Enhanced Modeling and Control of Hybrid Power Systems: A Deep Reinforcement Learning Approach for Optimal Decision-Making

Tijmen Schipper

Supervisor:
Dr. Chiheb Ben Hammouda

A thesis presented for the degree of
Bachelor of Science

Department of Mathematics
Utrecht University
June 2024

Abstract

The management of present-day power systems has become increasingly complex due to the integration of distributed and renewable energy resources. This added complexity makes the power system optimization problem challenging to solve with conventional methods because of its increased dimensionality. Reinforcement learning offers a model-free approach to optimal decision-making. However, popular algorithms such as Deep Q-Networks (DQN) struggle to produce feasible solutions due to their inability to strictly enforce operational constraints. To address this issue, Hou Shengren et al. (International Journal of Electrical Power & Energy Systems 152, 2023) proposed a variation of the DQN algorithm called "MIP-DQN". Building on the work of Matteo Fischetti and Jason Jo (Constraints 23.3: 296-309, 2018), this algorithm formulates the trained critic neural network as a mixed-integer programming problem, enabling the algorithm to identify the action that maximises the action-value function while satisfying operational constraints.

With this thesis, we aim to provide an accessible study of the use of reinforcement learning, particularly the MIP-DQN algorithm, for the optimal management of hybrid power systems from both theoretical and applied perspectives. Theoretically, we present the necessary mathematical foundations to understand reinforcement learning in the context of power system optimization. This includes a detailed mathematical formulation of the power system optimization problem and an explanation of the novel MIP-DQN algorithm, touching on various mathematical topics such as optimization, Markov Decision Processes, optimal control, and reinforcement learning.

Our main contribution is the implementation of the MIP-DQN algorithm and the analysis of its performance in optimising the management of a specified power system.

Contents

1 Introduction

Distributed and renewable-based energy resources are a modern addition to power systems. Examples include solar panels, wind turbines and distributed generators. These resources make power systems more difficult to manage because of two reasons. Firstly, the number of individual components that need to be managed increases significantly, leading to longer time requirements for finding optimal schedules. Secondly, due to their weather dependence, the amount of power produced by these resources is uncertain. This results in significant discrepancies between good and bad days, with power values changing moment-to-moment throughout any given day. To ensure that system demand is consistently met, faster and smarter algorithms are necessary.

The approach we study in this thesis involves using reinforcement learning to solve the optimization problem of the power system. This method allows us to utilize real-time data to enhance decision-making. Being model-free, it does not require precise knowledge of the dynamics of each system component, thereby reducing the time spent on modelling the problem and allowing more focus on determining the optimal actions to take. Traditionally, the drawback of reinforcement learning was the lack of algorithms capable of strictly enforcing operational constraints. In practice, a solution is only feasible if it satisfies the power balance constraint.

A recent paper by Hou Shengren et al. (2023, [7]) presents a new reinforcement learning algorithm which is able to keep the power balance constraint satisfied. The algorithm is named "MIP-DQN", which stands for "Mixed Integer Programming Deep Q-Network". The algorithm trains neural networks using the DQN algorithm. After this it proposes to formulate the trained critic as a mixed integer programming problem (MIP). This is done as described in the work of Matteo Fischettie and Jason Jo (2018, [1]). The MIP is then optimised over the available actions in the power system. The result is the highest value action which maintains the power balance constraint.

1.1 Outline of the thesis

In Chapter 2 we provide the mathematical foundations necessary for understanding the algorithms used in reinforcement learning. The chapter begins with a description of the reinforcement learning problem, including the formulation of a Markov decision process (MDP). We explain fundamental concepts such as the action-value function, the policy function, and the Bellman equations. We then illustrate methods for finding the policy function when the dynamics of the Markov decision process are known. Finally, we explain neural networks.

In Chapter 3 we first formulate the power system we study as an optimization problem and then as a Markov decision process. Chapter 4 details the reinforcement learning algorithms capable of solving the Markov decision process defined in Section 3.4. Initially, we explain advanced concepts such as ϵ-greedy exploration, SARSA, Q-learning, and the actor-critic structure. We then describe the DQN algorithm used for training the actor and critic neural networks, covering techniques like fixed Q-targets and experience replay. We conclude this chapter with a formulation of the DQN algorithm we used. Following this, we introduce the MIP-DQN algorithm and explain how a neural network can be represented as a mixed integer programming problem.

This brings us to Chapter 5, where we present and analyze the results of using the DQN and MIP-DQN algorithms. We begin by showcasing the training process and conclude with the results obtained from deploying the neural networks.

Finally, in Chapter 6, we interpret the results presented in Chapter 5. We find that the MIP-DQN algorithm shows an improvement over the DQN algorithm. We conclude this chapter by highlighting the shortcomings of the algorithms and providing suggestions for further improvements.

2 Mathematical foundations

2.1 Reinforcement learning

Before we give mathematical definitions, theorems and proofs, this section serves to explain what reinforcement learning is and why we use it. This section is inspired by chapter 1 of [10]. We make use of the terminology and examples of [10].

Reinforcement learning is part of the field of machine learning. In machine learning we make use of datasets. We use machine learning to find some structure in the data or to extrapolate a function on the data. In reinforcement learning we strive to define a map from a set of states to a set of actions. We call this map a policy. Solving the reinforcement learning problem thus comes down to finding a good policy.
To reach this goal, we guide the machine by giving numerical rewards. Some states are particularly good, and the machine will get a high reward. When a state is bad, the reward will be lower or even negative. When we look at a lot of states and actions, we want to have a policy that maximises the total reward we get.

As an example, in games like tic-tac-toe or chess, the goal is to win the game. In tic-tac-toe one wins by being the first playing to claim three positions in a row. While in chess the rules are very different, and a player wins by checkmating the opposing king. Even though these games are different, they both make use of a board. We define each position on the board as a state. And we define each move we can make as an action.
In chess we could decide to give rewards after every action the machine takes. We would give a high reward for taking a queen, and a low reward if it doesn't take any pieces. But it's not immediately clear how we could give rewards after a single action in tic-tac-toe. In this case we could decide to only give rewards for winning, drawing or losing a game.

A first approach to solve these games is to make a model of all their rules. After this, we could use a machine to solve this model. We directly compute the best way to reach the goal. This approach is called planning.
In reinforcement learning we don't restrict ourselves to solving a model. In fact, we don't even need to know the rules of the game. Instead, we interact with the problem through the actions we take and in return we receive a new state and a reward. This allows reinforcement learning to be used for very complex problems, for which finding a good model can be challenging.

The concept of reinforcement learning is fascinating in itself. However, before making rigorous statements about this solution method, we need a solid mathematical formulation. For this reason, we introduce the Markov decision process (MDP) in Section 2.2. This framework describes time-dependent problems using states, actions, and rewards. Our goal is to predict the future rewards received when in a particular state. To achieve this, we define value functions in Section 2.3, which are considered optimal when they provide the best possible prediction. We will see that the optimal policy involves consistently choosing actions that correspond to the highest value.

To simplify the computations, we introduce recursive equations known as the Bellman equations, also covered in Section 2.3. These equations will be used in Section 2.4 to determine the optimal policy. We can only find this optimal policy when the dynamics function of the MDP is known. When the dynamics function is unknown, we will introduce reinforcement learning algorithms to find the policy, as discussed in Chapter 4. In Section 2.5, we explain neural networks,

which form the basis of our reinforcement learning algorithms, crucial for our study.

2.2 Markov Decision Process

In this section we give the framework we will use to model reinforcement learning problems. This framework is called a Markov decision process (MDP). As in Section 2.1, we will have states and actions. We use MDPs to describe time-dependent problems. Time really matters because at each moment in time, the data we have is one reward, one state and one action. Because time is sequential this means our data is dependent on the order in time.

In a MDP the actions won't only determine the immediate reward we get, but they can also have long term consequences. Taking an action will decide a change in our state. This might mean that taking a certain action prevents reaching good states. This is why in a MDP, a good policy sometimes sacrifices short term rewards, in order to get more rewards later in time.
We start by describing all the elements of a MDP and we work our way towards a formal definition. This section makes use of the notation and definitions of chapters 3.1 up to and including 3.3 in [10].

For some $k \in \mathbb{N}$, let $I \coloneqq \{0, \dots, k\} \subseteq \mathbb{N}$ and $i \in I$. Let $T \in \mathbb{R}_{\geq 0}$. For all $i \in I$, let $t_i \in \mathbb{R}_{\geq 0}$ be defined such that $t_0 = 0$, $t_k = T$ and $t_{i-1} \leq t_i$ for all $i > 0$. We define the set of possible times $\mathcal{T}$ as

$$\mathcal{T} = \{t_0, t_1, \dots, t_k\}.$$

Note that we consider a finite number of times. In our definition we gave a minimum time $t_0 = 0$ and a maximum time $t_k = T$.

As mentioned before, it is now time to define the states, actions and rewards.
A state gives information about the variables of the problem for which the MDP is modelling. To keep the MDP simple, we want a state to include enough information to define the transition to the next state. This is formalised by the requirement that the states must satisfy the Markov property. To make the notation clear, we first define the states, and then we give the Markov property.

Definition 2.2.1. We define the set of states as $\mathcal{S}$, such that every state $s \in \mathcal{S}$ satisfies the Markov property. We index a state $S_{t_i} \in \mathcal{S}$ by the time $t_i \in \mathcal{T}$, to indicate this is the state at time t_i.

Definition 2.2.2. A state $S_{t_i} \in \mathcal{S}$ satisfies the Markov property if and only if [8]

$$\mathbb{P}[S_{t_{i+1}} \mid S_{t_i}] = \mathbb{P}[S_{t_{i+1}} \mid S_{t_0}, \dots, S_{t_i},\ 0 \leq i \leq k-1].$$

We now define the actions and the rewards.

Definition 2.2.3. We define the set of actions $\mathcal{A}$. Similarly to states, we can index an action $A_{t_i} \in \mathcal{A}$ by the time $t_i \in \mathcal{T}$.

Definition 2.2.4. We define a reward r as some real number, $r \in \mathbb{R}$. Similarly to states and actions we define a set of all possible rewards $\mathcal{R} \subseteq \mathbb{R}$. For any set of rewards $\mathcal{R}$ we define $0 \in \mathcal{R}$. We write $R_t \in \mathcal{R}$ for the reward at time $t \in \mathcal{T}$. We define $R_{t'} = 0$ for all $t' \in \mathbb{R}$ where $t' \notin \mathcal{T}$. We also define $R_{t_0} = 0$.

We will now give a function which defines how an action determines the changes between states. This function also defines the reward we get for changing states. Note that the rewards are our own interpretation of how good it is to change states by a particular action. These rewards aren't intrinsic to the problem, but a part of the model. The part of the MDP model that defines these changes and rewards is called the dynamics function.

Definition 2.2.5. We define the probability for reward $r \in \mathcal{R}$ when action $a \in \mathcal{A}$ changes the state of the problem from $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ as $p_d(s, a, s', r)$, where the dynamics function $p_d : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to [0, 1]$ is defined as

$$p_d(s, a, s', r) \coloneqq \mathbb{P}[S_{t_{i+1}} = s', R_{t_{i+1}} = r \mid S_{t_i} = s, A_{t_i} = a,\ 0 \leq i \leq k-1],$$

with

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s, a, s', r) = 1, \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}.$$

Using the dynamics function we will formulate two other functions. We define the transition probability function, which gives a probability for the next state. And we define the reward function, which gives the expected next reward.

Definition 2.2.6. We define the probability with which an action a changes the state of the problem from s to s' as $p_t(s, a, s')$. Where we introduce the transition probability function $p_t : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ as

$$\begin{aligned} p_t(s, a, s') &\coloneqq \mathbb{P}[S_{t_{i+1}} = s' \mid S_{t_i} = s, A_{t_i} = a,\ 0 \leq i \leq k-1] \\ &= \sum_{r \in \mathcal{R}} p_d(s, a, s', r), \end{aligned}$$

where

$$\sum_{s' \in \mathcal{S}} p_t(s, a, s') = 1, \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}.$$

Definition 2.2.7. We define the expected reward for changing from state s to s' by action a as $r(s, a, s')$. Where we introduce the reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ as

$$\begin{aligned} r(s, a, s') &\coloneqq \mathbb{E}[R_{t_{i+1}} \mid S_{t_i} = s, A_{t_i} = a, S_{t_{i+1}} = s',\ 0 \leq i \leq k-1] \\ &= \sum_{r \in \mathcal{R}} r \frac{p_d(s, a, s', r)}{p_t(s, a, s')}. \end{aligned}$$

We now understand the concepts of time, states, actions, and rewards, as well as how they interact with each other. If we start in some state S_{t_0} and we take an action A_{t_0}, the dynamic function gives information about what S_{t_1} and R_{t_1} will be. Using the dynamics function we can make a sequence of states, actions and rewards. Note that these sequences can't get infinitely long because we defined a minimum and a maximum time in the definition of the times $\mathcal{T}$. We call the maximal sequences the episodes.

Definition 2.2.8. We define an episode as a sequence of states, actions and rewards of the form

$$R_{t_0}, S_{t_0}, A_{t_0}, R_{t_1}, S_{t_1}, \ldots, R_{t_k}, S_{t_k}, A_{t_k},$$

where time t_k is the maximum time in $\mathcal{T}$.

Note that it is possible that there are states where every action leads to staying in this state. In this case an episode has a subsequence with only one state, which keeps being repeated. If the reward from this state is always 0 we call it a terminal state.

Lastly we introduce the discount rate $\gamma \in [0, 1]$ to the MDP. This parameter will help us model if we want to value immediate rewards higher than future rewards.

Definition 2.2.9. We define the return G_{t_i} as the sum of all discounted rewards starting from time $t_i \in \mathcal{T}$:

$$G_{t_i} \coloneqq R_{t_{i+1}} + \gamma R_{t_{i+2}} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t_{i+k+1}}.$$

Note that the sum in Definition 2.2.9 is well-defined because Definition 2.2.4 gives $R_t = 0$ when $t \notin \mathcal{T}$.

Definition 2.2.10. Using the previously defined set of times $\mathcal{T}$, set of states $\mathcal{S}$, set of actions $\mathcal{A}$, set of rewards $\mathcal{R}$, a dynamics function p_d and a discount rate γ; we define a Markov decision process (MDP) as a tuple $(\mathcal{T}, \mathcal{S}, \mathcal{A}, \mathcal{R}, p_d, \gamma)$.

In Definition 2.2.10, we introduced the MDP framework for modeling our problems. But how do we solve our problem using a MDP? Remember, solving a reinforcement learning problem involves maximising the total reward. To achieve this, we need to understand which states and actions lead to high rewards. In the next section, we will define value functions to help us determine this.

2.3 Bellman equations

Consider this: In this section, we define value functions for an MDP. These functions assign a value to every state and action, allowing us to determine which states and actions are better than others. We also formalise the concept of a policy, which describes the decisions to make between certain actions. By taking actions with high values, we can expect to maximise the cumulative future reward. Additionally, we explore the Bellman equations, which are recursive functions that can be used to find the value functions given an MDP.

This section makes use of the notation and definitions of chapters 3.5 and 3.6 in [10]. All definitions, lemmas and theorems are defined for a MDP $(\mathcal{T}, \mathcal{S}, \mathcal{A}, \mathcal{R}, p_d, \gamma)$.

Definition 2.3.1. We define the probability that an action a changes the state s in the MDP as $\pi(s, a)$, where the policy function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is defined as

$$\pi(s, a) \coloneqq \mathbb{P}[A_{t_i} = a \mid S_{t_i} = s,\ 0 \leq i \leq k],$$

where

$$\sum_{a \in \mathcal{A}} \pi(s, a) = 1, \quad \text{for all } s \in \mathcal{S}.$$

Both the policy function as well as the transition probability function give information about how the state in the MDP will change. Note that the policy function gives us a probability for the next action given that we only know the current state. Whereas the transition probability function gives us a probability for the next state, given that we know the current state as well as the current action.

Using our previously defined notion of the return we can now define the value of a state. We consider a particular policy π and a state s. Starting from state s and taking actions according to the policy π we get a sequence of states, actions and rewards. Because a policy is probabilistic, such a sequence doesn't have to be uniquely defined. This means we can get different values for the return. For this reason we define the value of a state as the expected return from this state. We write $\mathbb{E}_\pi$ for the expected value that we get by taking actions according to policy π.

Definition 2.3.2. We define the value of a state $s \in \mathcal{S}$ for a policy function π as the state-value function $v_\pi(s) : \mathcal{S} \to \mathbb{R}$ which is expressed as

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s,\ 0 \leq i \leq k].$$

Similarly to defining a function for the value of a state, we can define a function to determine the value of a particular action. Since one action can be taken from multiple different states, we define the value of an action for each state from which it is taken.

