more exploration, and lower α corresponding to more exploitation [16].

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

Figure 6: Pseudocode of SAC algorithm

Large continuous state and action spaces require to perform a practical approximations both for the Q-functions and the policy. In summary, SAC makes use of three types of network: a state value function V parametrized by ψ, a soft Q-function parametrized θ and a policy function π parametrized by ϕ. As discussed before, in the context of function approximation, the use of neural network is ideal to represent the complexity of those functions. In principle there is no need to create a different approximator for the state-value which is related to the Q-functions and to the policy. The author [15], however, suggests that introducing a separate network to represent the value function can stabilize the training. The general idea behind SAC is to alternate between the optimization of all the networks via gradient descent steps. At every step the SAC algorithm samples a minibatch of transitions from sample buffer $\mathcal{D}$. This minibatch is then used to perform an optimization step for all of the previously mentioned network. Let us now illustrate the optimization process for all of the networks. First, the value network V is trained by minimizing the following cost:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right] \right)^2 \right]$$

The idea behind this cost function is that we want to minimize the squared difference between the prediction of our value function and the expected prediction of the Q-function plus the entropy of the policy π_ϕ. Here the definition of entropy of a distribution is used. Then the Q-network is trained to minimize the following cost:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right]$$

with

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right],$$

Again, the idea behind this objective function is very simple. We want to minimize the squared difference between the predicted q-value and the immediate time-step reward plus the expected value function of the next state [16]. The update makes use of a target value network $V_{\overline{\psi}}$, where $\overline{\psi}$ can be an exponentially moving average of the value network weights, which has been shown to stabilize training [17].
Finally, we need to optimize our policy function π_ϕ, that can be learned by

directly minimizing the following Kullback-Leiber divergence:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_\phi(\cdot|\mathbf{s}_t) \,\middle\|\, \frac{\exp\left(Q_\theta(\mathbf{s}_t, \cdot)\right)}{Z_\theta(\mathbf{s}_t)} \right) \right]$$

The idea is to minimize the distance between the policy distribution and the distribution of the exponentiation of our Q Function normalized by another function Z. Since the target density involves the Q function, which is represented by a neural network, we can apply the *reparametrization trick* to obtain a lower variance estimator of the optimal policy. Following [15], we first consider an unbounded Gaussian as the action distribution. Then, in order make sure that the actions are bounded to a finite interval, we apply a *squashing function*, hyperbolic tangent $\tanh(u)$. Since this function is applied element-wise, our variable is now restricted in the interval (-1,1). Overall, to compute the log-probabilities in closed form we can apply the following reasoning. Let $u \in \mathrm{R}^D$ represent an unbounded random variable and $\mu(u \mid s)$ be the corresponding density function. Then, we can apply a squashing function to create a bounded variable $a \in \mathrm{R}^D$ by setting $\mathrm{a} = \tanh(u)$. To compute the log probabilities of the sampled actions, which are necessary to calculate the KL divergence, we can use the change of variable formulas for multi dimensional random variables. In particular, since $\frac{da}{du} = 1 - \tanh^2 u$, we get that:

$$\log \pi(\mathbf{a}|\mathbf{s}) = \log \mu(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log \left(1 - \tanh^2(u_i)\right)$$

It is worth noting that generally speaking not all action spaces are bounded between (-1,1). That is why it might be necessary to normalize the sampled batch, before applying gradient descent steps.

3 Problem Formulation

In this section we will formulate the problem, delving deeper in our proposed approach to compute the dispatch plan. We will analyze all the variables that come into play and formulate a proper objective function, upon which we will construct our reward. *Dispatching* means to commit day-ahead for the next day a plan of positive or negative power values at the Point of Common Coupling (PCC) with the main grid. Our goal is to predict how much power we need from the upstream grid the next day.

A *dispatch plan* is effective when the real-time control algorithm can take feasible battery charge / discharge decisions with which:

- The realized dispatch plan is very close to the planned one.
- The grid constraints are satisfied.

The problem of dispatching has been effectively solved with CoDistFlow [21] which:

- solves iteratively a multi-period scenario-based AC OPF
- yields a solution satisfying the exact AC power flow equations for all scenarios
- is computationally efficient

3.1 System Modelling

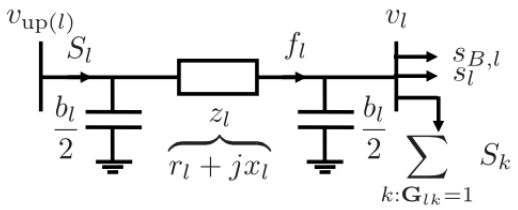

Figure 7: Representation of the π model of a line

Our experiment involves training the model on a real Swiss distribution grid. We will use data of loads and PV productions from a simple radial distribution network. To understand the simulation of the environment, we have to formalize the modelling of the network itself. We consider a balanced and transposed radial network where the Point of Common Coupling between the local distribution network and the upstream power grid is at index 0. We will represent the distribution lines by their π - models, as represented in *Figure 7*. Following the

notation of [21], the node at the top of the line l (i.e., electrically closer to the PCC),is denoted as *up(l)*, and the node at the bottom as l. The adjacency matrix G is characterized by $\mathrm{G_{k,l}} = 1$ for two buses $k, l \neq 0$, if k = *up(l)*, otherwise $\mathrm{G_{k,l}} = 0$. We can now provide the constraints for a scenario-based AC OPF for a radial distribution network with intermittent renewable energy sources and battery storage. We will use the indexes t, d, l to indicate respectively a generic time-step, scenario and line. $\forall t, d, l$:

$$P_{\mathrm{l}}{}^{\mathrm{d}}(t) = \sum_{k:G_{lk}=1}(P_{\mathrm{k}}{}^{\mathrm{d}}(t)) + p_{\mathrm{l}}{}^{\mathrm{d}}(t) + p_{\mathrm{B,l}}{}^{\mathrm{d}}(t) + r_{\mathrm{l}} f_{\mathrm{l}}{}^{\mathrm{d}}(t)$$

$$Q_{\mathrm{l}}{}^{\mathrm{d}}(t) = \sum_{k:G_{lk}=1}(Q_{\mathrm{k}}{}^{\mathrm{d}}(t)) + q_{\mathrm{l}}{}^{\mathrm{d}}(t) + q_{\mathrm{B,l}}{}^{\mathrm{d}}(t) - (v_{\mathrm{up(l)}}{}^{\mathrm{d}}(t) + v_{\mathrm{l}}{}^{\mathrm{d}}(t))b_{\mathrm{l}}/2 + x_{\mathrm{l}} f_{\mathrm{l}}{}^{\mathrm{d}}(t)$$

$$f_{\mathrm{l}}{}^{\mathrm{d}}(t) = (||S_{\mathrm{l}}{}^{\mathrm{d}}(t) + j\frac{v_{\mathrm{up(l)}}{}^{\mathrm{d}}(t) b_{\mathrm{l}}}{2}||)^2 / v^{\mathrm{d}}{}_{\mathrm{up(l)}}(t)$$

