DEGREE IN INDUSTRIAL TECHNOLOGIES

BACHELOR'S FINAL PROJECT

POWER SYSTEM SECURITY ANALYSIS USING REINFORCEMENT LEARNING

Author: José María Sunyer Nestares

Director: Richard Y. Zhang

Co-director: Andrés Ramos Galán

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Power System Security Analysis Using Reinforcement Learning

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/20 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.:  Fecha: 20/07/2020

Autorizada la entrega del proyecto
EL DIRECTOR DEL PROYECTO

Fdo.:  Fecha: 20/07/2020

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

DEGREE IN INDUSTRIAL TECHNOLOGIES

BACHELOR'S FINAL PROJECT

POWER SYSTEM SECURITY ANALYSIS USING REINFORCEMENT LEARNING

Author: José María Sunyer Nestares

Director: Richard Y. Zhang

Co-director: Andrés Ramos Galán

Madrid

Acknowledgments

I would like to express my gratitude to Professor Richard Y. Zhang for giving me the opportunity to work on this fascinating project which has allowed me to learn about the promising possibilities of combining power system security and Artificial Intelligence. Without his original idea and his guidance this work would not have been possible.

Also, I would like to thank the University of Illinois for allowing me to register to Artificial Intelligence and Machine Learning. These subjects have been crucial for the success and completion of this project.

ANÁLISIS DE SEGURIDAD EN LOS SISTEMAS DE POTENCIA APLICANDO APRENDIZAJE REFORZADO

Autor: Sunyer Nestares, José María.
Director: Y. Zhang, Richard.
Codirector: Ramos Galán, Andrés
Entidad Colaboradora: University of Illinois at Urbana Champaign

RESUMEN DEL PROYECTO

Actualmente, los sistemas de potencia utilizan como medida de seguridad, en parte, el criterio N-1 en el cual se requiere que el sistema permanezca operacional tras la caída de cualquier elemento individual. Esto no evalúa el riesgo de fallo en cascada, el cual es mas probable que ocurra con la integración distribuida a gran escala de elementos pequeños como generadores de energía renovable o vehículos eléctricos.

En este proyecto se presenta una herramienta de consulta basada en técnicas de inteligencia artificial capaz de verificar el criterio N-k en líneas de transmisión de un sistema de potencia. Se comprobará la seguridad del sistema tras la desconexión de k de un total de N líneas. Esta herramienta está diseñada para detectar posibles fallos en cascada, en donde el fallo de una línea resulta en la sobrecarga de otra ocasionando por tanto una secuencia de desconexiones en cadena. La aportación de este proyecto es formular este problema como un ejemplo de *el problema del camino más corto*, un problema clásico en programación dinámica y aprendizaje reforzado con una serie de soluciones estándar.

Palabras clave: seguridad de sistemas de potencia, fallo en cascada, aprendizaje reforzado, programación dinámica, criterio *N-k*

1. Introducción

El controlar y asegurar la seguridad de los sistemas eléctricos es un reto que evoluciona constantemente. La penetración de sistemas distribuidos de energía, el aumento de la energía renovable y la electrificación del transporte presentan nuevas oportunidades para desarrollar herramientas novedosas para mejorar la seguridad de los sistemas de potencia.

En los últimos años la inteligencia artificial ha visto numerosos éxitos en una gran variedad de industrias. Este proyecto está inspirado concretamente en las victorias en 1997 del superordenador diseñado por IBM contra el campeón mundial de ajedrez Garry Kasparov y en 2011 de AlphaGo al juego Go contra el número uno del momento.

En este proyecto se defiende que la maduración y el éxito de la inteligencia artificial presenta una oportunidad perfecta para enfrentar los retos futuros que afectarán a la seguridad de los sistemas de potencia.

2. Definición del Proyecto

Este proyecto desarrolla un índice capaz de evaluar la seguridad de un sistema de potencia y de cada una de sus líneas de transmisión. Formulamos el problema como *el problema del camino más corto*, en el que tratamos de encontrar la sucesión optima de líneas a desconectar que causen el máximo número de líneas sobrecargadas en un sistema sin variación de carga.

De la misma forma que los algoritmos de inteligencia artificial que derrotaron al hombre al ajedrez y al juego Go, hacemos uso de aprendizaje reforzado para crear una herramienta capaz de evaluar y detectar las líneas mas peligrosas.