Definition 2.3.3. We define the value of an action $a \in \mathcal{A}$ taken from a state $s \in \mathcal{S}$, for a policy function π as the action-value function $q_\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, expressed as

$$q_\pi(s, a) \coloneqq \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s, A_{t_i} = a].$$

We will now show and prove three lemmas. These will be used to derive the Bellman equations. The first lemma gives a recursive expression for the return.

Lemma 2.3.4. For the return G_{t_i} at time $t_i \in \mathcal{T}$, where $0 \leq i \leq k$, the return is given by

$$G_{t_i} = R_{t_{i+1}} + \gamma G_{t_{i+1}}.$$

Proof. Using Definition 2.2.9 in (2.1) and in (2.2), we get

$$\begin{aligned} G_{t_i} &= \sum_{k=0}^{\infty} \gamma^k R_{t_{i+k+1}} && (2.1) \\ &= \gamma^0 R_{t_{i+1}} + \sum_{k=1}^{\infty} \gamma^k R_{t_{i+k+1}} \\ &= R_{t_{i+1}} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t_{i+k+1}} \\ &= R_{t_{i+1}} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t_{i+k+2}} \\ &= R_{t_{i+1}} + \gamma G_{t_{i+1}}. && (2.2) \end{aligned}$$

□

The next lemma tells us that the value of a state is equal to the summed values of the actions we can take from this state.

Lemma 2.3.5. For a policy function π and action-value function $q_\pi(s, a)$, the state-value function is given by

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) q_\pi(s, a).$$

Proof. We use Definition 2.3.2 in (2.3), Definition 2.3.1 in (2.4), and Definition 2.3.3 in (2.5) to obtain

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s,\ 0 \leq i \leq k] && (2.3) \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s, A_{t_i} = a,\ 0 \leq i \leq k] && (2.4) \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) q_\pi(s, a). && (2.5) \end{aligned}$$

While Lemma 2.3.5 described how we can express the state-value function in terms of the action-value function, the next lemma does the opposite. The value of an action will be equal to the expected immediate reward summed with the values of the states the action transitions towards.

Lemma 2.3.6. For a policy function π, state-value function $v_\pi(s)$ and dynamics function p_d, the action-value function is given by

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s')\right).$$

Proof. We use Definition 2.3.3 in (2.6), Lemma 2.3.4 in (2.7), Definition 2.2.5 in (2.8) and Definition 2.3.2 in (2.9) to obtain

$$\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s, A_{t_i} = a,\ 0 \le i \le k] && (2.6)\\
&= \mathbb{E}_\pi[R_{t_{i+1}} + \gamma G_{t_{i+1}} \mid S_{t_i} = s, A_{t_i} = a,\ 0 \le i \le k-1] && (2.7)\\
&= \mathbb{E}_\pi[R_{t_{i+1}} \mid S_{t_i} = s, A_{t_i} = a,\ 0 \le i \le k-1] + \gamma \mathbb{E}_\pi[G_{t_{i+1}} \mid S_{t_i} = s, A_{t_i} = a,\ 0 \le i \le k-1]\\
&= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) r + \gamma \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \mathbb{E}_\pi[G_{t_{i+1}} \mid S_{t_{i+1}} = s',\ 0 \le i \le k-1] && (2.8)\\
&= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) r + \gamma \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) v_\pi(s') && (2.9)\\
&= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s')\right).
\end{aligned}$$

□

We now have all the necessary tools to characterise the Bellman equations.
We can use the Bellman equation for v_π to recursively define the value of a state. We look at each action we take from this state s using our policy. For each action we take the corresponding reward and we add the values of the states we end up in.

Theorem 2.3.7. For a policy function π and dynamics function p_d, the Bellman equation for the state-value function v_π is given by

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \left(\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s')\right) \right).$$

Proof. Using Lemma 2.3.5 in (2.10) and Lemma 2.3.6 in (2.11) we get

$$\begin{aligned}
v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(s,a) q_\pi(s,a) && (2.10)\\
&= \sum_{a \in \mathcal{A}} \pi(s,a) \left(\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s')\right) \right). && (2.11)
\end{aligned}$$

□

We can use the Bellman equation for q_π to recursively define the value of an action. First we take our expected immediate reward. We add to this the rewards of the future actions we consider using our policy.

Theorem 2.3.8. For a policy function π and dynamics function p_d, the Bellman equation for the action-value function q_π is given by

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma \left(\sum_{a' \in \mathcal{A}} \pi(s',a') q_\pi(s',a') \right) \right).$$

Proof. Using Lemma 2.3.6 in (2.12) and Lemma 2.3.5 in (2.13) we get

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s') \right) \tag{2.12}$$

$$= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma \left(\sum_{a' \in \mathcal{A}} \pi(s',a') q_\pi(s',a') \right) \right). \tag{2.13}$$

□

In both Bellman equations we take the values of our successors and we use them to determine the value for the current state or action. Using the Bellman equations we don't have to use the full return from each state or action to determine the value of this state or action. If we have some certainty about the values of the successor states and actions, we easily get the value of the current state or action.

We have defined the policy, the state-value function and the action-value function. In the reinforcement learning problem we want to find the optimal functions that maximise our total reward. We will now define what it means for these functions to be optimal.

Definition 2.3.9. A policy π_* is called an optimal policy if and only if

$$v_{\pi_*}(s) \geq v_\pi(s)$$

for all policy functions π and all $s \in \mathcal{S}$.

In the following definitions we use the notation $\max_\pi$ to take the maximum value over all policy functions $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$.

Definition 2.3.10. We define the optimal state-value function $v_*(s) : \mathcal{S} \to \mathbb{R}$ as

$$v_*(s) = \max_\pi \ v_\pi(s).$$

Definition 2.3.11. We define the optimal action-value function $q_*(s,a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as

$$q_*(s,a) = \max_\pi \ q_\pi(s,a).$$

We can also make recursive expressions for the optimal value functions. These are called the Bellman optimality equations. We proof a lemma before we state the Bellman optimality equations. This lemma expresses the optimal state-value function in terms of the optimal action-value function.

Lemma 2.3.12. For the optimal action-value function $q_*(s,a)$, the optimal state-value function v_* is given by

$$v_*(s) = \max_{a \in \mathcal{A}} \ q_*(s,a).$$

Proof. We use Definition 2.3.10 in (2.14), Definition 2.3.2 in (2.15), Definition 2.3.3 in (2.16) and Definition 2.3.11 in (2.17) to obtain

$$
\begin{aligned}
v_*(s) &= \max_{\pi} \; v_\pi(s) && (2.14)\\
&= \max_{\pi} \; \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s,\; 0 \le i \le k] && (2.15)\\
&= \max_{a \in \mathcal{A}} \; \max_{\pi} \; \mathbb{E}_\pi[G_{t_i} \mid S_{t_i} = s, A_{t_i} = a,\; 0 \le i \le k] \\
&= \max_{a \in \mathcal{A}} \; \max_{\pi} \; q_\pi(s,a) && (2.16)\\
&= \max_{a \in \mathcal{A}} \; q_*(s,a). && (2.17)
\end{aligned}
$$

□

We now have all the necessary tools to characterise the Bellman optimality equations. Similarly to the Bellman equation for v_π, the Bellman optimality equation for v_* is a recursive equation.

Theorem 2.3.13. For a policy function π and dynamics function p_d, the Bellman optimality equation for v_* is given by

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_*(s')\right).$$

Proof. We use Lemma 2.3.12 in (2.18), Definition 2.3.11 in (2.19), Lemma 2.3.6 in (2.20) and Definition 2.3.10 in (2.21) to obtain

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}} \; q_*(s,a) && (2.18)\\
&= \max_{a \in \mathcal{A}} \; \max_{\pi} \; q_\pi(s,a) && (2.19)\\
&= \max_{a \in \mathcal{A}} \; \max_{\pi} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_\pi(s')\right) && (2.20)\\
&= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma \max_{\pi} \; v_\pi(s')\right) \\
&= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma v_*(s')\right). && (2.21)
\end{aligned}
$$

□

Similarly to the Bellman equation for q_π, the Bellman optimality equation for q_* is a recursive equation.

Theorem 2.3.14. For a policy function π and dynamics function p_d, the Bellman optimality equation for q_* is given by

$$q_*(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s,a,s',r) \left(r + \gamma \max_{a'} \; q_*(s',a')\right).$$

Proof. We use Definition 2.3.11 in (2.22), Lemma 2.3.6 in (2.23), Definition 2.3.10 in (2.24) and

Lemma 2.3.12 in (2.25) to obtain

$$
\begin{aligned}
q_*(s,a) &= \max_\pi \; q_\pi(s,a) && (2.22)\\
&= \max_\pi \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma v_\pi(s')\right) && (2.23)\\
&= \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma \max_\pi \; v_\pi(s')\right)\\
&= \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma v_*(s')\right) && (2.24)\\
&= \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma \max_{a'} \; q_*(s',a')\right). && (2.25)
\end{aligned}
$$

□

2.4 Solving the reinforcement learning problem

In this section, we present methods used to iteratively find a good policy. The methods discussed here assume that the dynamics function of the MDP is known. In Section 4, we will explore methods that do not rely on the dynamics function.

We first show how to determine the value functions if we are given a policy. Then we show ways to improve a policy if we are given the value functions. The combination of these two steps leads to an iterative process called policy iteration. We will also explain another variation called value iteration.

This section makes use of the notation and definitions of chapters 4.1 up to and including 4.4 in [10].

2.4.1 Policy evaluation

Computing the value functions when we are given a policy is called "policy evaluation". We show an iterative approach based on the Bellman equation for v_π from Theorem 2.3.7. Recall that this theorem gives us the following equality for a state $s \in \mathcal{S}$:

$$
v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \left(\sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma v_\pi(s')\right)\right).
$$

The idea of iterative value evaluation is to turn this equation into an iterative approximation. We initialise a state-value function v_0 with arbitrary values. We define v_k to be our estimate of v_π at iteration k, for $k \in \mathbb{N} \setminus \{0\}$. Iterating with the Bellman equation gives us

$$
v_{k+1}(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \left(\sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s,a,s',r)\left(r+\gamma v_k(s')\right)\right).
$$

The intuition behind the convergence is as follows: The first values of v_0 at every state are arbitrary and are most likely not close to the real values of v_π. Even so, by doing more iterations, we add more real rewards r to every value of v_k. The more real rewards we add, the more the functions converge to v_π. However, there is no guarantee that we convergence to v_π in a finite number of iterations.

We won't prove the convergence of this algorithm. Nevertheless, we do note that v_π is a fixed point of this iteration. When $v_k = v_\pi$, the iterative function will be the same as the Bellman equation for v_π. We also note that if there are any terminal states we must give them a zero value. Otherwise their wrong value will propagate itself in all the next iterations.

Lastly, we remark that once an approximation for the state-value function v_π is known, we can use Lemma 2.3.6 to compute an approximation for the action-value function q_π.

2.4.2 Policy improvement

After computing the value function, we consider it's use to improve the policy. This process is called "policy improvement". When we are given a policy π and its value functions v_π and q_π (defined in definitions 2.3.2 and 2.3.3), we wish to construct a policy π' that is better. One way to do this is by replacing low value actions in π, by actions that have a higher value. We clarify this in the following theorem.

Theorem 2.4.1. We assume that $\gamma \in [0, 1)$. For two deterministic policy functions π and π', if for all states $s \in \mathcal{S}$ the inequality

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{2.26}$$

holds, then for all states $s \in \mathcal{S}$

$$v_{\pi'}(s) \geq v_\pi(s). \tag{2.27}$$

If we have a strict inequality in equation (2.26) for a particular state, then at this state we have a strict inequality in equation (2.27) as well.

Proof. We prove the non-strict inequality. The strict case follows similarly.
We use inequality (2.26), Definition 2.3.3 and Lemma 2.3.4 to obtain, for $s \in \mathcal{S}$ and $0 \leq i \leq k$:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}_\pi \left[G_{t_i} \mid S_{t_i} = s, A_{t_i} = \pi'(s) \right] \\ &= \mathbb{E}_\pi \left[R_{t_{i+1}} + \gamma G_{t_{i+1}} \mid S_{t_i} = s, A_{t_i} = \pi'(s) \right] \\ &= \mathbb{E} \left[R_{t_{i+1}} + \gamma v_\pi(S_{t_{i+1}}) \mid S_{t_i} = s, A_{t_i} = \pi'(s) \right] \\ &= \mathbb{E}_{\pi'} \left[R_{t_{i+1}} + \gamma v_\pi(S_{t_{i+1}}) \mid S_{t_i} = s \right]. \end{aligned} \tag{2.28}$$

Note that in equation (2.28) we made use of the fact that π is a deterministic policy. By Definition 2.3.2, this gives us that $v_\pi(S_{t_{i+1}}) = \mathbb{E}_\pi[G_{t_{i+1}} \mid S_{t_{i+1}}] = G_{t_{i+1}}$.
Using the previous result twice we obtain, for $0 \leq i \leq k$:

$$\begin{aligned} v_\pi(s) &\leq \mathbb{E}_{\pi'} \left[R_{t_{i+1}} + \gamma \, v_\pi(S_{t_{i+1}}) \mid S_{t_i} = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t_{i+1}} + \gamma \, \mathbb{E}_{\pi'} \left[R_{t_{i+2}} + \gamma \, v_\pi(S_{t_{i+2}}) \mid S_{t_{i+1}} \right] \mid S_{t_i} = s \right] \\ &= \mathbb{E}_{\pi'} \left[R_{t_{i+1}} + \gamma \, R_{t_{i+2}} + \gamma^2 \, v_\pi(S_{t_{i+2}}) \mid S_{t_i} = s \right] \end{aligned}$$

By repeatedly using this result we obtain, for $0 \leq i \leq k$:

$$\begin{aligned}
v_\pi(s) &\leq \mathbb{E}_{\pi'}\left[R_{t_{i+1}} + \gamma\ R_{t_{i+2}} + \gamma^2\ v_\pi(S_{t_{i+2}}) \mid S_{t_i} = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t_{i+1}} + \gamma\ R_{t_{i+2}} + \gamma^2\ R_{t_{i+3}} + \gamma^3\ v_\pi(S_{t_{i+3}}) \mid S_{t_i} = s\right] \\
&\leq \dots \\
&\leq \mathbb{E}_{\pi'}\left[\lim_{n\to\infty}\left(\sum_{k=0}^{n}\gamma^k R_{t_{i+k+1}} + \gamma^{n+1} v_\pi(S_{t_{i+n+1}})\right) \mid S_{t_i} = s\right] && (2.29) \\
&= \mathbb{E}_{\pi'}\left[\sum_{k=0}^{\infty}\gamma^k R_{t_{i+k+1}} \mid S_{t_i} = s\right] && (2.30) \\
&= \mathbb{E}_{\pi'}\left[G_{t_i} \mid S_{t_i} = s\right] && (2.31) \\
&= v_{\pi'}(s). && (2.32)
\end{aligned}$$

Note that the infinite sums in equations 2.29 and 2.30 are well defined, because we defined $R_t = 0$ for $t \notin \mathcal{T}$, and $\mathcal{T}$ is a finite set.
In equation (2.30) we used that $\gamma \in [0, 1)$. This gives us that $\lim_{n\to\infty} \gamma^{n+1} = 0$.
We used Definition 2.2.9 in equation (2.31) and Definition 2.3.2 in equation (2.32). □

Theorem 2.4.1 gives us a method to do policy improvement. Given a policy π, we can construct a better policy π' as follows. First we keep $\pi' = \pi$. This way we have $q_\pi(s, \pi'(s)) = v_\pi(s)$ for all states $s \in \mathcal{S}$. Then we try to find an action a such that for a state s we have $q_\pi(s, a) > v_\pi(s)$. Theorem 2.4.1 tells us that when we take this action $\pi'(s) = a$ instead of $\pi(s)$, we will have $v_{\pi'}(s) > v_\pi(s)$ in this state, and we have $v_{\pi'}(s') \geq v_\pi(s')$ in all other states $s' \in \mathcal{S}$ as well. Implying an improved policy. We call this a greedy policy update of π with respect to v_π.
Of course, we don't have to restrict ourselves to improve the policy using a single action. We can do policy improvement by greedily taking the action with the highest value at every state.

It is possible that there is no action with a higher value at any state. In this case our policy can't be improved and it is an optimal policy by definition. This method doesn't give a guarantee to obtain the optimal policy within a certain time. However, each iteration does give a guaranteed improvement of the policy. For a finite MDP this means that at some iteration the policy will be optimal.