Here:

- $P_{\mathrm{l}}{}^{\mathrm{d}}(t)$ and $Q_{\mathrm{l}}{}^{\mathrm{d}}(t)$ denote the direct sequence of active and reactive power entering from bus *up(l)* at time t and for scenario d.
- $P_{\mathrm{k}}{}^{\mathrm{d}}(t)$ and $Q_{\mathrm{k}}{}^{\mathrm{d}}(t)$ denote the direct sequence of active and reactive power entering from bus *up(k)* $= l$ at time t and for scenario d.
- $p_{\mathrm{l}}{}^{\mathrm{d}}(t)$ and $q_{\mathrm{l}}{}^{\mathrm{d}}(t)$ denote the active and reactive power injection for scenario d and time t.
- $p_{\mathrm{B,l}}{}^{\mathrm{d}}(t)$ and $q_{\mathrm{B,l}}{}^{\mathrm{d}}(t)$ are the active and reactive power per battery connected at node l, at time t and scenario d.
- $f_{\mathrm{l}}{}^{\mathrm{d}}(t)$ denote the square magnitude current through the longitudinal impedance of the line.
- r_{l}, b_{l} and x_{l} are the direct sequence longitudinal resistance, reactance and shunt susceptance. These parameters are constant with respect to the scenario and to the time-step.
- $v_{\mathrm{l}}{}^{\mathrm{d}}(t)$ is the square magnitude of the direct sequence voltage at node l at time t and for scenario d.

In the formulation of the AC OPF equations there are also some electrical constraints to keep into account:

- ${v_{\mathrm{lb}}}^2 \leq v_{\mathrm{l}}{}^{\mathrm{d}}(t) \leq {v_{\mathrm{ub}}}^2$
- $SOC^{\mathrm{B}}{}_{\mathrm{min}} < SOC^{\mathrm{B,d}}(t) < SOC^{\mathrm{B}}{}_{\mathrm{max}}$

The idea behind these constraints is to bound the voltages and batteries' state of charge. In the original AC OPF formulation [21], multiple constraints on the

power factor and on the ampacity are considered. Here, to keep the formulation simple, we decided to ignore these constraints to keep the solution flexible enough.

The non convex AC OPF is formulated as a non convex optimization problem. Thus, the initial version of the objective function to minimize is the following:

$$\mathrm{c}(t) = \mathrm{w}_1 \sum_{d,t}(\lambda_\mathrm{d}|{Q^\mathrm{d}}_\mathrm{PCC}(t)|)+w_2 \sum_{d,t}(\lambda_\mathrm{d}|{P^\mathrm{d}}_\mathrm{PCC}(t)|)+w_3 \sum_{d,t}(\lambda_\mathrm{d}{P^\mathrm{d}}_\mathrm{PCC}(t))+ w_4 \sum_{d,t} \lambda_\mathrm{d}||({S^\mathrm{d}}_\mathrm{PCC}(t) - S^\mathrm{DP}(t))|| \quad (1)$$

The objective function is constructed such that the optimal solution trades-off the following four objectives:

1. Maximize the power factor at the PCC by minimizing the reactive power at this bus.
2. Reduce the active power exchange with the external upstream grid.
3. Maximize the power export to the main grid.
4. Minimize the error between the dispatch plan and the optimal power at the PCC for every scenario.

The importance of each objective is measured by each weights $\mathrm{w}_i, \forall i = 1, .., 4$. Furthermore, the objective function is a weighted average over the scenarios. The probability of occurrence of each scenario λ_d acts as a weight. In our formulation we will assume that all scenarios are equally possible and thus we will get rid of the weights λ_d.

Let us now define which variables should we optimize and which parameters are known in advance.

Known parameters

- All the power injections (PV productions and loads), ${\mathrm{p}^\mathrm{d}}_\mathrm{l}(\mathrm{t})$ and ${\mathrm{q}^\mathrm{d}}_\mathrm{l}(\mathrm{t})$ at time t and scenario d.
- Parameters of the line: resistance, susceptance, battery resistance.
- Lower and upper bounds on the voltages.
- Lower and upper bounds on the state of charge of the batteries.

Optimization variables

- Power ${\mathrm{S}^\mathrm{d}}_\mathrm{PCC}(\mathrm{t})$ at the PCC.

- Dispatch plan $S^{DP}(t)$.
- Voltages at the nodes $v_l{}^d(t)$.
- Magnitude of the current flowing through the longitudinal impedances of all lines
- Charging / Discharging power of each battery B, at node l, $p^d{}_{B,l}(t)$ and $q^d{}_{B,l}(t)$

To apply a Reinforcement Learning algorithm we need to fully define the underlying Markov Decision Process. We will first derive the state space S and the action space A. We will then show how to compute the reward.

3.2 Time Discretization and Training Data

To train the agent accurately, multiple thousands of scenarios were used. The set of scenarios was constructed prior to this thesis based on historical data, as explained in [21]. A 24 hours day is discretized in 96 time-steps. Thus, we compute a dispatch plan every quarter of an hour. From now on, we will use the index k to refer to a generic time-step. Each training scenario contains the trajectories of the loads and of the PV production of the network. Generally speaking, our model is more sensible to the PV production data, whose value are multiple times higher than those of the loads. From *Figure 9* we can observe that the PV plant of the network mainly produces energy during the day.