3. Descripción del modelo/sistema/herramienta

De esta manera, definimos el estado de un sistema eléctrico con N líneas mediante tres elementos ($s = (\mathcal{D}, \mathcal{C}, \mathcal{T})$): el conjunto de las líneas desconectadas ($\mathcal{D}$), el conjunto de líneas sobrecargadas por efecto cascada ($\mathcal{C}$) y una variable binaria que indica si es posible continuar el análisis desconectado líneas ($\mathcal{T}$).

Como vemos en la Figura I, este proyecto formula el problema en forma de un proceso de decisión de Markov donde la recompensa es el número de líneas sobrecargadas cada vez que se desconecta una línea. Emplea las funciones valor de los estados y de las acciones para evaluar el índice de riesgo de cada sistema y de cada línea respectivamente. Estos han sido calculados a través de dos programas en Python: uno de programación dinámica y otro de aprendizaje reforzado (Monte Carlo).

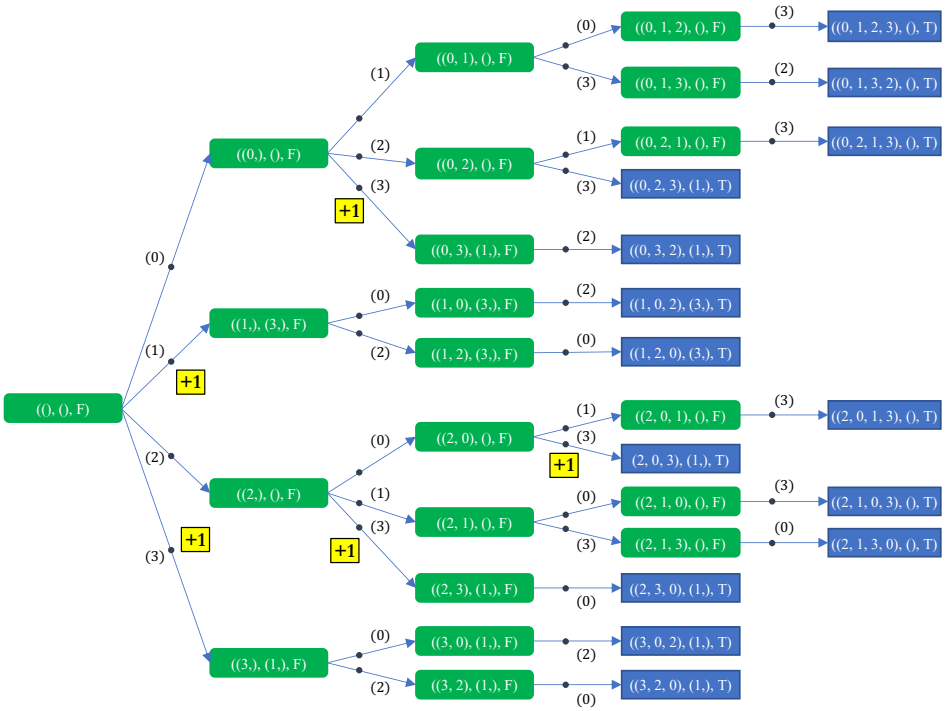

Figura I: Diagrama del proceso de decisión de Markov para el sistema Case4gs

Acción	$Q^*(s_0, a)$	$q_{\epsilon,500}(s_0, a)$
(0)	0.9	0.55
(1)	1	1
(2)	0.9	0.54
(3)	1	1

Tabla I: Evaluación de riesgo de cada contingencia en el estado inicial s_0 con $\gamma = 0.9$ usando programación dinámica ($Q^*(s_0, a)$) y Monte Carlo ($q_{\epsilon,500}(s_0, a)$)

4. Resultados

Las asunciones y la formulación han sido validadas en sistemas de prueba IEEE de pequeña escala (4 y 9 nudos) empleando una búsqueda exhaustiva y aprendizaje profundo. En la Tabla I podemos ver los resultados obtenidos para cada línea del sistema de 4 nudos con cada método. Mientras que para el sistema de 4 nudos la búsqueda exhaustiva ha funcionado en apenas unos segundos, para el sistema de 9 nudos el programa ha tardado unas 9 horas en dar el resultado.