One thing to note is that we considered deterministic policy functions in Theorem 2.4.1. If a policy is not deterministic, the same principles apply (chapter 4.2 of [10]). We can do a greedy policy update with respect to the value function.

2.4.3 Policy iteration

Previously, we have shown the methods of policy evaluation and policy improvement. The first of which uses a policy to determine the value function, while the second uses a value function to find a better policy. These methods can be repeated to find an optimal policy. We evaluate and improve a policy function, after which the improved function is used for the evaluation and improvement steps again. This iterative process is called "policy iteration".

2.4.4 Value iteration

One downside of policy iteration is that we have to do policy evaluation often. The algorithm of policy evaluation converges in the limit. It turns out that policy iteration still converges to the

optimal policy if we don't wait for convergence in policy evaluation. Even doing one iteration of policy evaluation is enough to get convergence. Doing policy iteration where we only do one iteration of policy evaluation, is called "value iteration".

We can combine the policy evaluation and improvement steps in value iteration. We initialise a state-value function v_0 with arbitrary values. We define v_k to be our estimate of v_π at iteration k, for $k \in \mathbb{N} \setminus \{0\}$. This gives us the following iterative update for $s \in \mathcal{S}$:

$$\begin{aligned} v_{k+1}(s) &= \max_{a \in \mathcal{A}} \mathbb{E}\left[R_{t_{i+1}} + \gamma v_k(S_{t_{i+1}}) \mid S_{t_{i+1}} = s, A_{t_{i+1}} = a,\ 0 \leq i \leq k\right] \\ &= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p_d(s, a, s', r)\left(r + \gamma v_\pi(s')\right). \end{aligned}$$

Note the similarity with the Bellman optimality equation in Theorem 2.3.13.
In value iteration we have to do fewer computations than in regular policy iteration. It is therefore often faster. It can be shown that value iteration converges to the optimal value function in a finite MDP (chapter 4.4 of [10]).

Value iteration gives an estimated value function. We can simply define a policy using this function. At every state we greedily take an action with the maximal value. Note that we can use Lemma 2.3.6 to calculate the action-value function.

So have we solved the reinforcement learning problem? The answer to this question is yes, if we know the dynamics function of our MDP. With the dynamics function, we can use the theory from this section to find a good policy. However, in practice, the dynamics function is often unknown. In the next sections, we explore neural networks, which will form the basis of Section 4. There, we will examine reinforcement learning algorithms designed to search for a good policy when the dynamics function is unknown.

2.5 Artificial neural networks

An artificial neural network (ANN) is a mathematical model which is often used to approximate non-linear functions. An ANN can have many parameters which can be adjusted to better fit this function. We first describe the intuition behind the ANN. Then we define what neurons are and we take a look at the topology of a neural network. We conclude this section by showing how parameters of a network can be adjusted to minimise the error. We make a reference to an algorithm called back-propagation.

2.5.1 Biological neuron

As the name "artificial neural network" suggests, it is modelled after a biological neural network. Before we define the components of an ANN, it is worthwhile to describe them in their biological context. Just like the networks in a brain, the ANN consists of neurons. Biological neurons send electrochemical signals to one another through their synapses. Similarly, the neurons in an ANN can be connected and give an output to other neurons. When a biological neuron receives a total signal greater than some threshold, it causes the neuron to send a signal as well. Similarly, an artificial neuron in an ANN takes the input signals, adds them together, and if the sum is great enough, it will give an output. To model the threshold which determines if an artificial neuron gives an output, we use a bias and an activation function [11].

2.5.2 Artificial neuron

Let $n \in \mathbb{N}$. Assume an artificial neuron p has input signals $\{x_1, \ldots, x_n\} \subseteq \mathbb{R}$. The neuron processes these signals by taking a weighted sum with weights $\{w_1^p, \ldots, w_n^p\} \subseteq \mathbb{R}$. The neuron adds a bias $b^p \in \mathbb{R}$ to this sum. The result of which gets mapped by an activation function $f^p : \mathbb{R} \to \mathbb{R}$. Typically all neurons in a network use the same activation function. However, different neurons often have different weights and biases. These are the parameters we can change to make the network better fit a function. To summarise, this neuron has a value $y^p \in \mathbb{R}$ equal to [4]

$$y^p = f^p\left(\sum_{i=1}^{n} w_i^p x_i + b^p\right).$$

2.5.3 Network terminology

We can make a neural network using a set N of neurons. The neural network has the structure of a directed graph with vertices N and a set of directed edges E. Consider a directed edge $e = (n_1, n_2)$. In this case the value of neuron n_1 is an input for neuron n_2. Typically, we design a neural network such that it has layers. A layer is an independent set of neurons. This means there is no edge connecting any two neurons in the layer.

Assume a neural network has $l \in \mathbb{N}$ layers. We can assign a number $i \leq l$ to each layer. The neurons in layer $i \leq l$ have edges directed to the neurons in layer $i + 1$. Layer 0 is called the input layer and doesn't have any ingoing edges. When we use the neural network we initialise the values of these neurons using an input vector. Layer l is called the output layer of the network and doesn't have any outgoing edges. After we compute all the values in our network, the values of the neurons in the output layer will be an output vector of our neural network. All the layers that are neither the input nor the output layers, are called hidden layers.

2.5.4 Network structures

What we have described so far is a feed-forward neural network. Many other structures can be made. For instance, a recurrent (also called "feed-back") neural network. In this case neurons can have directed edges to neurons in the same or in previous layers.

Computing the values of the neurons in a feed-forward neural network is simple. Layer 0 is initialised by the input vector. Every layer $i \geq 1$ uses the values of the previous layer $i - 1$ as input. This means we can compute the values of the neurons in increasing order of the layers. In a recurrent neural network this works a little differently. Neurons don't only use values of the previous layer. We have to initialise the value of all neurons, typically as 0. Let's consider time in our neural networks. In this case, the feed-forward neural network will have only one output after we computed the output vector. However, this doesn't have to be the case in a recurrent neural network. As we iterate over all the layers, to compute the values of the neurons, the values of the output neurons may change. It is possible that we end up in an equilibrium state, after which the output stays the same [4, 11].

Figure 2.1 shows a feed-forward neural network. The dots represent neurons and have names $k_1, k_2, \ldots, n_1, n_2$. The layers are visualised by green boxes. All the neurons in a green box are part of that layer. We have drawn black arrows between neurons to indicate that the value of a neuron is an input for the neuron at the tip of the arrow. Furthermore, input and output vectors have been drawn. The vector $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ initialises layer 0 and the vector $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ stores the values of the neurons in layer 3. The grey arrows show that the values of these neurons are used.

We call the layers in this network "fully connected", because every neuron in every layer has an arrow pointing to every neuron in the next layer.

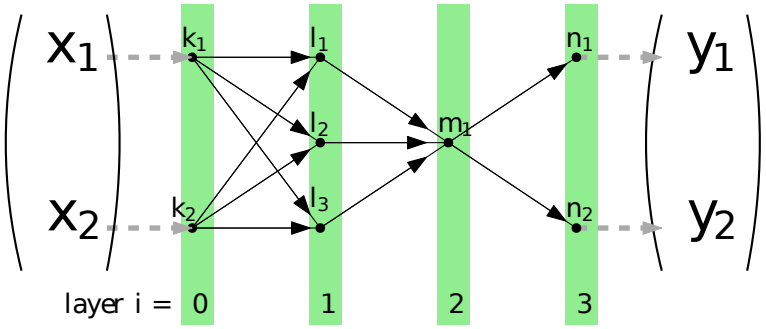

Figure 2.1: A feed-forward neural network.

Figure 2.2 shows a recurrent neural network. It has a similar structure to the network in Figure 2.1. The difference is that arrows have been drawn within a layer, or back to a previous layer. This shows that neurons in a recurrent network can have input values from neurons in the same layer, or from a higher layer.

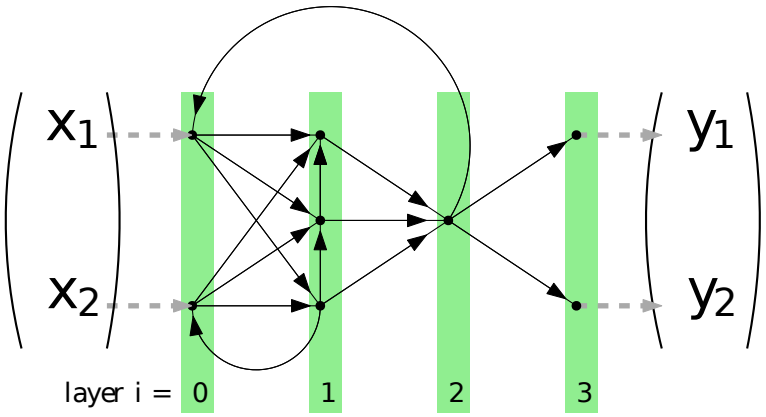

Figure 2.2: A recurrent neural network.

We demonstrate how the values x_1 and x_2 propagate through the network in Figure 2.1. We assume every neuron p has an activation function f^p, weights w^p and a bias b^p. We abuse notation and use the name of the neuron p to denote it's value y^p.
Using the input values x_1 and x_2, the values of neurons k_1 and k_2 are

$$k_1 = x_1,$$
$$k_2 = x_2.$$

These values are used to obtain those of neurons l_1, l_2 and l_3 as

$$l_1 = f^{l_1}(w_1^{l_1} k_1 + w_2^{l_1} k_2 + b^{l_1}),$$
$$l_2 = f^{l_2}(w_1^{l_2} k_1 + w_2^{l_2} k_2 + b^{l_2}),$$
$$l_3 = f^{l_3}(w_1^{l_3} k_1 + w_2^{l_3} k_2 + b^{l_3}).$$

In a similar fashion we obtain the value of m_1 as

$$m_1 = f^{m_1}(w_1^{m_1} l_1 + w_2^{m_1} l_2 + w_3^{m_1} l_3 + b^{m_1}).$$

Lastly we obtain the values of n_1 and n_2,

$$n_1 = f^{n_1}(w_1^{n_1} m_1 + b^{n_1}),$$
$$n_2 = f^{n_2}(w_1^{n_2} m_1 + b^{n_2}).$$

The output vector of the network is

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix}.$$

We now know how a neural network works. However, to use one in practice we need to choose weights, biases and activation functions. These need to be chosen in a way such that our network gives the outcomes that we want.
There are many functions we could take for the activation functions. In practice we often choose the same activation function for all neurons. One example is the sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$, where

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

2.5.5 Gradient descent

In order to find good weights and biases we can make use of the gradient descent method. We describe this algorithm as it is explained in [6]. For a proof of the convergence we refer to [6].
Assume we are given a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$. Gradient descent is an iterative method which we use to find

$$\min_{\alpha \in \mathbb{R}^d} f(\alpha)$$

as well as

$$\alpha^* = \operatorname*{argmin}_{\alpha \in \mathbb{R}^d} f(\alpha).$$

The gradient descent algorithm initialises some $\alpha_0 \in \mathbb{R}^d$, $k = 0$, and defines the learning rate $\gamma_k \in \mathbb{R}_{\geq 0}$. We obtain α_{k+1} by the iteration

$$\alpha_{k+1} = \alpha_k - \gamma_k \nabla f(\alpha_k).$$

After an iteration, the algorithm increments k and defines the next value of γ_k. We repeat this process until the gradient is sufficiently small. The learning rate has to be chosen small enough to obtain convergence.

In a neural network, we want the difference between the output it returns and the output we expect to obtain, to be small. We define this difference as the error of the output. Using gradient descent we can minimise this error. We want a function of the error to depend on the parameters of our network. If we use gradient descent we can find the parameters that minimize our error. We define an error function, also called an objective function, $E : \mathbb{R}^d \to \mathbb{R}$. Here d is the number of parameters of the network, which in our case are the weights and biases.

Assume the network has an input vector p and an output vector y^p. We define the target t^p as the output which we think the network should have. One way to define the error is

$$E^p = \frac{1}{2} \sum_{k=1}^{M} (t_k^p - y_k^p)^2.$$

We can express the vector y_p as a function of the parameters of the network. After which, we can compute the derivative of the network using an algorithm called "backpropagation". Using the derivate, we can apply gradient descent and minimise the error for the training pair (y^p, t^p). Because this method uses training pairs, we call it a "supervised" method.

For an explanation of the backpropagation algorithm, we refer the reader to [3]. A description of the algorithm is also given in Appendix A.

3 Power System: Modelling and Optimal decision problem

In this section we describe the studied power system. We first give an overview of the five different components, along with notation for their corresponding variables and parameters. After this we will describe the cost functions as well as the constraints in the power system. The described components, cost functions and constraints will be used to formulate an optimisation problem. This model is taken from the problem formulation as described in [7]. We conclude the chapter with a Markov decision process formulation of the power system.

3.1 Description of the different components

For the studied power system we define five components. Each component has properties that we have to keep track of. We will discuss these components one by one.

- **Loads** demand power from the system. We define a set of loads $\mathcal{L}$. We define that load $k \in \mathcal{L}$ at time t demands power $P_{L_k,t} \in \mathbb{R}_{\geq 0}$.
- **Distributed generators** give power to the energy system and have an operating cost. We define a set of distributed generators $\mathcal{G}$. We define that distributed generator $i \in \mathcal{G}$ at time t has power $P_{G_i,t} \in \mathbb{R}_{\geq 0}$. We define the cost to operate the distributed generator i at time t as $C_{G_i,t} \in \mathbb{R}_{\geq 0}$.
- **Photovoltaic systems** give power to the energy system. As opposed to the generators, they do not have an operating cost. We define a set of photovoltaic systems $\mathcal{V}$. We define that photovoltaic system $m \in \mathcal{V}$ at time t has power $P_{V_m,t} \in \mathbb{R}_{\geq 0}$.
- **Energy storage systems** are a component in which we can store energy and retrieve it later. we define a set of energy storage systems $\mathcal{B}$. We define that energy storage system $j \in \mathcal{B}$ at time t has power $P_{B_j,t} \in \mathbb{R}$. This power is positive when the storage system is discharging. This power is negative when we charge the storage system. We define the total energy capacity of energy storage system j as $E_{B_j} \in \mathbb{R}_{\geq 0}$. We use the state of charge as the metric to measure what percentage of the total storage an energy storage system is currently holding. The stage of charge is expressed as a decimal between 0 and 1. We define the state of charge of storage system j at time t as $SOC_{B_j,t} \in [0, 1]$. We express the bounds on the state of charge of this system as the paramaters $\overline{SOC}_{B_j} \in \mathbb{R}_{\geq 0}$, for the maximum state of charge, and $\underline{SOC}_{B_j} \in \mathbb{R}_{\geq 0}$, for the minimum state of charge.
- **The utility grid** is the component where we can buy power from and sell power to. We define the utility grid as a singleton $\mathcal{N}$. We define the power of the utility grid at time t as $P_{N,t} \in \mathbb{R}$. This power is positive when we buy power from the grid. This power is negative when we sell power to the grid. We define the cost with which we can buy or sell a unit of power at time t as $C_{N,t} \in \mathbb{R}$. This cost is positive when we buy power from the grid. This cost is negative when we sell power to the grid.

Before we go in depth to describe the constraints on our components, we give a visualisation to describe the direction of power in the system. Figure 3.1 shows a flow chart of the power system. The arrows indicate the direction of power. The components are all shown in a rectangle. The centre piece of the power system, shown as an octagon, is not a component but it is the connection between the components. There are three directions where power can go out of our system. In particular it can go to the loads, to the utility grid and to the energy storage systems. There are four directions from which energy can come into our system. In particular it can come from the distributed generators, the photovoltaic systems, the utility grid and the energy storage systems. Note that the power from the distributed generators and from the utility grid have a positive cost for our system. Energy to the utility grid has a negative cost.

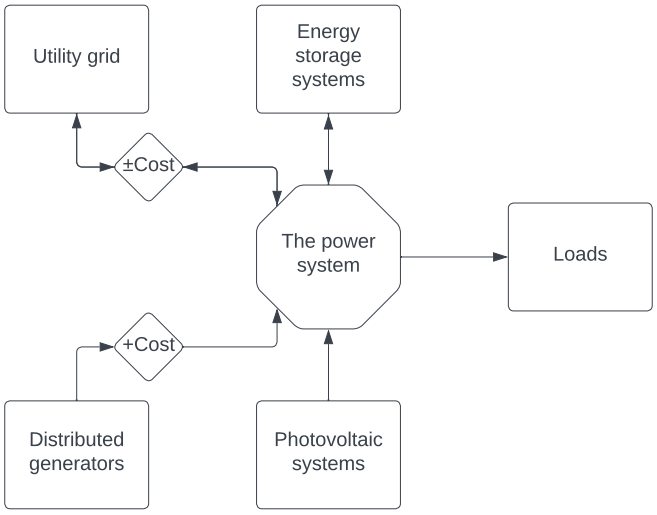

Figure 3.1: The relationship between the components in the studied power system.