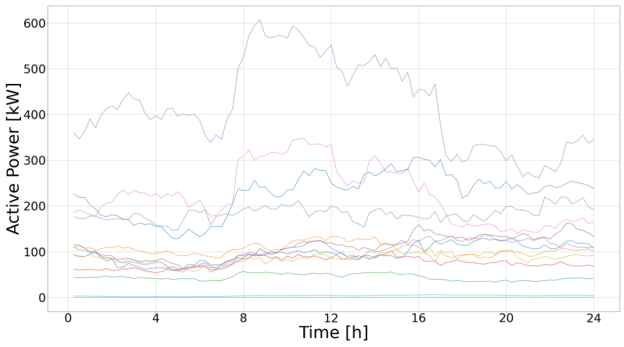

Figure 8: Load trajectories for different scenarios

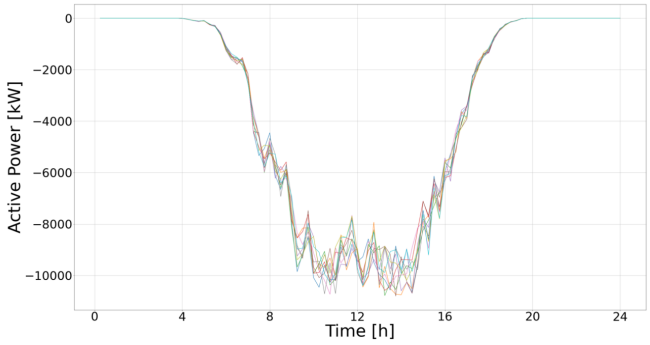

Figure 9: PV production trajectories for different scenarios

3.3 State space

For each time interval k and for each scenario d, we can define the state of the environment ${s_k}^d$. The state is constructed as a vector with the following components:

- k: Time-step (index of a quarter of an hour).
- $\mathrm{SOC^d_B}$*(k)*:State of charges for each battery B.
- $\mathrm{V^d}$*(k - 1)*: Vector of bus voltages, obtained after the actions performed at time-step *(k - 1)*
- $\mathrm{p_l^d}$*(k)*: Vector with each entry being aggregated uncontrollable active power injection/extraction at the respective bus l (forecasts).
- $\mathrm{P^d_{PCC}}$*(k - 1)*: Actual active power at PCC, obtained after the actions performed at time-step *k - 1*
- $\mathrm{Q^d_{PCC}}$*(k - 1)*: Actual reactive power at PCC, obtained after the actions performed at time-step *k - 1*

3.4 Action space

The agent will decide on the active and reactive power at the PCC and on the charging or discharging rate of the batteries. Therefore, in each time interval K, the following set of actions are decided:

- $\mathrm{p^d_B}$*(k)* : Vector consisting of the charging/discharging active power of each battery B.
- $\mathrm{q^d_B}$*(k)*: Vector consisting of the charging/discharging reactive power of each battery B.

- $P^{d}_{DP}(k)$: Planned active power dispatch of PCC node.
- $Q^{d}_{DP}(k)$: Planned reactive power dispatch of PCC node.

3.5 Reward function

Since we want to compute a day-ahead dispatch plan, we have to simulate the interaction with the environment and the reward function. The reward at time-step k depends not only on the action taken at time-step k, but also on the next state of the environment. Our reward is computed as the negative of the instantaneous cost function detailed in (1). In particular for each time-step k:

$$r_k = - w_1(Q_{PCC}(k)) - w_2(|P_{PCC}(k)|) - w_3(P_{PCC}(k)) - w_4||(S_{PCC}(k) - S^{DP}(k))|| - w_5 \sum_i \max({v_{lb}}^2 - {v_l}^d(k), 0, {v_l}^d(k) - {v_{ub}}^2) \quad (2)$$

Solving an optimization problem via a reinforcement learning algorithm yields, however, a little inconvenience. In particular, we cannot force the actor neural network to take actions that satisfy the constraints. We have included the constraints on the bus voltages in the cost function so to give the actor proper feedback whenever a violation occurs. Here, v_{lb} and v_{ub} represent the lower and upper safety bound on the voltages of the network.

4 Application of the SAC algorithm

Algorithm 1 SAC training

Input: initial policy parameters ϕ, Q-function parameters θ_1 and θ_2, value function parameters ψ, empty replay buffer $\mathcal{D}$, update parameter τ.
1.Initialize critic networks $Q_1(s,a)$, $Q_2(s,a)$, actor π(s) and value network with weights θ_1, θ_2 , ϕ and ψ.
2. Initialize target networks $Q_{targ,1}(s,a)$, $Q_{targ,2}(s,a)$ with θ_1, θ_2.
3. Initialize empty replay buffer $\mathcal{D}$.
4. For $d = 1, \ldots, \#scenarios$:
5. Receive initial state observation ${s_1}^d$.
6. For $k = 1, \ldots, 96$:
7. Select action ${a_k}^d$ from actor network π_θ.
8. Correct battery actions $\mathrm{p_B}(k)^d$and $\mathrm{q_B}(k)^d$ if necessary.
9. Apply action ${a_k}^d$ to the environment.
10. Simulate ${s_{k+1}}^d$ via LoadFlow, obtaining $\mathrm{P^d_{PCC}}(k)$, $\mathrm{Q^d_{PCC}}(k)$.
11. Compute reward ${r_k}^d$.
11. Store transition (${s_k}^d$, ${a_k}^d$, ${r_k}^d$, ${s_{k+1}}^d$) in $\mathcal{D}$.
12. Sample a minibatch of transitions from $\mathcal{D}$.
13. Perform gradient descent step optimization as follows:
14. Value network update: $\psi \leftarrow \psi - \lambda_\mathrm{v} \nabla_\psi J(v)_\psi$
15. Critic 1 network update: $\theta_1 \leftarrow \theta_1$ - $\lambda_\mathrm{q} \nabla_{\theta_1} J(q)_{\theta_1}$
16. Critic 2 network update: $\theta_2 \leftarrow \theta_2$ - $\lambda_\mathrm{q} \nabla_{\theta_2} J(q)_{\theta_2}$
17. Actor network update: $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J(\pi)_\phi$
18. Target update: $\psi_{targ} \leftarrow \tau\psi + (1-\tau)\psi_{targ}$