Finalmente, hemos comprobado que la búsqueda exhaustiva no es una opción viable para sistemas IEEE de mediana escala (39 y 89 nudos) y que la técnica de Monte-Carlo es la única opción que ofrece un resultado para los índices de riesgo.

5. Conclusiones

La programación dinámica claramente no puede escalar a los sistemas reales y por tanto la investigación de otras alternativas es necesaria. En conclusión, el aprendizaje profundo tiene el potencial de convertirse en una herramienta indispensable por tres motivos:

1. Tiene la capacidad de aprender con la experiencia y de mejorar sus estimaciones en el tiempo y por tanto es capaz de dar las mejores estimaciones posibles en cualquier espacio temporal.
2. Puede utilizar información de otros análisis para mejorar sus estimaciones y por tanto es compatible con otros análisis de seguridad.
3. La capacidad de ajustar diferentes parámetros del modelo abre la puerta a un gran abanico de posibilidades para personalizar la herramienta en función de las necesidades de cada sistema de potencia.

6. Referencias

[1] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Adrian Kelly, Aidan O'Sullivan, Patrick de Mars, and Antoine Marot. Reinforcement learning for electricity network operation. *arXiv preprint arXiv:2003.07339*, 2020.

POWER SYSTEM SECURITY ANALYSIS USING REINFORCEMENT LEARNING

Author: Sunyer Nestares, José María.
Director: Y. Zhang, Richard.
Codirector: Ramos Galán, Andrés
Colaborating Entity: University of Illinois at Urbana Champaign

ABCTRACT

Existing power systems are made secure, in part, using the N-1 criterion, in which the system is required to remain within operational limits with the loss of any individual component. This does not assess the risk of cascading failure, which is likely to become more commonplace with the large-scale, distributed integration of small, stochastic components, such as renewable generators or electric vehicles.

In this thesis, we describe an AI-based advisory tool to verify the N−k criterion over transmission lines, meaning that the system is required to be secure with the disconnection of k out of N total transmission lines. The tool is designed to identify cascading mechanisms, in which the disconnection of one line overloads another, thereby resulting in a sequence of disconnections downstream. Our key insight is to formulate this cascading problem as an instance of the shortest path, a classic problem in dynamic programming and reinforcement learning with a number of standard solutions.

Keywords: power system security, cascading failure, reinforcement learning, dynamic programming, security analysis, N-k

1. Introduction

The challenge of controlling and guaranteeing the security of the power system is constantly evolving, particularly in light of significant predicted growth in the deployment of renewable energy, the penetration of distributed energy systems and increasing use of electric vehicles. These new challenges present new opportunities for developing new tools and technologies to improve the security of the power system.

Artificial intelligence (AI) has seen tremendous success in recent years in a wide range of sectors. This project is inspired by the computer designed by IBM which in 1997 beat the reigning world champion, Garry Kasparov, and by the Google project AlphaGo which beat the number one player at the game of Go.

The maturing of AI technologies presents a perfect opportunity to address the new challenges we have presented for power systems.

2. Project definition

This project develops and risk index which evaluates the security of both a power system and of each of its transmission lines. The problem is formulated as an instance of *the shortest path*, where we seek to find the optimal succession of lines to disconnect from a system which will cause the maximum number of overloaded lines in a system withour power flow variation.

The same way the AI algorithms which defeated human at both chess and Go, in this project we apply reinforcement learning to create an advisory tool able to assess and generate situations self-awareness about the vulnerable lines within a system.

3. Model description

In this project we will define the state of a system with N lines with three elements ($s = (\mathcal{D}, \mathcal{C}, \mathcal{T})$): the disconnected lines ($\mathcal{D}$), the cascaded lines ($\mathcal{C}$) and the terminal state indicator ($\mathcal{T}$) which indicates whether is possible or not to continue with the analysis. Our model assumes a constant power demand profile in order to focus our attention on the reinforcement learning aspect of the problem.

As we show in Figure I, the tool poses the problem of assessing the risk of cascading failure as a Markov Decision Process where we model an agent disconnecting lines from a power system and obtaining rewards in the form of the number of cascaded lines after each line disconnection. We use the state value function to assess the risk of cascade for a particular state of the system and the action value functions to assess the risk of cascading failure associated with each line contingency. Using Python, we have developed a program using dynamic programming and another one using reinforcement learning (Monte Carlo).