3.2 Formulating the cost and constraints

In Section 3.1 we discussed the functionality of the components of the power system. We have yet to explain how the costs are determined or what constraints the components should satisfy. These constraints serve to make the model more realistic. In this section we will start by explaining the cost functions. After this we consider the components of the power system and explain their constraints.

- **The total cost** is what we want to minimize in our system. The total cost is the sum of the cost for enabling the distributed generators and the cost we get from the utility grid. We choose to model the cost of the distributed generator $i \in \mathcal{G}$ at time t as a quadratic function depending on the power of the generator, for some parameters $a_i, b_i, c_i \in \mathbb{R}$:

$$C_{G_i,t} = a_i \left(P_{G_i,t}\right)^2 + b_i P_{G_i,t} + c_i.$$

The cost $C_{N,t}$ of the utility grid is the price of a unit of power times the power of the grid. We use a constant $\rho_t \in \mathbb{R}_{\geq 0}$ to define the price of power at time t. To model that the buying and selling of power doesn't have the same price, we introduce a parameter $\beta \in \mathbb{R}$:

$$C_{N,t} = \begin{cases} \rho_t P_{N,t} & \text{for } P_{N,t} \geq 0, \\ \beta \rho_t P_{N,t} & \text{for } P_{N,t} < 0. \end{cases}$$

For the total cost we consider all the moments in time $t \in \mathcal{T}$ and distributed generators $i \in \mathcal{G}$. For practical reasons we don't want to optimize over a sum of infinitesimally small intervals of time. We introduce a discretization parameter Δt for the time. Hence, the total cost is:

$$\sum_{t \in \mathcal{T}} \sum_{i \in \mathcal{G}} \left(C_{G_i,t} + C_{N,t} \right) \Delta t.$$

- **The distributed generators** have limits on the power generation. Each generator will be able to generate power within certain bounds. For generator $i \in \mathcal{G}$ we define a lower bound for the power as $\underline{P}_{G_i}$ and an upper bound as $\overline{P}_{G_i}$. This leads to the following constraint:

  $$\underline{P}_{G_i} \leq P_{G_i,t} \leq \overline{P}_{G_i}.$$

  We also define parameters to model how much the generated power can change between consecutive moments in time. For this purpose for each generator $i \in \mathcal{G}$ we define $RD_i \in \mathbb{R}_{\geq 0}$ as the bound for lowering the power and $RU_i \in \mathbb{R}_{\geq 0}$ for increasing the power. This leads to the following two constraints:

  $$P_{G_i,t_{k+1}} - P_{G_i,t_k} \leq RU_{t_k},$$
  $$P_{G_i,t_k} - P_{G_i,t_{k+1}} \leq RD_{t_k}.$$

- **The energy storage systems** have similar limits as the distributed generators. For energy storage system $j \in \mathcal{B}$ we define a lower bound for the power as $\underline{P}_{B_j} \in \mathbb{R}$ and an upper bound as $\overline{P}_{B_j}\mathbb{R}$. Note that the power of an energy storage system can be negative, in the case that we are sending power to the energy storage system. This leads to the following constraint:

  $$\underline{P}_{B_j} \leq P_{B_j,t} \leq \overline{P}_{B_j}.$$

  When the storage system is used we update the value of its state of charge. We also model the efficiency of power storage systems using a parameter $\eta_B \in \mathbb{R}$. We obtain the following equality constraint for storage system $j \in \mathcal{B}$:

  $$SOC_{B_j,t_{k+1}} = SOC_{B_j,t_k} + \eta_B P_{B_j,t_{k+1}} \Delta t / E_{B_j}.$$

  We also define a constraint which models the state of charge limit of storage system j. For storage system j we define a lower bound for the state of charge as $\underline{SOC}_{B_j} \in [0,1]$ and an upper bound as $\overline{SOC}_{B_j} \in [0,1]$. This leads to the following constraint:

  $$\underline{SOC}_{B_j} \leq SOC_{B_j,t} \leq \overline{SOC}_{B_j}.$$

- **The utility grid** has one constraint which models the power limit of the grid. We define a bound for the power as $\overline{P}^C \in \mathbb{R}_{\geq 0}$ and this gives us the following constraint:

  $$-\overline{P}^C \leq P_{N,t} \leq \overline{P}^C.$$

  Note that we allow negative values of $P_{N,t}$ as this means that our power system is sending power to the utility grid.

- **The power balance constraint** is the last constraint we introduce. It describes the situation where the power system satisfies the loads. In our model this power comes from the distributed generators, photovoltaic systems, the utility grid and the energy storage systems. Ideally we know how to efficiently use all the power in our system and we can enforce an equality in this constraint:

$$\sum_{i\in\mathcal{G}} P_{G_i,t} + \sum_{m\in\mathcal{V}} P_{V_m,t} + P_{N,t} + \sum_{j\in\mathcal{B}} P_{B_j,t} = \sum_{k\in\mathcal{L}} P_{L_k,t}.$$

  In practice this constraint can be challenging and sometimes impossible to satisfy. The loads could ask for more power than is available in the system. Due to the constraints on the grid, it would not be possible to satisfy the demand by buying enough power.

3.3 Constrained Optimization Problem for the Optimal Management of the Power System

The descriptions for the components, as explained in Section 3.1, together with the descriptions for the cost and the constraints, as explained in Section 3.2, give us the following optimisation problem:

$$\min_{\{P_{B_j,t}\}_{j\in\mathcal{B}},\ \{P_{G_i,t}\}_{i\in\mathcal{G}}} \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{G}} \left(C_{G_i,t} + C_{N,t}\right)\Delta t$$

Where

$$C_{G_i,t} = a_i \left(P_{G_i,t}\right)^2 + b_i P_{G_i,t} + c_i, \quad \forall i \in \mathcal{G}$$

$$C_{N,t} = \begin{cases} \rho_t P_{N,t} & \text{for } P_{N,t} \geq 0, \\ \beta\rho_t P_{N,t} & \text{for } P_{N,t} < 0. \end{cases}$$

Such that

$$\begin{aligned}
&\sum_{i\in\mathcal{G}} P_{G_i,t} + \sum_{m\in\mathcal{V}} P_{V_m,t} + \sum_{j\in\mathcal{B}} P_{B_j,t} + P_{N,t} = \sum_{k\in\mathcal{L}} P_{L_k,t}, && \\
&\underline{P}_{G_i} \leq P_{G_i,t} \leq \overline{P}_{G_i} && \text{for } i \in \mathcal{G}, \\
&P_{G_i,t_{k+1}} - P_{G_i,t_k} \leq RU_i && \text{for } i \in \mathcal{G}, \\
&P_{G_i,t_k} - P_{G_i,t_{k+1}} \leq RD_i && \text{for } i \in \mathcal{G}, \\
&\underline{P}_{B_j} \leq P_{B_j,t} \leq \overline{P}_{B_j} && \text{for } j \in \mathcal{B}, \\
&SOC_{B_j,t_{k+1}} = SOC_{B_j,t_k} + \eta_B P_{B_j,t_{k+1}} \Delta t / E_{B_j} && \text{for } j \in \mathcal{B}, \\
&\underline{SOC}_{B_j} \leq SOC_{B_j,t} \leq \overline{SOC}_{B_j} && \text{for } j \in \mathcal{B}, \\
&-\overline{P}^C \leq P_{N,t} \leq \overline{P}^C. &&
\end{aligned}$$

This optimisation problem describes the power system as we study it. We want to minimise the total cost in the system while satisfying all of the constraints.
We assume that the power values $P_{L_k,t}$, for the loads, as well as $P_{V_m,t}$, for the photovoltaic systems, are known. This implies we do not optimise over their values. Note that in our system we want to satisfy the power balance constraint. If we know the power values $P_{G_i,t}$ of the distributed generators and $P_{B_j,t}$ of the energy storage systems, we can determine the value $P_{N,t}$, of the power of the utility grid, from the power balance equation. This is because every other variable in the equation is known. However, we mentioned before that in certain situations it

is impossible to satisfy the power balance constraint. In this case we will choose $P_{N,t} = \overline{P}^C$ or $P_{N,t} = -\overline{P}^C$, in order to minimise the unbalance. We conclude that we only need to optimise the power system problem over the variables $P_{G_i,t}$ of the distributed generators and $P_{B_j,t}$ of the energy storage systems. All of the other variables will either be known, or follow from the power balance constraint.
We have now formulated an optimisation problem which describes the power system. This problem has many variables. Each of the sets describing the components of the power system, apart from the utility grid, can have many elements. This makes the dimensionality of the problem high. None of the constraints are more complicated than a quadratic formula, which means we could use a conventional or commercial MIP solver to find the best values of $P_{G_i,t}$ (the power of the generator at time t) and $P_{B_j,t}$ (the power of the energy storage system at time t) for each $i \in \mathcal{G}$ and $j \in \mathcal{B}$. However, due to the high dimensionality, the problem is computationally expensive to solve. In the next section we formulate the power system as a Markov decision problem. This will allow us to use reinforcement learning to solve the optimisation problem. In Chapter 4 we will present reinforcement learning algorithms used for solving this Markov decision process. Figure 3.2 visualises our approach to solving the optimisation problem for the power system.

Figure 3.2: Approach to solving the optimisation problem.

3.4 Markov Decision Process Formulation of the Power System

As explained in Section 3.3 we model the optimisation problem for the power system as a Markov decision process. Remember that Section 2.2 characterized a Markov decision process (MDP) with a tuple

$$(\mathcal{T}, \mathcal{S}, \mathcal{A}, \mathcal{R}, p_d, \gamma),$$

where we have a set of times $\mathcal{T}$, set of states $\mathcal{S}$, set of actions $\mathcal{A}$, set of rewards $\mathcal{R}$, a dynamics function p_d and a discount rate γ. Below, we will specify each of these parameters for the power system problem. Once these definitions are established, we can construct an MDP for the power system problem.

Times. We define a set of times $\mathcal{T} \subseteq \mathbb{R}$. We have $0 \in \mathcal{T}$. We define a parameter called the "time-step" $\Delta t \in \mathbb{R}$. This time-step determines the other elements of $\mathcal{T}$. We have $\Delta t = t_i - t_{i-1} \in \mathbb{R}_{\geq 0}$ for $i \in \mathbb{N}_{\geq 1}$ for values of i up to $k \in \mathbb{N}$. This means our set $\mathcal{T}$ has a finite cardinality.

States. We define $\mathcal{S}$ as the set of all possible states. For every time $t \in \mathcal{T}$ we define a state $s_t \in \mathcal{S}$ to describe the state of our energy system. In this state we include the power of every

photovoltaic system and load (as $P_{V_m,t}$ and $P_{L_k,t}$ for $m \in \mathcal{V}$ and $k \in \mathcal{L}$). For both of these components we assume the power values will be known. Furthermore we include the power value $P_{N,t}$ of the utility grid as explained in Section 3.3. We also include the state of charge $SOC_{B_j,t}$ for each of the energy storage systems $j \in \mathcal{B}$. We define the state s_t at time $t \in \mathcal{T}$ as

$$s_t = (\{P_{L_k,t}\}_{k\in\mathcal{L}}, \{P_{V_m,t}\}_{m\in\mathcal{V}}, P_{N,t}, \{SOC_t\}_{j\in\mathcal{B}}).$$

Actions. We define $\mathcal{A}$ as the set of all possible actions. We include in our action the same variables as those we have to decide for in our optimisation problem as described in Section 3.3. We define an action at time t as $a_t \in \mathcal{A}$, where

$$a_t = (\{P_{B_j,t}\}_{j\in\mathcal{B}}, \{P_{G_i,t}\}_{i\in\mathcal{G}}).$$

Rewards. We define a set of rewards $\mathcal{R} \subseteq \mathbb{R}$. The rewards will be determined by a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ as in Definition 2.2.7. For parameters $\sigma_1, \sigma_2 \in \mathbb{R}$, we define this reward function as

$$\begin{aligned} r(s_{t_p}, a_{t_p}, s_{t_{p+1}}) &= r((\{P_{L_k,t_p}\}_{k\in\mathcal{L}}, \{P_{V_m,t_p}\}_{m\in\mathcal{V}}, P_{N,t_p}, \{SOC_{t_p}\}_{j\in\mathcal{B}}), \\ &\quad (\{P_{B_j,t_p}\}_{j\in\mathcal{B}}, \{P_{G_i,t_p}\}_{i\in\mathcal{G}}), \\ &\quad (\{P_{L_k,t_{p+1}}\}_{k\in\mathcal{L}}, \{P_{V_m,t_{p+1}}\}_{m\in\mathcal{V}}, P_{N,t_{p+1}}, \{SOC_{t_{p+1}}\}_{j\in\mathcal{B}})) \\ &= -\sigma_1 \left[\sum_{i\in\mathcal{G}} \left(C_{G_i,t_p} + C_{N,t_p}\right)\right] - \sigma_2 \Delta P_t. \end{aligned}$$

Here we define ΔP_t as the power unbalance

$$\Delta P_t = \left| \sum_{i\in\mathcal{G}} P^G_{i,t} + \sum_{m\in\mathcal{V}} P^V_{m,t} + P^N_t + \sum_{j\in\mathcal{B}} P^B_{j,t} - \sum_{k\in\mathcal{L}} P^L_{k,t} \right|.$$

Note that the reward function depends on the state and action because of the definitions for the cost functions and the power unbalance. Maximising this reward means that both the cost and power unbalance will be minimised.

Dynamics function. As in the theory of Definition 2.2.5 our MDP must have a dynamics function $p_d : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to [0, 1]$ where

$$p_d(s, a, s', r) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a]$$

and

$$\sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p_d(s, a, s', r) = 1, \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}.$$

We can now take two approaches to define the dynamics function p_d for the problem. We have a model-based approach. Following this approach we explicitly determine a value for each tuple in the domain of p_d. Alternatively we sample from a probability distribution. We also have a model-free approach. In this case we don't define these dynamics ourselves. Instead, our algorithms will learn these dynamics by interactions with the system. In our case this means that the dynamics function will be determined by a reinforcement algorithm as explained in Chapter 4.

4 Reinforcement learning algorithms

When we solved the reinforcement problem in Section 2.4, we assumed the dynamics function was known. At the end of Section 2.4 we mentioned the existence of solution methods for an unknown dynamics function. These methods will be presented in this section. We will describe how the reinforcement learning problem can be solved, in the case of an unknown dynamics function.

For each taken action, the model for the problem gives some reward in return. The methods in this section make use of these rewards to determine a policy. We make a distinction between action-value methods and policy-based methods. In action-value methods, we learn the action-value function. This function will then be used to select actions. In policy-based methods, we learn a policy without having to consider the values of the actions. The theory and notations in this section will be based upon [10]. For every topic we make clear which particular chapter is used as a reference.

We will first describe a method called "ϵ-greedy exploration". This will allow us to define a policy if we are given an action-value function. The reference we use for the theory is chapter 5.4 of [10]. After giving the definition, the following sections will describe action-value and policy-based methods.

4.1 ϵ-Greedy exploration

We are assuming that an action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is given. We can define a policy function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ by greedily taking the action with the highest value, for every state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$.

Definition 4.1.1. Given an action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We define greedy exploration, as formulating a policy function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ through

$$\pi(s, a) = \begin{cases} 1 & \text{if } a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\ q(s, a'), \\ 0 & \text{otherwise} \end{cases}.$$

If we had the optimal action-value function, Definition 4.1.1 allows us to define an optimal policy function. However, in practice we are only giving an estimate for this action-value function. Because we can only use an estimate, there is no guarantee that greedily taking actions gives an optimal policy. For example, the estimated action-value function could underestimate actions that lead to good states, which leads to the policy not considering these actions at all. In order to prevent this from happening, we add a small error term in Definition 4.1.1. This term allows us to explore action we wouldn't have considered otherwise. Consequently, we obtain the following definition.

Definition 4.1.2. Given an action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and $\epsilon \in [0, 1]$. We define ϵ-greedy exploration, as formulating a policy function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ through

$$\pi(s, a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\ q(s, a'), \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise.} \end{cases}.$$

4.2 Action-value methods

We will now consider action-value methods, which approximate the action-value function (defined in Definition 2.3.3). In practice, we do not know how the state of the system changes, because the dynamics function is unknown. However, if we do know the available actions, then we can use the action-value function to define a policy function. We can define a policy by greedily taking the actions with the highest value. Specifically, we will consider an action-value method called "Q-learning". We start by giving intuition and conclude with a definition. We follow chapter 6 of [10]. We note that alternative action-value methods exist. Which are also described in [10].