4.1 Training process

We will now summarize how the training loop works. Training an actor critic agent requires in general a lot of time, in order to guarantee the convergence for the parameters of the neural networks. Our training data consists in 15000 scenarios with loads trajectories and PV production plant estimates. At first, we have to initialize the parameters of our networks. Let us recall that in the SAC framework, we have to train: a value network, two critic networks, an actor network as well as the target value network, as outlined in [15]. All of these networks feature multiple hyperparameters to tune. For every scenario d in the training dataset, we give the initial state of the environment ${s_1}^d$ as an input to the actor network. With its output ${a_1}^d$ we can compute our reward and simulate the next state of the network via a LoadFlow calculation. It is worth noting that the LoadFlow takes as input the battery actions provided by the actor network and outputs the actual power at the PCC. Here, *pandapower* library is used to perform the LoadFlow calculation. Before applying the actions to the environment, we first have to correct the battery actions. The charging and discharging power of the battery in the network are actually limited, and

therefore have to be bounded. In fact, we have to ensure that batteries are not charged or discharged further if they are already full or empty. In particular, we perform the following transformation:

$$p_{\mathrm{B}}(k) = \min(p_{\mathrm{B}}(k), \tfrac{SOC_{\max} - SOC_{\mathrm{B}}(k)}{\Delta t}) \quad \text{if } p_{\mathrm{B}}(k) > 0$$

$$p_{\mathrm{B}}(k) = -\min(|p_{\mathrm{B}}(k)|, \tfrac{SOC_{\mathrm{B}}(k) - SOC_{\min}}{\Delta t}) \qquad \text{if } p_{\mathrm{B}}(k) < 0$$

Here, Δt denotes our discretization interval of 15 minutes, whereas $\mathrm{SOC}_{\min}$ and $\mathrm{SOC}_{\max}$ are respectively the minimum and maximum level of state of energy achievable for battery B. Finally, we can simulate the next state of the environment using a simple LoadFlow computation. The optimization of the neural networks is done by constant gradient descent steps, with fixed learning rates. A transition minibatch is sampled from the transition buffer to compute the gradients. In order, the parameters of the value, the critic and the actor networks are optimized in each step. The choice of the learning rates and of the batch size is critical to achieve optimal performances.

4.2 Dispatch Plan computation

After training the network parameters over several scenarios (15000 in our case), we achieve an almost optimal configuration of the parameters, provided that our choice of hyperparameters is sounded. We can now compute the day ahead dispatch plan with *Algorithm 2*. In our experiments we computed the dispatch plan by averaging among 80 different testing scenarios.

Algorithm 2 Dispatch Plan generation

Input: Trained parameters ϕ of the agent, testing scenarios
1.Initialize agent network using ϕ the parameters of the trained agent.
2. For $d = 1, \ldots, \#scenarios$:
3. Receive initial state observation ${s_1}^d$
4. For $k = 1, \ldots, 96$:
5. Select action ${a_k}^d$ from actor π_θ, according to current state ${s_k}^d$
6. Correct battery actions ${\mathrm{p^d_B}}(k)$ and ${\mathrm{q^d_B}}(k)$ if necessary
7. Run LoadFlow to obtain new voltages, ${\mathrm{P_{PCC}}^{\mathrm{d}}}(k)$ and ${\mathrm{Q_{PCC}}^{\mathrm{d}}}(k)$
8. Store the actor actions ${\mathrm{P^d_{DP}}}(k)$ and ${\mathrm{Q^d_{DP}}}(k)$.
9. Simulate next state ${s_{\mathrm{k+1}}}^{\mathrm{d}}$.
10 Compute final dispatch plan by averaging over all scenarios:

$P_{\mathrm{DP}}(k) = \frac{\sum_d {P^{\mathrm{d}}}_{\mathrm{DP}}(k)}{\#scenarios}$, for k = 1, ..., 96

$Q_{\mathrm{DP}}(k) = \frac{\sum_d {Q^{\mathrm{d}}}_{\mathrm{DP}}(k)}{\#scenarios}$, for k = 1, ..., 96

We now notice a major advantage of the Reinforcement Learning approach, compared to traditional methods. The calculation of the dispatch plan for a scenario can be done in just a few seconds, which allows us to consider multiple scenarios. The algorithm for the generation of the dispatch plan is quite similar to the training loop. Again, the network is simulated after the Load-Flow computation to compute the successive state. However, gradient descent optimization is not performed as the actor network is already trained. For each scenario d and time-step k, the dispatch plan actions $\mathrm{P^d_{DP}}(k)$ and $\mathrm{Q^d_{DP}}(k)$ are stored. At the end, the final day-ahead dispatch plan is simply computed by averaging over the scenarios.

$$P_{\mathrm{DP}}(k) = \frac{\sum_d P^{\mathrm{d}}{}_{\mathrm{DP}}(k)}{\#scenarios}, \text{ for } k = 1, ..., 96$$
$$Q_{\mathrm{DP}}(k) = \frac{\sum_d Q^{\mathrm{d}}{}_{\mathrm{DP}}(k)}{\#scenarios}, \text{ for } k = 1, ..., 96$$

In this way, we obtain a final 96 time-steps trajectory which is our dispatch plan. If the various forecasts have different probabilities of occurrence, the scenarios can be weighted accordingly. Here, since we assume our scenarios are all equally likely, we compute a standard average.

5 Network Modelling

In this section we will present the model of the grid used to evaluate the SAC algorithm. Moreover, we will formulate the model of the battery included in the network.

5.1 34-Bus Grid

In *Figure 10*, the 34-bus real distribution grid is shown. It is a medium size grid, which featuring a connection to the upper level grid, a PV power plant generator as well as a storing device. The Point of Common Coupling is situated at bus 0. 19 loads are connected to the network. The PV generator plant is connected at bus 21. As in [21] and [26], which are the main references for this work, one battery is connected at bus 1 to store energy.