In order to get an estimate of the action-value function, we make use of experience. When the system is in a state S_{t_i} and an action A_{t_i} is taken, we get a reward $R_{t_{i+1}}$ and the state of the system changes to a state $S_{t_{i+1}}$. In this case the experience is a tuple

$$(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}}).$$

We can also define experience tuples with more states, actions and rewards. It can even be an entire episode. Recall that an episode is a sequence of rewards, states and actions for $k \geq 1$:

$$R_{t_0}, S_{t_0}, A_{t_0}, R_{t_1}, S_{t_1}, \ldots, R_{t_k}, S_{t_k}, A_{t_k}.$$

We are going to use this experience by comparing it to the values that our estimated value functions give.
One way to do this is by using temporal difference learning, abbreviated as TD. We first describe TD(0), which means that we only look one step ahead in the experience.
If we have a state S_{t_i}, the TD(0) comparison for the state-value function $v : \mathcal{S} \to \mathbb{R}$ (defined in Definition 2.3.2) is

$$R_{t_{i+1}} + \gamma v(S_{t_{i+1}}) - v(S_{t_i}).$$

We say that the value $v(S_{t_i})$ is compared to the target $R_{t_{i+1}} + \gamma v(S_{t_{i+1}})$. Note that a value function gives the expected return. Our target consists of the immediate next reward and bootstraps the value function for the remainder of the return.
Similarly, if we have a state S_{t_i}, an action A_{t_i} and discount factor $\gamma \in [0, 1]$, the TD(0) comparison for the action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is

$$R_{t_{i+1}} + \gamma q(S_{t_{i+1}}, A_{t_{i+1}}) - q(S_{t_i}, A_{t_i}).$$

We can make this more general and define a TD(n) comparison, for $n \in \mathbb{N}$.

Definition 4.2.1. Given $n \in \mathbb{N}$, a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, \ldots, S_{t_{i+n}}, A_{t_{i+n}}, R_{t_{i+1+n}})$ and a discount rate $\gamma \in [0, 1]$, the TD(n) comparison for the action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as

$$\sum_{k=0}^{n} \gamma^k R_{t_{i+1+k}} + \gamma^{n+1} q(S_{t_{i+1+n}}, A_{t_{i+1+n}}) - q(S_{t_i}, A_{t_i}).$$

4.2.1 SARSA

Using the previously explained TD(0) comparison, we can define an iterative method to update the action-value function. We take the difference of our expected return with the TD(0) target. This difference indicates whether the value of this state-action pair should be higher or lower. We use a learning rate to rescale this difference, and update the value function. Note that this algorithm is approximating the value function. The learning rate is important to achieve convergence to the real value function.

This method is called Sarsa which stands for "state, action, reward, state, action", which is the subsequence of an episode we use to update the value function.

Definition 4.2.2. Given a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}}, A_{t_{i+1}})$, discount rate $\gamma \in [0, 1]$ and a learning rate $\alpha \in]0, 1]$, we define Sarsa as updating the action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ in the pair (S_{t_i}, A_{t_i}) by

$$q(S_{t_i}, A_{t_i}) = q(S_{t_i}, A_{t_i}) + \alpha(R_{t_{i+1}} + \gamma q(S_{t_{i+1}}, A_{t_{i+1}}) - q(S_{t_i}, A_{t_i})).$$

In Sarsa we use the experience tuple to determine which state-action pair we are updating. We use the same tuple to determine our TD(0) target. In practice, we get this tuple by following a policy. Because we use the same tuple for both of these steps, we call Sarsa an "on-policy" method.

4.2.2 Q-learning

Another idea is to use the experience tuples only to determine which state-action pair we are updating. This allows us to formulate the TD(0) target differently. This is called an "off-policy" approach. An example is the Q-learning method. Here we update towards the maximum value of the next state-action pair.

Definition 4.2.3. Given a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$, discount rate $\gamma \in [0, 1]$ and a learning rate $\alpha \in]0, 1]$, we define Q-learning as updating the action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ in the pair (S_{t_i}, A_{t_i}) by

$$q(S_{t_i}, A_{t_i}) = q(S_{t_i}, A_{t_i}) + \alpha(R_{t_{i+1}} + \gamma \max_{a \in \mathcal{A}} q(S_{t_{i+1}}, a) - q(S_{t_i}, A_{t_i})).$$

Note that we can take the maximum of $q(S_{t_i}, a)$ for all $a \in \mathcal{A}$ because the function q is known. This function is our approximation to the "true" action-value function.

It can be shown that both Sarsa as well as Q-learning converge to the optimal action-value function, as described in [10]. It is worthwhile to mention the sufficient condition for Q-learning to have convergence, which is that the algorithm never stops updating the value of any state-action pair. In practice this means that multiple episodes of experience could be needed. We need these episodes to consider all state-action pairs. In other words, we need our algorithm to continue exploring. One way to obtain such episodes is by using the previously discussed ϵ-greedy exploration.

4.3 Policy-based methods

In the above discussion we considered action-value methods. As explained previously, there exists another approach to solving reinforcement learning. Namely, by using policy-based methods. These methods do not define a policy by greedily taking actions with respect to the value functions. Instead, they optimise the policy directly. This is achieved by parametrising the policy function. One way to update these parameters is through the use of gradient descent, as explained in Section 2.5.5. We will describe policy-based methods, in particular the actor-critic structure, as they are explained in chapter 13 of [10].

To make use of policy-based methods, we parametrise the policy function. An example of a parametrised function is a neural network. The parameters of the function will be the weights and biases of the network, as explained in Section 2.5. When we model the policy as a neural network, the input of the network will be a vector representing the state. The output will be

a vector that determines the next action. If there are a finite number of actions, we can use the output vector to determine probabilities for each action. When we have an infinite number of actions, we have multiple options. In some problems, every action is a set of numbers. The output vector then gives specific values for each part of the action. We can also let the network output parameters for a probability distribution. After which we take an action according to this distribution.

Actor-critic A well-known policy-based method is the actor-critic method. In these systems we approximate both the policy as well as a value function. The actor is the function that approximates the policy. The critic is the function that approximates a value function. We can use the values of the critic to update the parameters for the actor. The benefit of using a critic is that it greatly reduces the variance in the estimates of the actor's gradient. The downside is that it introduces a bias into these estimates (chapter 13.8 of [10]).

Twin critic The theory in [2] gives a way to reduce the previously mentioned bias. The idea is to use two critics. When updating the actor, we take the lowest value estimate of the value function.

Advantages In some problems the optimal policy is stochastic. If we used an ϵ-greedy based policy, we could not converge to this optimal policy. An advantage of policy approximation is that it can converge to a stochastic policy. As explained above, the output of our parametrised policy can directly give probabilities, or values for parameters of some probability distribution.

4.4 Deep Q-Network (DQN)

In this section we explore an algorithm called "Deep Q-Network" ("DQN"). This algorithm is a variation of Q-learning. DQN makes use of a parametrised action-value function and stores tuples of experience to repeatedly learn from. These ideas give good properties to the algorithm. More details can be found in chapter 16.5 of [10].
The section starts by describing two fundamental ideas for the DQN algorithm. These are called "fixed Q-targets" and making use of "experience replay". Both of these will be used in any DQN algorithm. After this we proceed by giving two variations of the DQN algorithm. First we consider the classical approach as explained in [8]. We conclude the chapter by giving the DQN algorithm as it is used in [7]. This is the algorithm we will be using to obtain the results in Section 5.

Fixing the targets. The first idea is called "fixed Q-targets". As mentioned before, the DQN algorithm parametrises the action-value function. Say at time t_i we define a parametrised action-value function $q(s, a; w_{t_i})$ with parameters w_{t_i}. In order to obtain the best approximated values for this action-value function, we tune these parameters. Fixing the Q-targets starts by defining a second set of target parameters $w_{t_i}^- \coloneqq w_{t_i}$. One way to change the parameters w_{t_i}, by using the action-value functions $q(s, a; w_{t_i})$ and $q(s, a; w_{t_i}^-)$, is done as follows:

$$w_{t_{i+1}} = w_{t_i} + \alpha \left[R_{t_{i+1}} + \gamma \max_{\alpha \in \mathcal{A}} q(S_{t_{i+1}}, a; w_{t_i}^-) - q(S_{t_i}, A_{t_i}; w_{t_i}) \right] \nabla q(S_{t_i}, A_{t_i}; w_{t_i}). \qquad ((1))$$

This is a modified Q-learning update for the parameters. It makes use of a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$ and the discount rate γ. We refer to chapter 10.1 of [10] for a discussion and derivation of this iteration.

After the update as in ((1)), we will have $w_t^- \neq w_t$. We keep updating using the target parameters. Every $K \in \mathbb{N}$ iterations, we update the targets and set $w_t^- = w_t$ again. This number K is a parameter which can be tuned to obtain better results. Note that in the equation above we are updating the parameters towards a difference with the target $R_{t_{i+1}} + \gamma \max_{\alpha \in \mathcal{A}} \ q(S_{t_{i+1}}, a; w_{t_i}^-)$.

Experience replay. The second method is called "experience replay". Note that in Definition 4.2.3 of Q-learning, the update of the action-value function makes use of a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$. Similarly, in the description of Q-learning given above, the update of the parameters makes use of a tuple $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$.

Experience replay consists of two steps. First, we take actions according to an ϵ-greedy policy and we obtain experience tuples $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$. After this, we store the tuples in a replay memory $\mathcal{D}$. We can uniformly sample a tuple from this memory, and use the tuple to update the action-value function in the Q-learning update. In this way, we update the values without any predefined order, which reduces the variance of the update.

The classical DQN algorithm. We are now ready to define the Deep Q-Network (DQN) algorithm. We first describe the algorithm as it is explained in [8]. DQN is a variation of Q-learning, with a parametrised action-value function and a replay memory $\mathcal{D}$. The algorithm learns the action-value function, which is defined in Definition 2.3.3. The algorithm initialises a parametrised action-value function $q(s, a; w_{t_i})$ with parameters $w_{t_i} \in \mathbb{R}^n$ for some $n \in \mathbb{N}$. We define copies $w_{t_i}^- \coloneqq w_{t_i}$ of these parameters. The following steps are repeated in order: [8]

Algorithm 1 The classical DQN algorithm.

1: Take actions according to an ϵ-greedy policy, as defined in 4.1.2. The ϵ-greedy policy is defined using the action-value function $q(s, a; w_{t_i}^-)$.
2: Store the obtained experience tuples $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$ in a replay memory $\mathcal{D}$.
3: Sample random tuples $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$ from $\mathcal{D}$.
4: Update the parameters w_{t_i} towards the fixed target parameters $w_{t_i}^-$ by making use of the sampled tuples. Here we use variations of Q-learning (Definition 4.2.3) and gradient descent (Section 2.5.5).
5: Every $K \in \mathbb{N}$ iterations, we update the target parameters $w_{t_i}^-$.

A variation of the DQN algorithm. We will now describe a variation of the DQN algorithm, as it is used in [7]. This variation makes use of an actor-critic structure, in particular it makes use of twin critics. We refer to Section 4.3 for a description of these concepts. The trained actor is used to obtain new experience tuples for the replay memory, as opposed to using an ϵ-greedy policy. The alternative Deep Q-Network (DQN) algorithm learns a policy function (Definition 2.3.1) as well the action-value function (Definition 2.3.3).
The algorithm initialises three tuples of weights and biases $(\mathbf{w}_1, \mathbf{b}_1)_{t_i}$, $(\mathbf{w}_{2_1}, \mathbf{b}_{2_1})_{t_i}$ and $(\mathbf{w}_{2_2}, \mathbf{b}_{2_2})_{t_i}$ at time $t_i \in \mathcal{T}$. We have a neural network $NN_1(\cdot; \mathbf{w}_1, \mathbf{b}_1)$ for the actor. We have two neural networks $NN_{2_1}(\cdot; \mathbf{w}_{2_1}, \mathbf{b}_{2_1})$ and $NN_{2_2}(\cdot; \mathbf{w}_{2_2}, \mathbf{b}_{2_2})$ for the twin critics. The critic neural networks are initialised with different parameters.
We define copies of the parameters as $(\bar{\mathbf{w}_1}, \bar{\mathbf{b}_1})_{t_i}$, $(\bar{\mathbf{w}_{2_1}}, \bar{\mathbf{b}_{2_1}})_{t_i}$ and $(\bar{\mathbf{w}_{2_2}}, \bar{\mathbf{b}_{2_2}})_{t_i}$, after which the following steps are repeated in order: [7]

Algorithm 2 A variation of the DQN algorithm.

1: Take actions according to the actor with target parameters $(\bar{\mathbf{w}}_1, \bar{\mathbf{b}}_1)_{t_i}$.
2: Store the obtained experience tuples $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$ in a replay memory $\mathcal{D}$.
3: Sample random tuples $(S_{t_i}, A_{t_i}, R_{t_{i+1}}, S_{t_{i+1}})$ from $\mathcal{D}$.
4: Update parameters $(\mathbf{w}_1, \mathbf{b}_1)_{t_i}$, $(\mathbf{w}_{2_1}, \mathbf{b}_{2_1})_{t_i}$ and $(\mathbf{w}_{2_2}, \mathbf{b}_{2_2})_{t_i}$ at time $t_i \in \mathcal{T}$ by minimising the loss.
5: For the actor the loss depends on the values of the twin critics with target parameters $(\bar{\mathbf{w}}_{2_1}, \bar{\mathbf{b}}_{2_1})_{t_i}$ and $(\bar{\mathbf{w}}_{2_2}, \bar{\mathbf{b}}_{2_2})_{t_i}$.
6: To update the critic neural networks, the loss is a function which compares the estimate values of the neural networks with the real reward obtained in the experience tuple. This is Q-learning, which we defined previously in Definition 4.2.3.
7: Every K iterations, we update the target parameters $(\bar{\mathbf{w}}_1, \bar{\mathbf{b}}_1)_{t_i}$, $(\bar{\mathbf{w}}_{2_1}, \bar{\mathbf{b}}_{2_1})_{t_i}$ and $(\bar{\mathbf{w}}_{2_2}, \bar{\mathbf{b}}_{2_2})_{t_i}$.

4.5 MIP-DQN

In the previous section we explained the Deep Q-Network (DQN) algorithm. In particular we gave two different implementations. We described the classical algorithm which only trains an action-value function. We also described a variation which makes use of an actor-critic structure. This is the implementation we will use to obtain the results in Section 5.

The DQN algorithm is capable of solving the problem defined in Section 3. However, the algorithm is not capable of strictly enforcing operational constraints. The trained critic neural network does not consider the power balance constraint. In practice we want the power balance constraint to be satisfied, otherwise our solution is not feasible. This shortcoming motives a new reinforcement learning algorithm called Mixed Integer Programming Deep Q-Networks (MIP-DQN). The MIP-DQN algorithm makes use of mixed integer programming to strictly enforce the power balance constraint. We will first present the algorithm, after which we explain how a neural network can be described as a mixed integer programming problem. We present the MIP-DQN algorithm as it is defined in [7].

The Mixed Integer Programming Deep Q-Networks algorithm repeats the following steps in order:

Algorithm 3 The MIP-DQN algorithm.

1: Train actor and critic neural networks according to the alternative DQN algorithm as it is described in Definition 4.4.
2: Describe the neural network for the critic as a mixed integer programming problem (MIP) as is explained in Section 4.5.1.
3: Add constraints to the MIP as is explained in Section 4.5.1.
4: Take actions according to the solved MIP.

In the MIP-DQN algorithm we model the action-value function as a neural network. We learn the parameters of this network by making use of a variation of the DQN algorithm. After having learned parameters for the action-value neural network, we redefine this network as a MIP. This allows us to add constraints to the output of the neural network. We will now explain how a neural network can be described as a MIP. We first describe the case of a single neuron and conclude with a description of a neural network.

4.5.1 Turning a neural network into a MIP

The idea of turning a neural network has been described in [1]. For more details we refer to this source. In short, we take each linear equation of the neural network and turn this into a constraint in our MIP. We maximise the MIP over the actions.

Single neuron. Let us consider the case of a single neuron. Let $k \in \{1, \dots, K\} \subseteq \mathbb{N}$. We assume this neuron is positioned in layer k of the network and has input values $x_i^{k-1} \in \mathbb{R}$ for $i \in \mathbb{N}_{\geq 1}$ up to $i = n_{k-1}$. Here $n_{k-1} \in \mathbb{N}$ is the number of neurons in layer $k-1$. We use $x_j^k \in \mathbb{R}$ to denote the output of this neuron. We assume the neuron has weights w_{ij}^{k-1} and a bias b_j^{k-1}. For an activation function f, the output of the neuron is then defined (as in Section 2.5) by

$$x_j^k = f\left(\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1}\right).$$

In our algorithm we use the ReLu function, defined as $f : \mathbb{R} \to \mathbb{R}$ by $f(x) = \max(0, x)$, as our activation function. This gives us the following definition for the output of the neuron:

$$x_j^k = \begin{cases} \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1}, & \text{when } \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

When we turn this equation into a linear constraint we have to enforce the zero case. We do this by using a decision variable z_j^k. We get:

$$\begin{aligned} \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} &= x_j^k - s_j^k \\ x_j^k, s_j^k &\geq 0 \\ z_j^k &\in \{0, 1\} \\ z_j^k = 1 &\Rightarrow x_j^k \leq 0 \\ z_j^k = 0 &\Rightarrow s_j^k \leq 0 \end{aligned}$$

Because we enforce either x_j^k or s_j^k to be 0 at all times, we will obtain the same behaviour as the ReLU function. If the output is positive, then we have output x_j^k and we have $s_j^k = 0$. But if the output is negative, then we set $x_j^k = 0$ and we use s_j^k such that the equality holds.

Neural network Now that we know how to turn the output of a single neuron into a linear constraint, we can do this for all of the neurons in the neural network. Because there is a lot of notation, we first give the optimisation problem and explain the notation afterwards. As described in [1] we get the following optimisation problem:

$$\min \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=1}^{K} \sum_{j=1}^{n_k} \gamma_j^k z_j^k.$$

Such that

$$\left.\begin{array}{l} \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - s_j^k \\ x_j^k, s_j^k \geq 0 \\ z_j^k \in \{0,1\} \\ z_j^k = 1 \Rightarrow x_j^k \leq 0 \\ z_j^k = 0 \Rightarrow s_j^k \leq 0 \end{array}\right\} k = 1, \ldots, K, j = 1, \ldots, n_k,$$

$$lb_j^0 \leq x_j^0 \leq ub_j^0, \quad j = 1, \ldots, n_0,$$

$$\left.\begin{array}{l} lb_j^k \leq x_j^k \leq ub_j^k \\ lb_j^k \leq s_j^k \leq ub_j^k \end{array}\right\} k = 1, \ldots, K, j = 1, \ldots, n_k.$$

Here we have neurons j in layer k, for $j \in \{1, \ldots, n_k\} \subseteq \mathbb{N}$ and $k \in \{1, \ldots, K\} \subseteq \mathbb{N}$. The parameters w_{ij}^{k-1} are the weights of the neural network, while the parameters b_j^{k-1} are the biases. We introduced parameters $c_j^k, \gamma_j^k, lb_j^0, ub_j^0, lb_j^k, ub_j^k \in \mathbb{R}$ for each neuron j in layer k. The parameters c_j^k and γ_j^k following from the loss function of the neural network. For $k \geq 1$ we define $lb_j^k = 0$ and $ub_j^k \in \mathbb{R}$. The parameters lb_j^0 and ub_j^0 should both be equal to the input values x_j^0 of the neural network. These input values x_j^0 can be used to add constraints to the optimisation problem. Note that in the MIP-DQN algorithm we want to maximise the output of this MIP over the actions. This gives us the following objective function:

$$\max_{a \in \mathcal{A}} \left(\min_{s \in \mathcal{S}} \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=1}^{K} \sum_{j=1}^{n_k} \gamma_j^k z_j^k \right).$$

Note that the variables x_j^0 define the input of the neural network. In the MIP-DQN algorithm we use the critic neural network to define the MIP, this neural network has the same inputs as the action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Because of this, the values for the actions and states will be given by the parameters x_j^0.

5 Numerical experiments and results

In this section, we conduct numerical experiments and results for solving the optimal decision problem for the studied power system (as explained in Chapter 3). We train neural networks for both the DQN algorithm (explained in Section 4.4) and the MIP-DQN algorithm (explained in Section 4.5). First, we present the results of this training. Then, we use the trained neural networks to manage the power system. The algorithms are compared based on three metrics: the achieved mean episode reward, the episode operation cost, and the power unbalance value.

Data for the parameter values of the power system as described in Chapter 3 has been made available by [7]. Data for the variables $\{P_{L_k,t_p}\}_{k\in\mathcal{L}}$ for the loads, $\{P_{V_m,t_p}\}_{m\in\mathcal{V}}$ for the photovoltaic systems, and the power cost ρ_t has also been made available by [7]. The time-step Δt we used is one hour. The data gives values for each hour during an entire year. We define an episode as taking 24 hours.

5.1 Training

We modelled the actor and the critic each with a neural network. We denote the neural network of the actor, with parameters ω, as π_ω. We denote the neural network for the critic, with parameters θ, as Q_θ. Note that in the DQN algorithm a critic network with fixed q-targets is being used (as described in Section 4.4). We denote the neural network for the critic with fixed q-target parameters θ^{target}, as $Q_{\theta^{\text{target}}}$.

We train the neural networks π_ω, Q_θ and θ^{target} using the DQN algorithm, as is described in Section 4.4. In our plots we will abbreviate the number of episodes by the letter "e". For example, "e100" corresponds with using 100 episodes. We also keep the random seed in python fixed. We abbreviate this seed using the letter "s". For example, "s1234" corresponds with using random seed 1234. Training of the neural networks makes use of two technical parameters called the "learning rate" and the "soft update tau". These will be abbreviated as "lr" and "sut" respectively. For example, "lr0.0001" corresponds with using a learning rate of 0.0001. And lastly, "sut0.01" corresponds with using a soft update tau of 0.01. We have varied between the number of episodes. We trained networks for 10, 100 and 1000 episodes. When training the networks using 1000 episodes, we also varied the learning rate and the soft update tau.

5.1.1 Metrics

During the training we have kept track of five metrics. Two of these metrics are strictly used to measure the training. These are the values for the actor loss function and the values for the critic loss function. The other three metrics were obtained by deploying the network for the actor during the training. This gave us values for the achieved mean episode reward, the episode operation cost and the value for the power unbalance. Having these values of the actor network enables us to compare the other algorithms with the actor. It also enables us to measure if the neural network for the actor is improving in its perfomance.
We used the WandB python package to keep track of the metrics. Because we have used five metrics, the number of steps in our plots will be five times the number of episodes we used for training. This is the case because we kept track of all five metrics after each episode. Figure 5.1

visualises that 5 steps were taking for each episode. The values on the vertical axis correspond to the episode number.

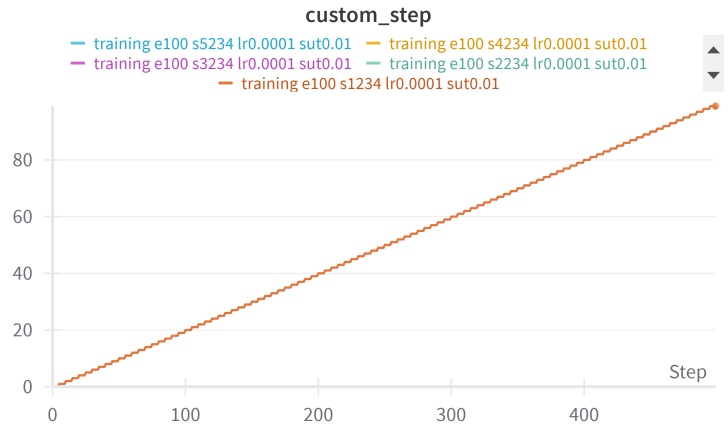

Figure 5.1: Steps taken in each episode.

5.1.2 Actor loss

During the training we kept track of the loss function for the neural network of the actor π_ω. This loss function is defined as $\text{Loss}_{\text{Actor}} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ by

$$\text{Loss}_{\text{Actor}}(S_t, A_t) \coloneqq -Q_{\theta^{\text{target}}}(S_t, A_t),$$

where S_t is the state at t and A_t is the action at time $t \in \mathcal{T}$. By minimising this loss function, we maximise the the value of the actions that the actor takes. If the loss function is high it means the actor is taking low value actions. In Figure 5.2 we see plots for the actor loss when we use 100 episodes. In Figure 5.3 we see plots for the actor loss when we use 1000 episodes.

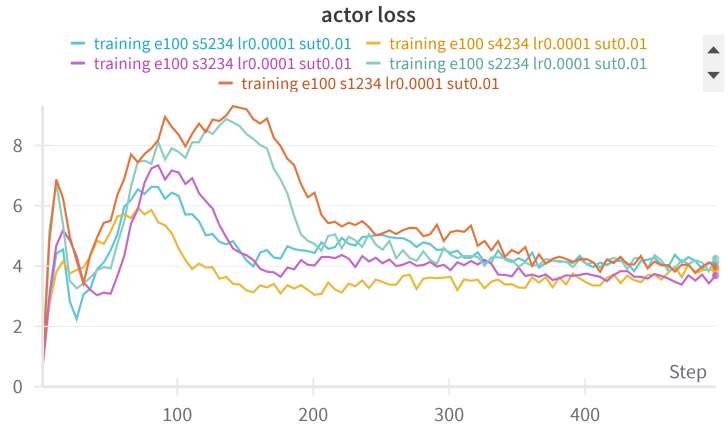

Figure 5.2: Actor loss for 100 episodes

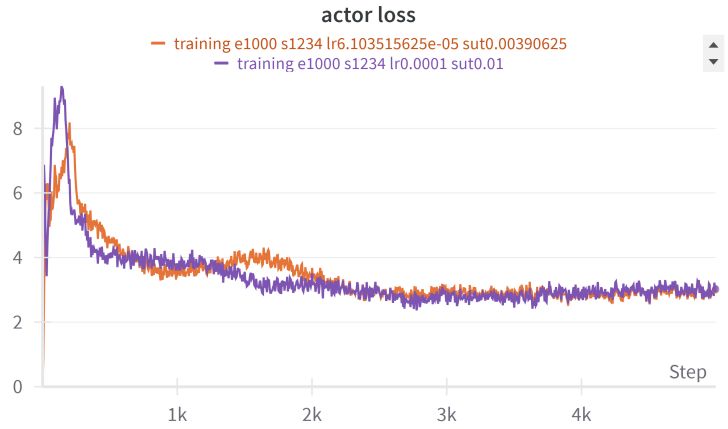

Figure 5.3: Actor loss for 1000 episodes

Observations. For the actor loss, good performance is indicated by a low value. First, we consider the results of the networks trained using 100 episodes of experience. In Figure 5.2, we observe that for all random seeds, the loss function value shows high variance during the first 150 steps. After this period, the plots seem to stabilize.

In Figure 5.3, we observe a similar pattern. The loss function values vary significantly during the initial steps but stabilize after many steps. However, the value towards which the plots converge is lower than the converged value observed after 500 steps in Figure 5.2. Consequently, we anticipate that the neural networks trained with 1000 episodes of experience will perform better than those trained with only 100 episodes.

5.1.3 Critic loss

During the training we kept track of the loss function for the neural networks of the critic Q_θ. This loss function is defined as $\text{Loss}_{\text{Critic}} : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \to \mathbb{R}$ by

$$\text{Loss}_{\text{Critic}}(S_{t_i}, A_{t_i}, R_{t_i}, S_{t_{i+1}}) := \text{SmoothL1Loss}(Q_\theta(S_{t_i}, A_{t_i}),\ R_{t_i} + \gamma Q_{\theta^{\text{target}}}(S_{t+1}, \pi_\omega(S_{t+1}))),$$

where we use the function $\text{SmoothL1Loss} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ defined by [9]

$$\text{SmoothL1Loss}(x, y) = \begin{cases} \frac{1}{2}(x-y)^2 & \text{if } |x-y| < 1 \\ |x-y| - 0.5 & \text{otherwise} \end{cases}.$$

where S_t is the state at t and A_t is the action at time $t \in \mathcal{T}$. By minimising this loss function, we minimise the difference between the action-value estimate and a bootstrap of the estimate using a real reward R_{t_i}. If the loss function is high it means the critic estimates are very different from the reward that the power system gives. In Figure 5.4 we see plots for the critic loss when we use 100 episodes. In Figure 5.5 we see plots for the critic loss when we use 1000 episodes.

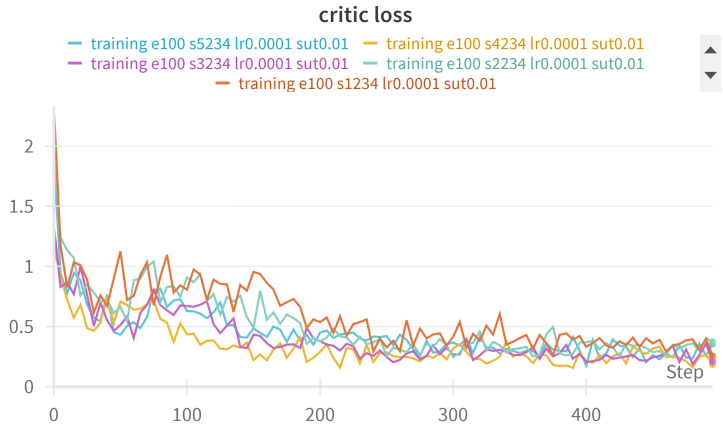

Figure 5.4: Critic loss for 100 episodes

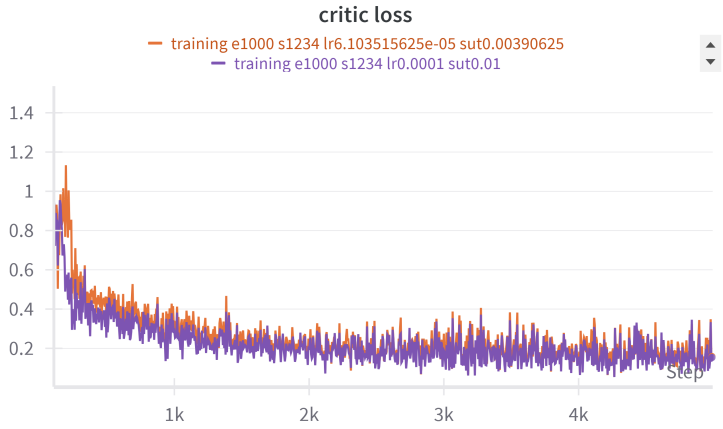

Figure 5.5: Critic loss for 1000 episodes

Observations. For the critic loss, good performance is indicated by a low value. First, we consider the results of the networks trained using 100 episodes of experience. In Figure 5.4, we observe that for all random seeds, the loss function value decreases consistently throughout the training period.

In Figure 5.5, we observe a similar pattern. The loss function values are initially high but continue to decrease for many steps. However, the plots seem to converge after 2000 steps, after which the values no longer improve and instead show high variance. A possible explanation is that the neural networks have reached a local optimum for the loss function.

5.2 Deployment

We have tested the deployment of three different approaches to solving the optimisation problem of the power system. First of all, we have deployed the neural network π_ω for the actor during the training. After training the neural networks we have deployed the neural network $\mathbb{Q}_\theta$ of the critic. Making use of this critic is equivalent to using the DQN algorithm as explained in Section 4.4. Lastly, we made use of the new MIP-DQN algorithm, as explained in Section 4.5. We have three metrics to compare the approaches on, as explained in Section 5.1.1. These metrics are the achieved mean episode reward, the episode operation cost and the value for the power unbalance.

5.2.1 Mean episode reward

100 episodes. First of all we consider the neural networks that have been trained using 100 episodes of experience. Figure 5.6 shows the mean episode reward of the actor neural network during the training. Figure 5.7 shows the mean episode reward of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.8 shows the mean episode reward of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

s1234 -2.7797145331379967,

s2234 -6.685039587438503,

s3234 -20.936838588915773,

s4234 -6.671462592111361,

s5234 -3.4398842968944803.

Here we denoted the different value for each random seed.