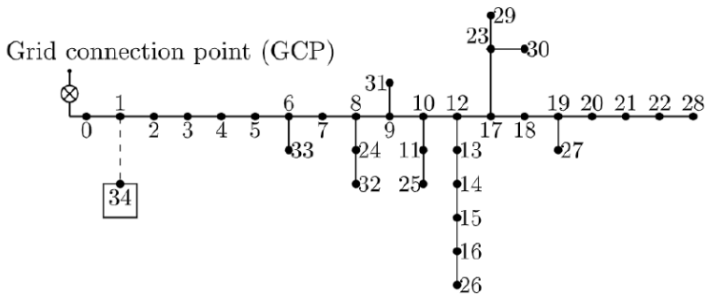

Figure 10: Model of the 34-bus grid

Name	Value	Unit	Description
S_B	25	MW	Base power (3-phase)
V_B	21	kV	Base voltage (3-phase)
I_b	1.191	kA	Base current (3-phase)
Z_b	17.64	Ω	Base impedance
Y_b	56.7	mS	Base admittance
$SOC_{B,i}^{max}$	6	MWh	Battery maximum capacity
$s_{B,i}^{max}$	6	MW	Battery nominal power
$SOC_{B,i}(0)$	33	%	Initial battery state of charge

Bus	Type
0	PCC
1	Battery connection point
2	Load connected
3	Load connected
4	Load connected
5	Load connected
6	Load connected
7	Load connected
8	-
9	Load connected
10	Load connected
11	Load connected
12	Load connected
13	Load connected
14	Load connected
15	-
16	-
17	Load connected

Bus	Type
18	-
19	Load connected
20	-
21	PV connected
22	-
23	Load connected
24	Load connected
25	-
26	Load connected
27	-
28	-
29	Load connected
30	-
31	Load connected
32	-
33	-
34	Virtual battery node for first battery

Figure 11: Overview of the grid parameters and of the grid buses

5.2 Battery model

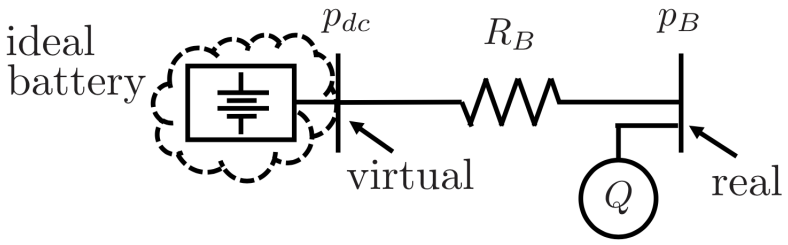

Figure 12: Battery equivalent resistance model

Any real grid battery is characterized by non negligible losses when charging and discharging. The state of charge $\mathrm{SOC_B}$ of the battery evolves as follows:

$$SOC_{\mathrm{B}}(k+1) = SOC_{\mathrm{B}}(k) + (\eta_1 p^{+}{}_{\mathrm{B}}(k) - \eta_2 p^{-}{}_{\mathrm{B}}(k))\Delta t$$

In particular, the parameters η_1 and η_2 represent the efficiency of the battery respectively during charging and discharging. We will first assume that all losses in the battery can be represented by a single resistance. With this model, the battery is represented as an ideal storage device $\eta_1 = \eta_2 = 1$, connected to a virtual bus. A single resistance (no shunt elements) connects the virtual bus with the real bus, as depicted in *Figure 12*. The active power is directly applied to the virtual bus, while the reactive power Q, which is the product of the power conversion, is connected to the real node. This *resistance model* is built on the assumption that internal losses are much more significant than those of the power converter.

6 Experiments and Results

In this section we will evaluate the dispatch plan computed by the trained RL agent, analyzing the effect of some hyperparameters. For the actor, the critic and the value network, it was chosen to setup a 2-hidden layer fully connected structure, where the intermediate layers feature 256 neurons. Of course this is only one of the multiple possible setups, as the networks do not need to have the same number of intermediate neurons. ReLU activation was used in between the layers.

Model parameters	
Number of hidden layers	2
Neurons in hidden layer 1	256
Neurons in hidden layer 2	256
Optimizer	*Adam*
Value network learning rate λ_v	0.0003
Critic networks learning rate λ_q	0.0003
Actor network learning rate λ_π	0.0003
Update parameter τ	0.005
Activation function	ReLU
Batch size	2048
Discount factor γ	0.99

For the optimization parameters the state-of-the-art Adam optimizer was chosen. The gradient descent optimization was performed with constant learning rates. After several attempts, the optimal learning rates to guarantee a stable and fast convergence were established to be $\lambda_v = \lambda_q = \lambda_\pi = 0.0003$.

6.1 Score metric

To evaluate the trained agent we can consider the *score* metric. As explained in section 5, the agent receives a reward after each time-step k, in each training scenario d. We define the daily score S_d as the sum of the rewards obtained during the day by the agent. Then:

$S_d = \sum_{k=1}^{96} r_d(k)$

The agent is trained on multiple scenarios as we have 15000 of them at our disposal. We can evaluate the trained agent by computing the average score S over the last 1000 scenarios. Averaging among the scenarios reduce the variance of the score and provides a reliable estimate on the quality of the agent. Furthermore, since we only consider the score over the last 1000 episodes, we do not keep into account the initial scores, when the agent has just begun training.

$S = \frac{\sum_{d=scenarios}^{scenarios}(S_d)}{1000}$

6.2 Reward scale

As explained in the original implementation of the SAC algorithm, the most critical hyperparameter to tune in the model is the *reward scale* α [15]. We can interpret this parameter as the inverse of the temperature in the maximum entropy framework. In general, there is no general rule to fix the reward scale and the optimal tuning depends on the environment. For smaller reward magnitudes, the policy tend to become uniform: the actor choices are mainly dictated by the entropy contribution to the loss, resulting in the model excessively focusing on exploration. On the other hand, larger reward scales usually lead to a quick convergence: the actor easily finds a local minimum but can get stuck in suboptimal solutions [16]. In *Figure 13*, we can observe how a small reward scale ($\alpha = 2$) generates a poor dispatch plan computation. The score follows an unstable trajectory, while the computed dispatch plan is not able to keep up with the actual active power at the PCC. In *Figure 14*, we can instead notice that a big reward scale $\alpha = 10$ yields a quick convergence of the model - the score history reaches the asymptote after a few epochs. However, the proposed dispatch plan is certainly not optimal but rather shows a significant error from the optimal power at the PCC. To perform this first experiments, we started with the optimal weights for the DDPG algorithm. Again, it is crucial to make w_4 the dominant weight to allow the agent to compute a feasible dispatch plan.

Training scores with different reward scales	
Reward scale α	Score S
2	-552.4
10	-228.2
100	-871.6

Properly tuning the reward scale means to balance exploration and exploitation. After several investigations, we have found out that setting the reward scale α to 10 yields a higher score.