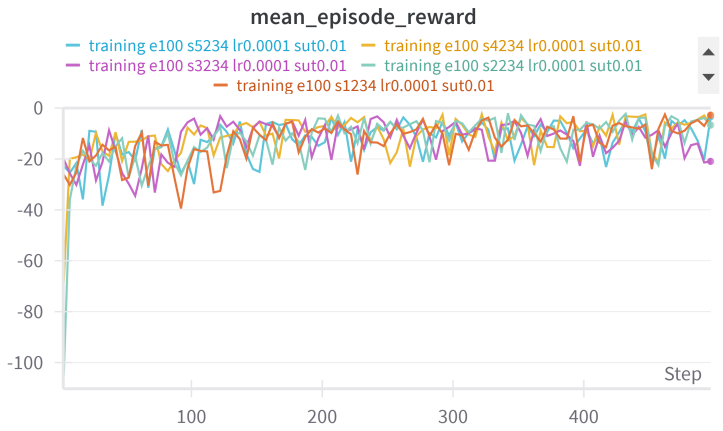

Figure 5.6: DQN actor - Mean episode reward for 100 episodes

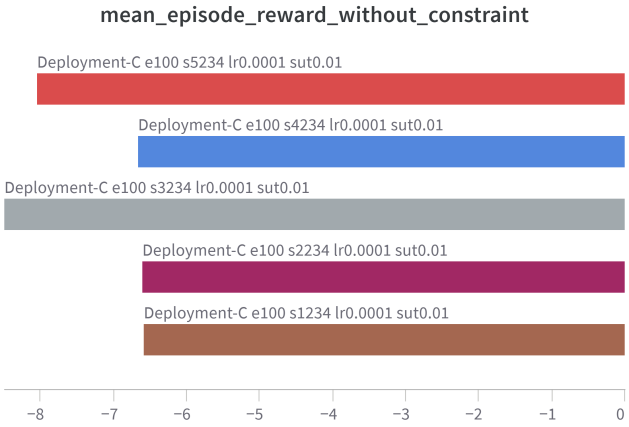

Figure 5.7: DQN critic - Mean episode reward for 100 episodes

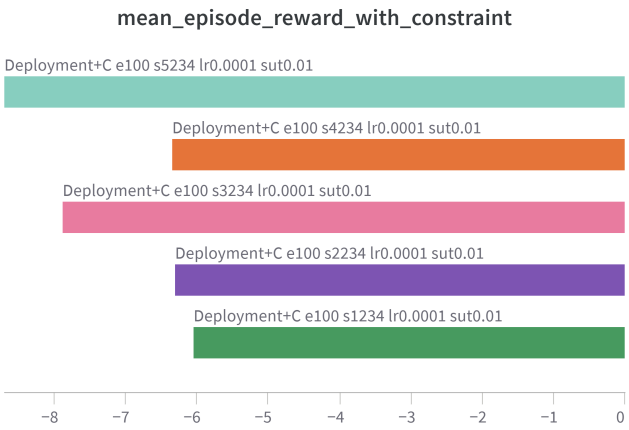

Figure 5.8: MIP-DQN: Mean episode reward for 100 episodes

1000 episodes. We consider the neural networks that have been trained using 1000 episodes of experience. Figure 5.9 shows the mean episode reward of the actor neural network during the training. Figure 5.10 shows the mean episode reward of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.11 shows the mean episode reward of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

lr0.0001 -7.833995437507792,

lr6.103515625e-05 -8.126822548935843,

Here we denoted the different value for each learning rate.

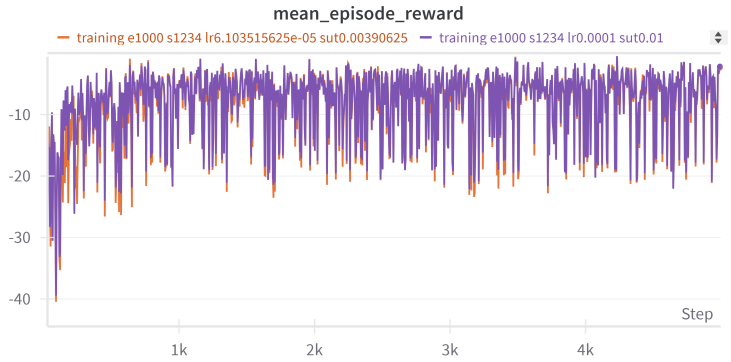

Figure 5.9: DQN actor - Mean episode reward for 1000 episodes

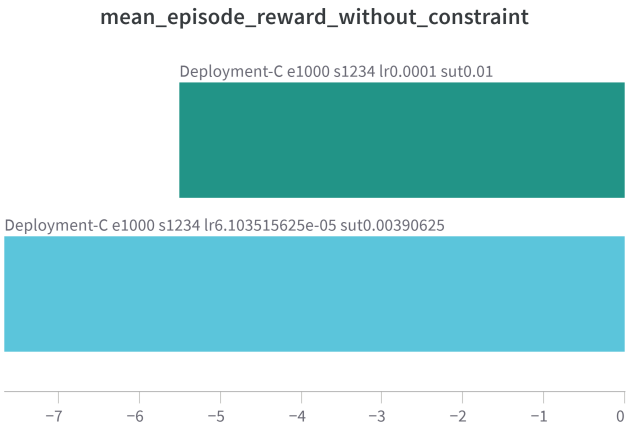

Figure 5.10: DQN critic - Mean episode reward for 1000 episodes

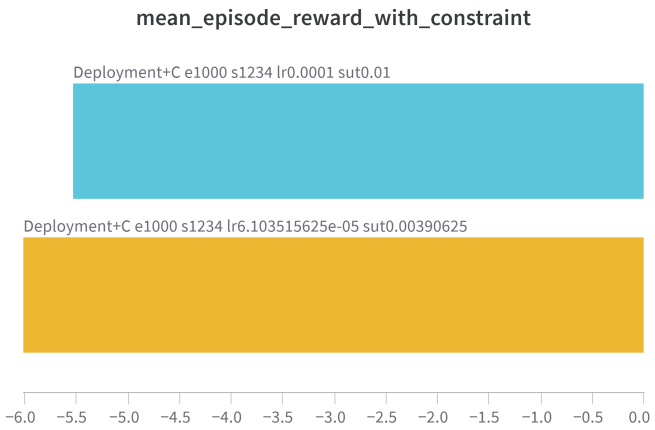

Figure 5.11: MIP-DQN: Mean episode reward for 1000 episodes

Observations. For this metric, good performance is indicated by a high value for the mean episode reward. First, we consider the results of the networks trained using 100 episodes of experience. In figures 5.6, 5.7 and 5.8 we observe that with random seeds S1234 and S5234, the actor neural network performs best. For random seeds S2234 and S4234, the performance is similar across all approaches. However, for random seed S3234, the actor produces a much worse result than the DQN or MIP-DQN algorithms.

When we consider figures 5.9, 5.10 and 5.11, we observe that using 1000 episodes of experience, the MIP-DQN algorithm achieves the highest mean episode reward, while the actor neural network performs the worst. Nonetheless, there are no significant outliers for any of the approaches.

5.2.2 Episode operation cost

100 episodes. First of all we consider the neural networks that have been trained using 100 episodes of experience. Figure 5.12 shows the episode operation cost of the actor neural network during the training. Figure 5.13 shows the episode operation cost of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.14 shows the episode operation cost of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

s1234 5559.429066275992,

s2234 13370.079174877012,

s3234 41873.67717783154,

s4234 13342.92518422272,

s5234 6879.768593788961.

Here we denoted the different value for each random seed.

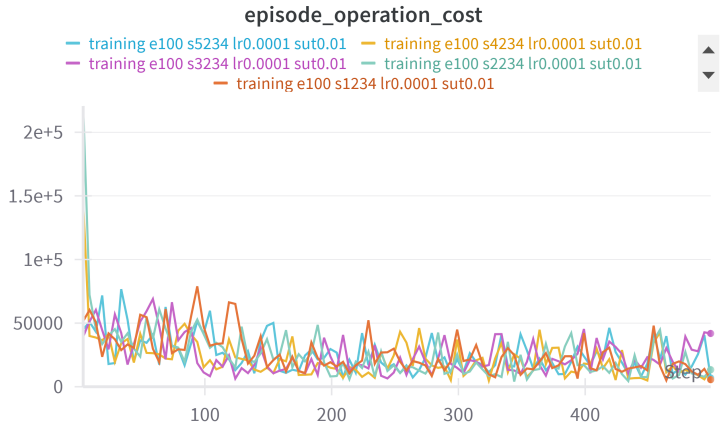

Figure 5.12: DQN actor - Episode operation cost for 100 episodes

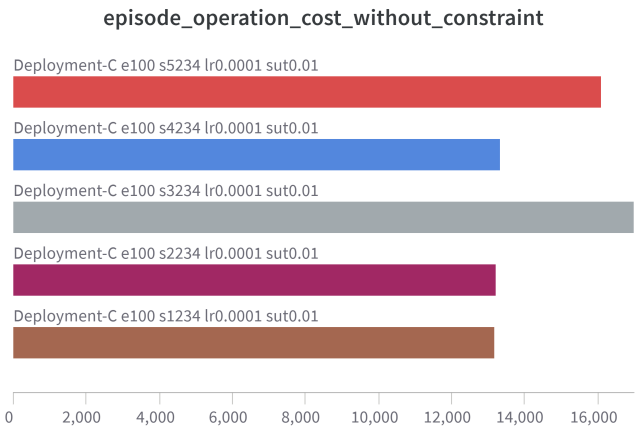

Figure 5.13: DQN critic - Episode operation cost for 100 episodes

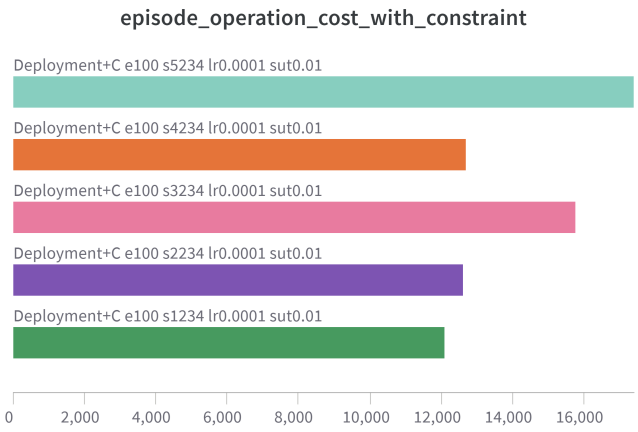

Figure 5.14: MIP-DQN: Episode operation cost for 100 episodes

1000 episodes. We consider the neural networks that have been trained using 1000 episodes of experience. Figure 5.15 shows the episode operation cost of the actor neural network during the training. Figure 5.16 shows the episode operation cost of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.17 shows the episode operation cost of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

lr0.0001 15667.990875015585,

lr6.103515625e-05 16253.645097871686,

Here we denoted the different value for each learning rate.

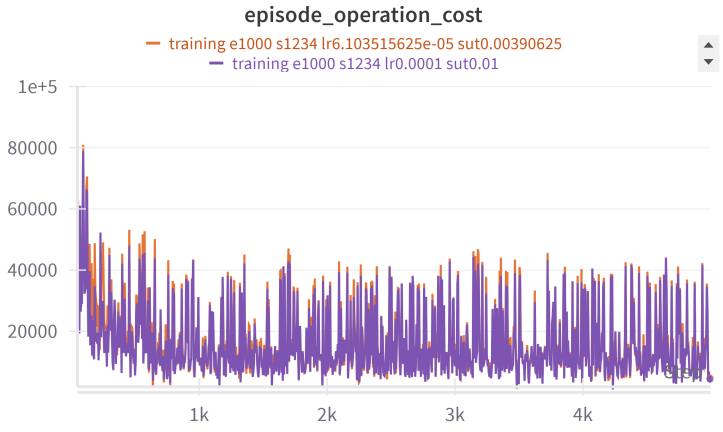

Figure 5.15: DQN actor - Episode operation cost for 1000 episodes

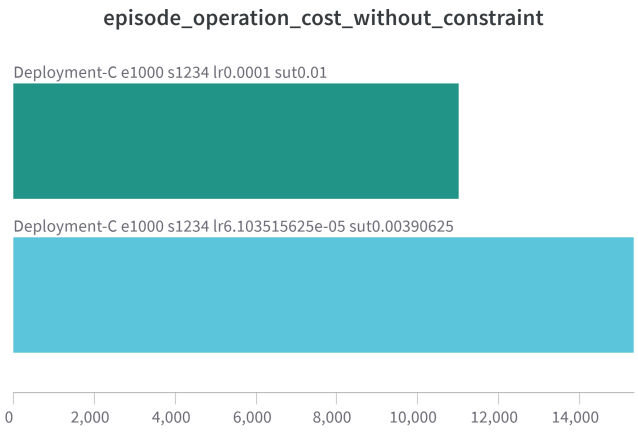

Figure 5.16: DQN critic - Episode operation cost for 1000 episodes

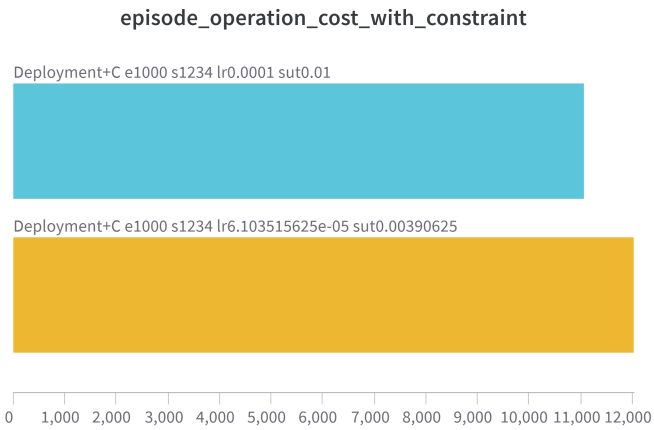

Figure 5.17: MIP-DQN: Episode operation cost for 1000 episodes

Observations. For this metric, good performance corresponds to a low value for the episode operation cost. We first consider the networks trained using 100 episodes of experience. In figures 5.12, 5.13 and 5.14, we observe that the performance of the actor for seeds S1234 and S5234 is better than in the other approaches. However, the actor has a much worse result for random seed S3234. For seeds S2234 and S4234, all three approaches yield similar results.

We now consider the results in figures 5.15, 5.16 and 5.17. We observe that when using 1000 episodes of experience, the MIP-DQN algorithm produces superior results with a smaller learning rate compared to the DQN algorithm. However, with a larger learning rate, both algorithms perform similarly. The actions decided by the neural network of the actor result in the highest cost.

5.2.3 Power unbalance

100 episodes. First of all we consider the neural networks that have been trained using 1000 episodes of experience. Figure 5.18 shows the power unbalance of the actor neural network during the training. Figure 5.19 shows the power unbalance of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.20 shows the power unbalance of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

s1234 33.802986922951206,

s2234 368.2289462842627,

s3234 1172.1069923925718,

s4234 92.57460528666269,

s5234 145.65432950093754.

Here we denoted the different value for each random seed.

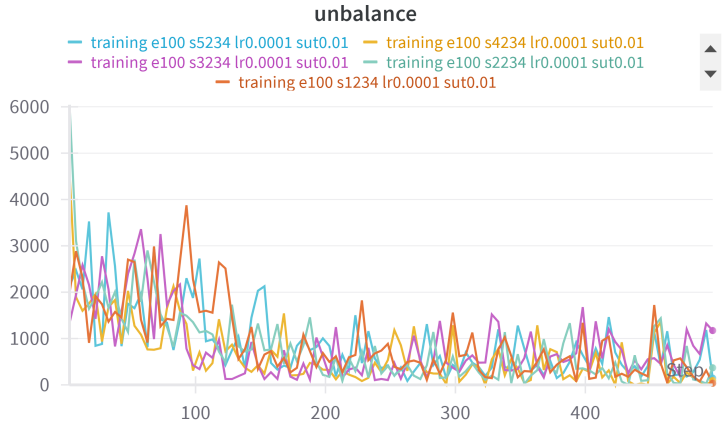

Figure 5.18: DQN actor - Power unbalance for 100 episodes

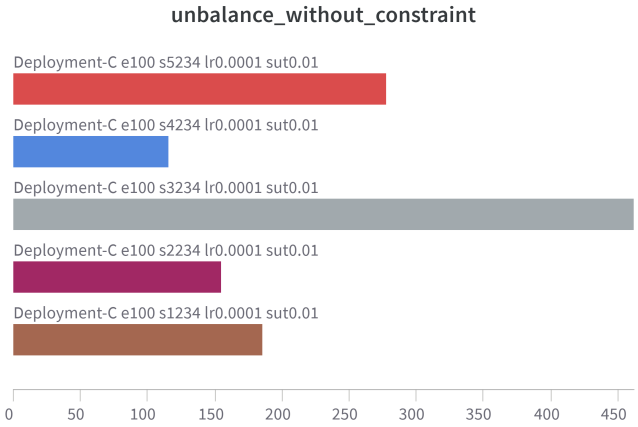

Figure 5.19: DQN critic - Power unbalance for 100 episodes

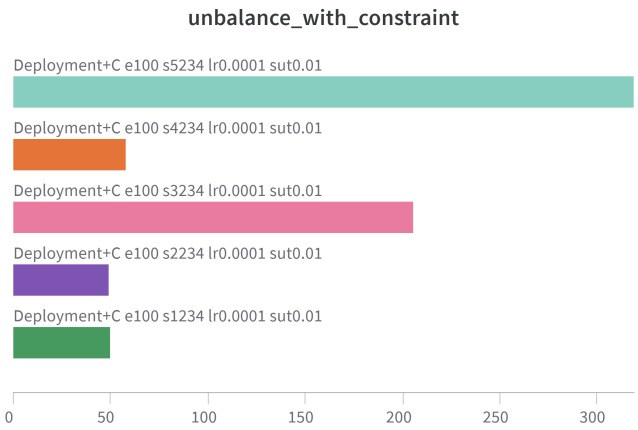

Figure 5.20: MIP-DQN: Power unbalance for 100 episodes

1000 episodes. We consider the neural networks that have been trained using 1000 episodes of experience. Figure 5.21 shows the power unbalance of the actor neural network during the training. Figure 5.22 shows the power unbalance of the DQN algorithm, equivalent to the critic neural network, after the training. Figure 5.23 shows the power unbalance of the MIP-DQN algorithm, equivalent to the critic neural network with the power balance constraint, after the training. The final values for the actor neural network are as follows:

lr0.0001 5.3147950181199 34,

lr6.103515625e-05 272.76747611587405,

Here we denoted the different value for each learning rate.