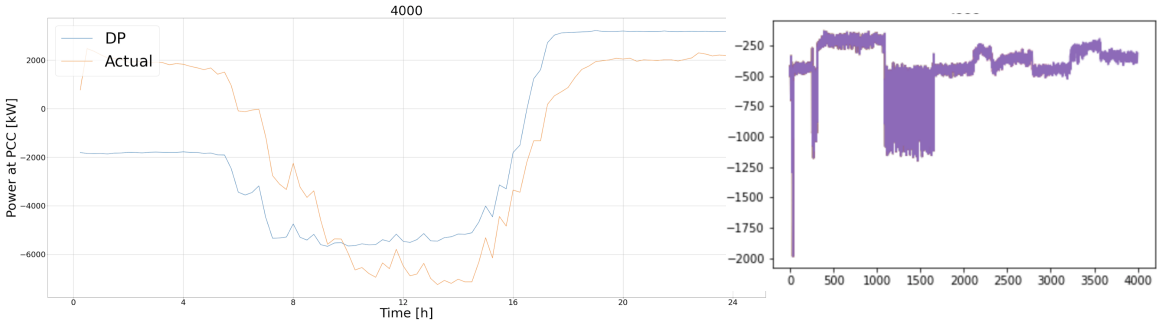

Figure 13: Training performances with reward scale $\alpha = 2$

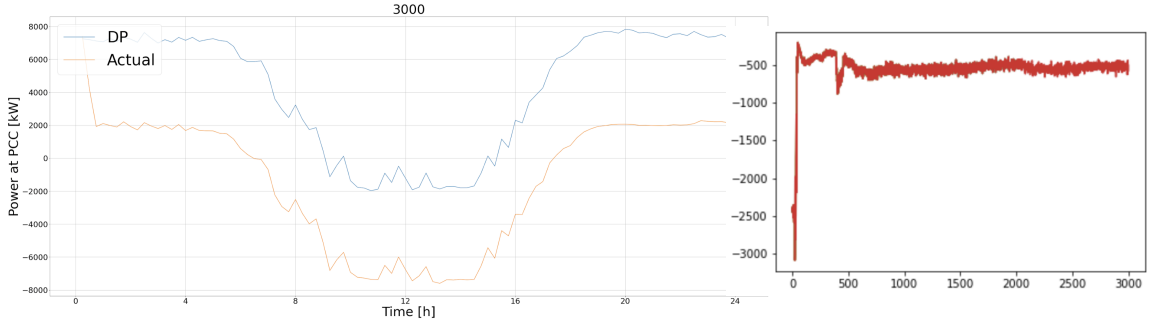

Figure 14: Training performances with reward scale $\alpha = 100$

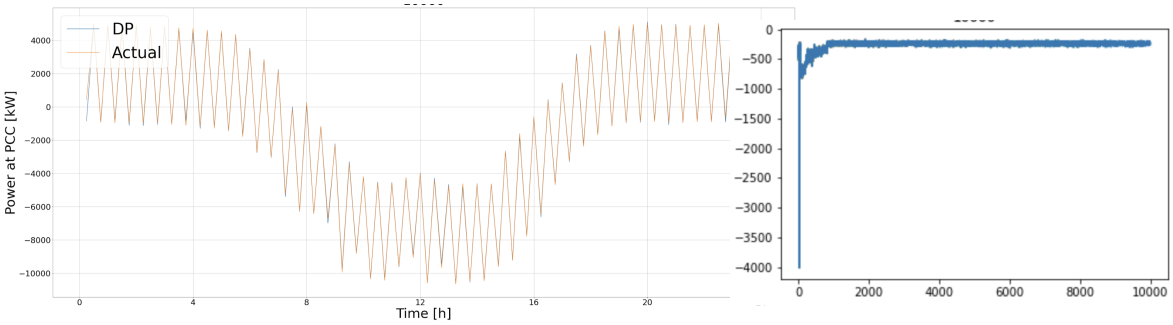

Figure 15: Training performances with reward scale $\alpha = 10$

6.3 Oscillating dispatch plan

With the tuned reward scale we obtain the results displayed in *Figure 15*. The computed dispatch plan closely matches the optimal power at the PCC, as the blue and the orange line are basically overlapping. However, this behaviour does not match the one obtained with the CoDistFlow algorithm. The power at the PCC presents a wavy behaviour, significantly oscillating between each quarter of an hour. Since these results were obtained multiple times and they all yield good scores, we deduce that the model finds this to be the optimal behaviour. If we plot the obtained battery SOC trajectory in the whole day (*Figure 13*), we notice that even the battery keeps charging and discharging. It appears that the agent is only able to to take the maximum charging and discharging action ${p_{B(k)}}^{d}$. The agents eems unable to learn intermediate battery actions.

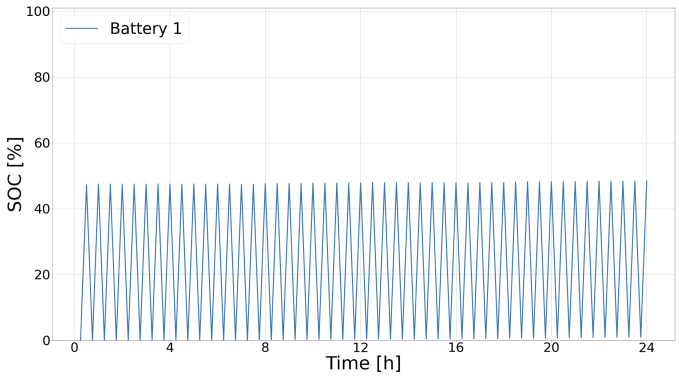

Figure 16: Battery behaviour with reward scale $\alpha = 10$

6.4 Adjustments

To tackle this problem, we have implemented a few adjustments in our model and simulation environment. The idea is to add a new penalty term in our reward function, which penalizes the magnitude of the battery action.

$$r(k) = -w_1(Q_{\mathrm{PCC}}(k)) - w_2(|P_{\mathrm{PCC}}(k)|) - w_3(P_{\mathrm{PCC}}(k)) - w_4||(S_{\mathrm{PCC}}(k) - S^{\mathrm{DP}}(k))|| - w_5 \sum_i \max({v_{\mathrm{lb}}}^2 - {v_{\mathrm{l}}}^{\mathrm{d}}(k), 0, {v_{\mathrm{l}}}^{\mathrm{d}}(k) - {v_{\mathrm{ub}}}^2) - w_6|p_{\mathrm{B}(k)}|$$

This new contribution to the reward function is controlled by the weight w_6. We have now introduced a term in the reward which evaluates the action taken by the actor. It is crucial here to compute the reward on the uncorrected battery actions, as we want to give the actor direct feedback on its actions. Overall, the uncorrected battery actions are used to compute the reward, while the corrected battery action will serve as input to the LoadFlow module to simulate the environment. With the followings set of weights we obtain very similar results to the CoDistFlow algorithm (*Figure 17*). The trajectory of the battery closely resembles the optimal one, with an initial discharge and a late charge at end of the day (*Figure 18*).

Weights of the reward function					
w_1	w_2	w_3	w_4	w_5	w_6
100	1	1	3000	1	100

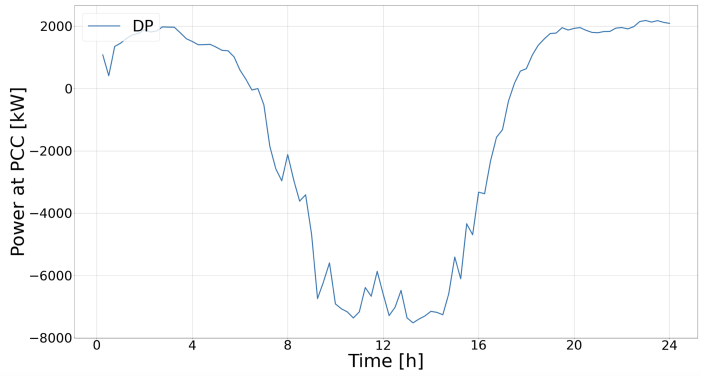

Figure 17: Computed dispatch plan after the proposed adjustments

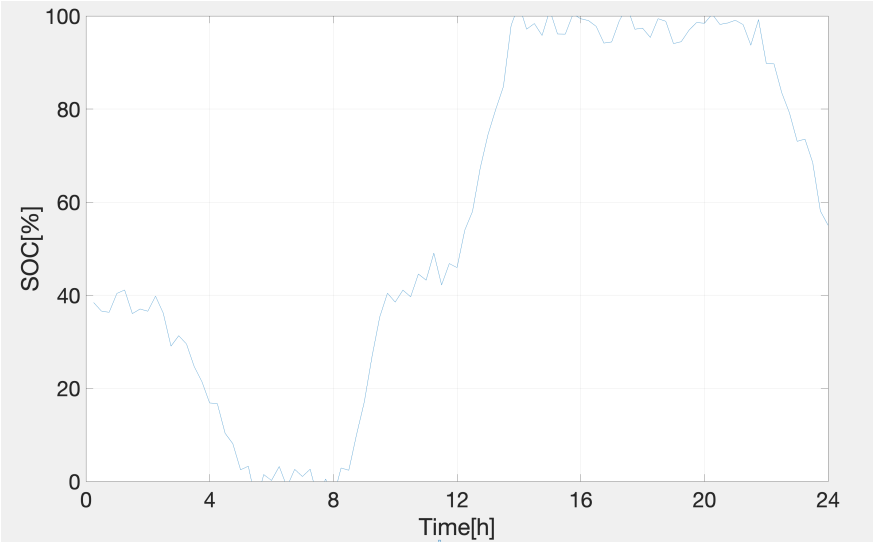

Figure 18: Battery trajectory after the proposed adjustments

6.5 Real-time application

After having investigated deeply on the model parameters and on the weights of the reward functions, we have obtained an RL agent that is able to compute a proper dispatch plan. To validate the computed dispatch plan, we decide to apply a real-time controller to the network. As detailed in [18], a Model Predictive Control algorithm tries to follow the proposed day-ahead dispatch plan as close as possible. The algorithm takes as input the dispatch plan as well as forecasts of the aggregated consumption and local distributed generation (PV power plant production). At each time step, the power at the PCC is set equal to the computed dispatch plan value. Then, if the given consumption at the loads and the PV generation would result in a power export or import from the upstream grid, the battery is used to store or deliver this energy. Since the battery has a limited capacity, whenever it is not able to balance out the power flow in the grid, energy will flow out or into the upstream grid. The amount of energy exchanged with the upstream main grid will be referred as *mismatch*. The total mismatch M_R is computed as the as the daily sum of the absolute differences between the actual realization at the PCC and the dispatch plan for a specific scenario. *Figure 19* shows the histogram of mismatches computed by

the intra-day algorithm. We observe that around 40% of scenarios yield a daily mismatch of less than 100kWh. Again, the performance are comprable to those achieved with DDPG and CoDistFlow.

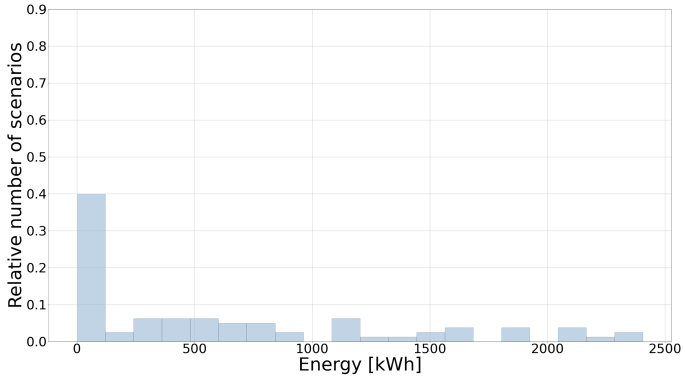

Figure 19: Computed histogram of mismatches after real time optimization with MPC algorithm.

6.6 DDPG vs SAC

We have shown that the Soft-Actor-Critic algorithm allows us to train a reliable agent to compute a dispatch plan in a medium size distribution grid. We can observe that the obtained results are similar to those achieved with the Deep Deterministic Policy Gradient method (*Figure 20*). Both reinforcement learning algorithms are able to compute an optimal dispatch plan after having been trained on several scenarios. In *Figure 21*, we notice how CoDistFlow still achieves slightly better performances in terms of real-time application with the distribution of scenarios skewed towards the left of the histrogram.
With the DDPG algorithm stability issues were experienced. When training several agents with the same load and PV scenarios, as well as with the same parameters, not all agents reached the same quality. This phenomenon, instead, was not observed with the SAC algorithm. Several experiments were conducted with the same set of weights. The agent was sooner or later able to converge to the similar results.