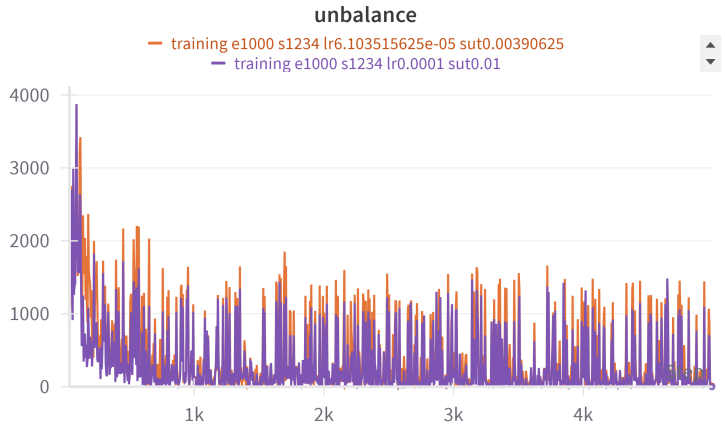

Figure 5.21: DQN actor - Power unbalance for 1000 episodes

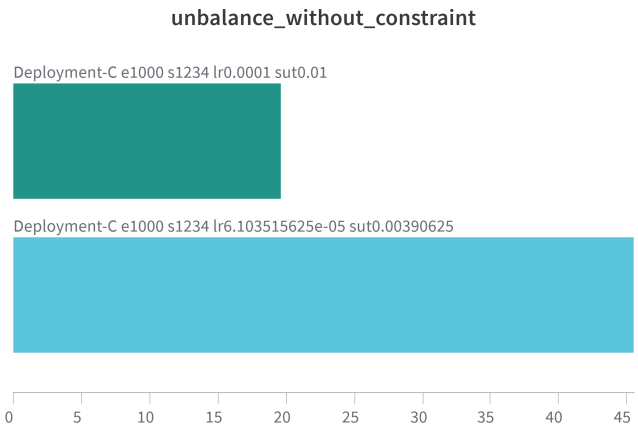

Figure 5.22: DQN critic - Power unbalance for 1000 episodes

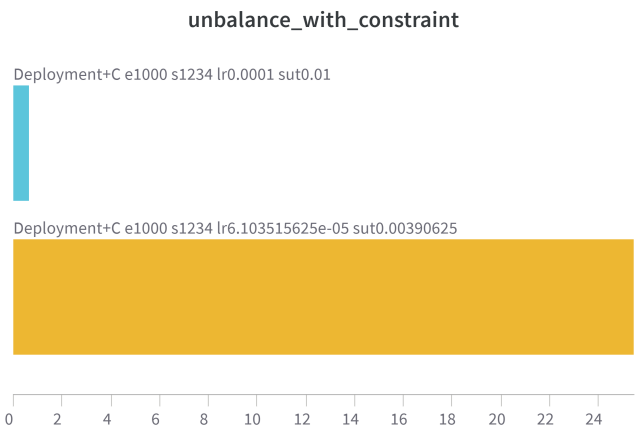

Figure 5.23: MIP-DQN: Power unbalance for 1000 episodes

Observations. For this metric, good performance is indicated by a low value for the power unbalance. We first consider figures 5.18, 5.19 and 5.20. Surprisingly, when trained using 100 episodes of experience and using random seeds S1234 or S5234, the neural network for the actor produces the lowest power unbalance during an episode, even lower than the MIP-DQN algorithm, which is designed to enforce this constraint. However, for the other three tested random seeds, the actor performs the worst. In particular, for seed S3234, the actor's power unbalance value is much higher. For seeds S2234 and S4234, the MIP-DQN algorithm yields the lowest power unbalance.

Lastly, we observe the results in figures 5.21, 5.22 and 5.23. When considering training with 1000 episodes of experience, we see that the MIP-DQN algorithm performs the best. The power unbalance after one episode is almost zero with the larger learning rate. The DQN critic's results are similar for both learning rates. The DQN actor's results vary significantly between learning rates; with the higher learning rate, the power unbalance is low, but not better than the MIP-DQN results. The highest power unbalance value is produced by the actor with a smaller learning rate.

5.3 Conclusion

With the produced results in sections 5.1 and 5.2, we can give a comparison for the performance of the actor neural network, the DQN critic and the MIP-DQN algorithm. The metrics we compare the algorithms on are the mean episode reward, the episode operation cost and the power unbalance. We conclude this chapter with an overview and discussion of the performance of the three approaches.

The actor neural network trained using the DQN algorithm showed a high variance in its produced results. This sometimes lead the neural network to have a better performance than the other algorithms. When we considered the plots of the actor during the training, this variance was also visible. For all metrics we saw that the actor improved during the first 20 episodes, after which the values oscillated for the remainder of the training, without much improvement. These observations suggest that one should conduct many runs and then take the average to reduce uncertainty.

The final results produced by the critic neural network, using the DQN algorithm, appeared to have a lower variance. For all metrics the produced results were close together. For most of the random seeds, the DQN critic had a similar or better result than the DQN actor.

The MIP-DQN algorithm produced similar, but slightly better results, as the DQN critic for the mean episode reward and episode operation cost. The values for the results were similar for all random seeds, with a comparable variance. Perhaps this is surprising, because an additional constraint has been added to the critic neural network, decreasing the cardinality of the feasible set of the MIP. However, adding this constraint also means we add additional information to the MIP-DQN algorithm when compared to the DQN algorithm. Our hypothesis is that this additional information makes the MIP-DQN algorithm take decisions that improve the state in the long term, while the DQN algorithm takes action which only maximise the reward given the current state. Due to the inherent unpredictability of the power system, the MIP-DQN algorithm therefore shows a better performance.

Lastly, we considered the power unbalance. As expected, the MIP-DQN algorithm acquired the lowest values for the power unbalance. What might be surprising is that the power balance constraint was not strictly enforced. We offer an explanation considering the constraints on the utility grid. When a power unbalance exists in the system, the algorithm will have to buy power from the utility grid in order to satisfy the demand. However, due to the upper bound on the power of the utility grid, it will not always be possible to buy enough power to satisfy the demand. Therefore, even though the algorithm considers the constraint, it is not able to maintain an equality in the power balance.

6 Discussion and Conclusion

In this thesis, we have studied a novel approach to reinforcement learning called MIP-DQN. We have explained the fundamental mathematical concepts required for understanding reinforcement learning, including the Markov decision process, state-value and action-value functions, and the Bellman equations. We have discussed methods for policy evaluation and policy improvement, enabling us to find the optimal policy when the dynamics function of the Markov decision process is known. Additionally, we have explained the workings of neural networks and how their parameters can be improved via gradient descent.

Building on the mathematical foundations, we have developed a model for the power system. Initially, we provided a non-linear programming formulation for the power system, detailing each component and constraint. Following this, we formulated the optimization problem for the power system as a Markov decision process.

We have explained algorithms capable of finding a good policy for managing the power system, covering advanced methods such as ϵ-greedy exploration, SARSA, Q-learning, and the actor-critic structure. These methods were used to define the Deep Q-Networks (DQN) algorithm, which employs fixed Q-targets and replay memory. We noted that a downside of the DQN algorithm is its inability to enforce the power balance constraint for the produced actions.

To address this, we introduced the new MIP-DQN algorithm. This algorithm utilises the DQN approach to train neural networks for the actor and critic. It then finds the highest value action by formulating the critic neural network as a mixed integer programming (MIP) problem, incorporating the power balance constraint into the MIP formulation. This allows us to find actions that satisfy the power balance constraint.

Finally, we presented results showcasing the increased performance of the MIP-DQN algorithm compared to the DQN algorithm. The metrics used to reach this conclusion were the mean episode reward, episode operation cost, and power unbalance. We first discussed the training of the actor and critic neural networks, which were then deployed to produce results for each metric.

Our analysis demonstrated the increased performance of MIP-DQN. We observed that the power balance is not always strictly enforced, even in the MIP-DQN algorithm. This occurs because there are states where it is impossible to maintain a zero power unbalance; the loads may demand more power than can be supplied due to constraints on the components of the power system. Nevertheless, the MIP-DQN algorithm consistently maintained good results across all metrics, achieving a significantly lower power unbalance.

The variance of the metrics during the training of the neural networks was large. We attempted different learning rates to reduce the oscillation but were unsuccessful. We suggest that further improvements should be made to lower this variance by optimally tuning the learning rate, and other parameters in the neural networks.

The power system we studied could be further improved. It did not account for uncertainty and noise in power dynamics, such as those from photovoltaic sources. Including these factors would make the decision problem more realistic and accurately reflect power uncertainty.

Lastly, regarding the discussed metrics, the results indicate better performance of the MIP-DQN algorithm compared to DQN. However, more certainty would be gained by repeating the experiments and taking the average.

Bibliography

[1] Matteo Fischetti and Jason Jo. "Deep neural networks and mixed integer linear optimization". In: *Constraints* 23.3 (2018), pp. 296–309.

[2] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[3] Kevin Gurney. *An introduction to neural networks*. CRC press, 2018, pp. 34–56.

[4] Andrej Krenker, Janez Bešter, and Andrej Kos. "Introduction to the artificial neural networks". In: *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech* (2011), pp. 1–18.

[5] Jiři Matoušek and Bernd Gärtner. *Understanding and using linear programming*. Vol. 1. Springer, 2007, pp. 1–31.

[6] Jeff M Phillips. "Gradient Descent". In: *Mathematical Foundations for Data Analysis* (2021), pp. 125–142.

[7] Hou Shengren et al. "Optimal energy system scheduling using a constraint-aware reinforcement learning algorithm". In: *International Journal of Electrical Power & Energy Systems* 152 (2023), p. 109230. ISSN: 0142-0615. DOI: https://doi.org/10.1016/j.ijepes.2023.109230. URL: https://www.sciencedirect.com/science/article/pii/S0142061523002879.

[8] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015.

[9] *SmoothL1Lossx2014; PyTorch 2.3 documentation — pytorch.org*. https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html. [Accessed 17-06-2024].

[10] R.S. Sutton and A. Barto. *Reinforcement learning: An introduction*. Cambridge, Massachusetts; London, England, 2020.

[11] Jinming Zou, Yi Han, and Sung-Sau So. "Overview of artificial neural networks". In: *Artificial neural networks: methods and applications* (2009), pp. 16–21.

A Backpropagation

In this appendix we describe the "backpropagation" algorithm. We use this algorithm to compute the gradient of an error function with respect to a feed-forward neural network. After which we can apply gradient descent to find parameters which minimise the error. This is a supervised algorithm because we define the error using targets. Given an input of a neural network, a target is a vector which we think the network should have as output. We describe backpropagation as it is explained in [3].

For the explanation of backpropagation we consider a special case. The network we consider has fully connected layers. We also consider an error which only depends on the weights of the network. Furthermore, we assume the network has the smooth sigmoid function σ as the activation function at all neurons. The case where the error also depends on the biases follows similarly. The case where there is no smooth activation function follows similarly as well. In both cases, extra care has to be taken when computing the gradient.

We start by making the error explicit. For weights $\{w_1, \ldots, w_n\}$ and $n \in \mathbb{N}$, we have $E = E(w_1 \ldots, w_n)$. For an input vector p we define the output of the network as y^p. We define the target t^p as the output which we think the network should have.
We define the error E^p for a training pattern (y^p, t^p) as

$$E^p = \frac{1}{2}\sum_{k=1}^{M}(t_k^p - y_k^p)^2.$$

As explained above, we define the error as the difference in the output. We take the square of each element, so the values are always non-negative. The factor $\frac{1}{2}$ is not necessary, but it makes the future equations more clear.

We first consider a network without hidden layers. It only has an input and an output layer. Consider the matrix W of weights for the neurons in the output layer, and let b be a vector of the biases. We have $y^p = \sigma(Wp + b)$. Define $a^p = Wp + b$. Note that we made use of the fact that the layer is fully connected here. This means that every neuron has the same input p. Lastly, define σ' as the derivative of σ.
By repeatedly using the chain rule, we can now take the derivative of E^p for an element w_{ij} of W. We obtain

$$\begin{aligned}
\frac{\partial E^p}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{k=1}^{M}(t_k^p - y_k^p)^2\right) \\
&= \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{M}\frac{1}{2}(t_k^p - \sigma(w_k p + b_k))^2\right) \\
&= (t_i^p - \sigma(w_i p + b_i))\frac{\partial}{\partial w_{ij}}(t_i^p - \sigma(w_i p + b_i)) \\
&= (t_i^p - \sigma(w_i p + b_i)) \cdot -\sigma'(w_i p + b_i)\frac{\partial}{\partial w_{ij}}(w_i p + b_i) \\
&= -(t_i^p - \sigma(w_i p + b_i))\sigma'(w_i p + b_i)p_j \\
&= -(t_i^p - y_i^p)\sigma'(a_i^p)p_j
\end{aligned}$$

Here w_k defined a row of the matrix W, a_i^p is the i-th row of a^p, b_k is the k-th element of b and i corresponds to neuron i in the output layer.

We can now define the derivative $\Delta w_{ij} = -(t_i^p - y_i^p)\sigma'(a_i^p)p_j$. We use Δw_{ij} to update the value of weight w_{ij} using gradient descent.

We now consider what happens to the weights of neurons in a network with more than two layers.
For a neuron k in the output layer we define the factor $\delta_k = (t_k^p - y_k^p)\sigma'(a_k^p)$. This factor is used in the derivative of the weights we explained above.
For a neuron k' in a hidden layer we define a different factor. We first define $I_{k'}$ as all the neurons that use the output of neuron k' as their input. We now define $\delta_{k'} = \sigma'(a_{k'}) \sum_{i \in I_{k'}} \delta_i w'_{ik'}$. Here $w'_{ik'}$ corresponds to the weights used in the next layer.

An intuition one might have for backpropagation is that we are trying to do gradient descent using the derivative of the error with respect to the entire network. However, instead of computing with respect to entirety of the network, we compute the derivative of each layer separately. Note that we multiply with the derivatives when we compute the δ term. This multiplication can be understood as if we are using the chain rule to compute the derivative of this layer, with respect to the error E^p.

We summarise the derivative Δw_{ij} for a weight w_{ij}, a neuron i and a training pattern (y^p, t^p). We have $\Delta w_{ij} = -\delta_i p_j$.
If i is in the output layer we have $\delta_i = \sigma'(a_i^p)(t_i^p - y_i^p)$.
If i is not in the output layer we have $\delta_i = \sigma'(a_i) \sum_{j \in I_i} \delta_j w'_{ji}$.

We can now define the derivative Δw_{ij} for any weight at a neuron i. After which we use this derivative to update the weights using gradient descent. In backpropagation we update the weights of the neurons in the output layer first. This is because we need the δ term when we update a neuron in a hidden layer. One can see that an error term δ is propagating backwards through the neurons. We update all the weights using a so-called "backward pass". This comes down to iteratively going back through the layers to update the values of the weights.
A "forward pass" means to use the network as usual, and compute an output vector y_p given an input vector p.