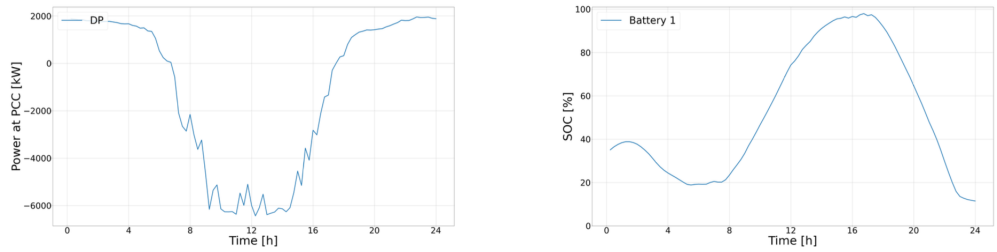

Figure 20: Dispatch plan and battery trajectory computed with DDPG algorithm

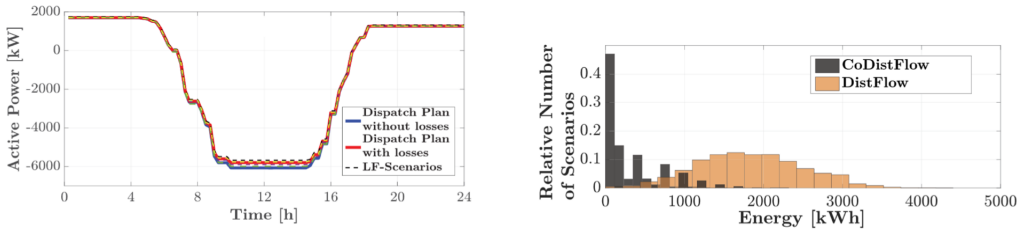

Figure 21: Dispatch plan and histogram of mismatches with CoDistFlow

7 References

[1] IEA, "Renewable energy market update, outlook for 2022-2023", https://www.iea.org/reports/renewable-energy-market-update-may-2022

[2] J.Tollefson, "What the war in Ukraine means for energy, climate and food", Nature, 2022.

[3] N.Gaind et al, "Seven ways the war in Ukraine is changing global science", Nature, 2022.

[4] C. Kathryne and P. Karen, "Renewables 101: Integrating renewable energy resources into the grid," 4 2020.

[5] P. L. Joskow, "Comparing the costs of intermittent and dispatchable electricity generating technologies," American Economic Review, vol. 101, pp. 238–41, May 2011.

[6] C. Root, H. Presume, D. Proudfoot, L. Willis and R. Masiello, "Using battery energy storage to reduce renewable resource curtailment," 2017 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017, pp. 1-5.

[7] N. Kawakami and Y. Iijima, "Overview of battery energy storage systems for stabilization of renewable energy in Japan," 2012 International Conference on Renewable Energy Research and Applications (ICRERA), 2012, pp. 1-5.

[8] Miao Fan, "A novel optimal generation dispatch algorithm to reduce the uncertainty impact of renewable energy," 2016 IEEE Power and Energy Society General Meeting (PESGM), 2016, pp. 1-5.

[9] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA, USA: A Bradford Book, 2018.

[10] David Silver et al, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv, 2017.

[11] Ryan Wong, "Getting Started with Markov Decision Processes: Reinforcement Learning", Towards Data Science, 2018.

[12] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[13] A. Choudhary, "A hands-on introduction to deep q-learning using openai gym in python.", AnalyticsVidya, 2019.

[14] D.Karunakaran, "The Actor-Critic Reinforcement Learning algorithm", Medium, 2020.

[15] T.Haarnoja, A.Zhou, P.Abbeel, S.Levinehttps, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor", ICML 2018.

[16] "Soft actor-critic." https://spinningup.openai.com/en/latest/algorithms/sac.html.

[17] Mnih et al."Human level control through deep Reinforcement Learning", Nature, 2015.

[18] F. Sossan, E. Namor, R. Cherkaoui and M. Paolone, "Achieving the Dispatchability of Distribution Feeders Through Prosumers Data Driven Forecasting and Model Predictive Control of Electrochemical Storage," in IEEE Transactions on Sustainable Energy, vol. 7, no. 4, pp. 1762-1777, Oct. 2016.

[19] N. Li, L. Chen, and S. H. Low, "Exact convex relaxation of opf for radial networks using branch flow model," in 2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm), pp. 7–12, 2012.

[20] M. Baran and F. Wu, "Optimal sizing of capacitors placed on a radial distribution system," IEEE Transactions on Power Delivery, vol. 4, no. 1, pp. 735–743, 1989.

[21] E. Stai, L. Reyes-Chamorro, F. Sossan, J.-Y. Le Boudec, and M. Paolone, "Dispatching stochastic heterogeneous resources accounting for grid and battery losses," IEEE Transactions on Smart Grid, vol. 9, no. 6, pp. 6522–6539, 2018.

[22] E. Stai, F. Sossan, E. Namor, J.-Y. Le Boudec, and M. Paolone, "A receding horizon control approach for re-dispatching stochastic heterogeneous resources accounting for grid and battery losses," Electric Power Systems Research, vol. 185, 2019.

[23] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," CoRR, 09 2015.

[24] J. Jin and Y. Xu, "Optimal Policy Characterization Enhanced Actor-Critic Approach for Electric Vehicle Charging Scheduling in a Power Distribution Network," in IEEE Transactions on Smart Grid, vol. 12, no. 2, pp. 1416-1428, March 2021.

[25] M. G. Dogan, Y. H. Ezzeldin, C. Fragouli and A. W. Bohannon, "A Reinforcement Learning Approach for Scheduling in mmWave Networks," MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM), 2021, pp. 771-776.

[26] J.Stoffel, "Dispatching Active Distribution Grids Using Reinforcement Learning", ETH Zurich, Power System Lab, Semester Thesis, 2021.