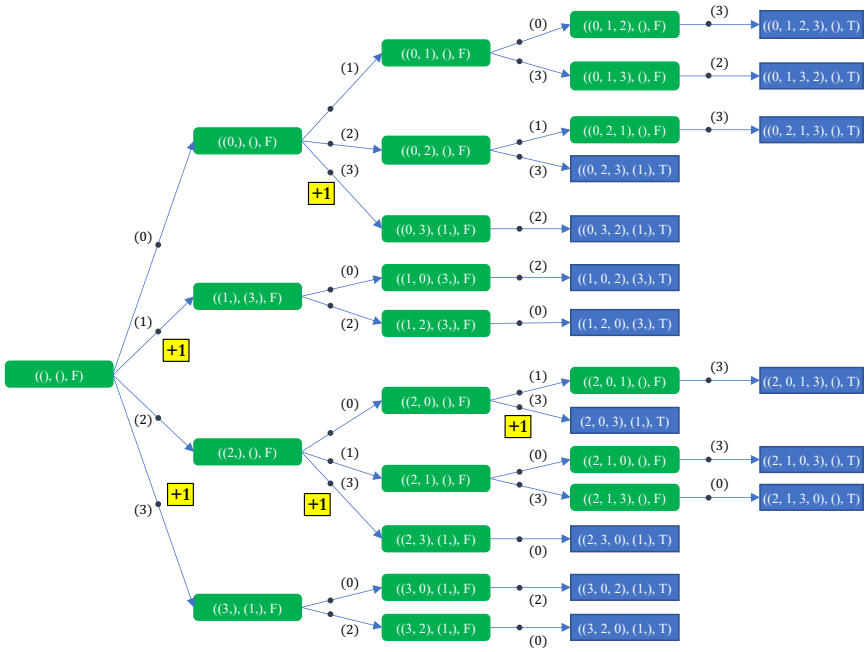

Figure I: Markov Decision Process diagram for Case4gs

Action	$Q^*(s_0, a)$	$q_{\epsilon,500}(s_0, a)$
(0)	0.9	0.55
(1)	1	1
(2)	0.9	0.54
(3)	1	1

Table I: Risk assessment of each line contingency at the initial state s_0 using Dynamic Programming and the Monte-Carlo method with 500 episodes for Case4gs with$\gamma = 0.9$ and $\epsilon = 0.2$

4. Results

The assumptions of the formulation are validated on small-scale IEEE test cases (4-bus and 9-bus systems) using exhaustive search and reinforcement learning. In Table I we can see the results obtained for each line of the system of 4 nodes with each method. While exhaustive search works perfectly in just seconds for the 4-bus system, it takes about 9 hours to get find the solution in the 9-bus system.

Because the dynamic programming does not scale well to medium size systems, we have just been able to test a 39-bus and a 89-bus system using Monte-Carlo which is the key insight of this project.

5. Conclusion

Clearly, dynamic programming cannot scale to real world power systems, so its necessary to investigate alternative approaches. In conclusion, reinforcement learning has the potential to be an invaluable advisory tool for grid operators for three main reasons:

1. It learns with experience, improves its estimations over time and, therefore, can generate information in real-time (on-demand).
2. It can us information from other analyses to enhance its estimations, and therefore stands to benefit from other power security and contingency analysis already in existence.
3. It can adjust several parameters of the model allowing customization of this technique according to the needs of each individual power system.

6. References

[1] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Adrian Kelly, Aidan O'Sullivan, Patrick de Mars, and Antoine Marot. Reinforcement learning for electricity network operation. *arXiv preprint arXiv:2003.07339*, 2020.

Index

List of figures

List of tables

Part I. MOTIVATION AND INTRODUCTION

The challenge of controlling and guaranteeing the security of the power system is constantly evolving. With the growth of renewable energy in the grid, the intensive penetration of distributed energy resources and changes in demand characteristics, new challenges for power grid operators arise. According to the International Energy Agency (IEA), low-carbon sources are expected to provide more than half of the total electricity throughout the world by 2040, with wind and solar PV becoming the main renewable energy sources (see Figure 1). Another key driver in the future transformation of the power systems is the electrification of the transportation sector. As shown in Figure 2, also taken from the IEA, the global stock of electric vehicles has increased by a factor of 13 between 2013 and 2018 and is expected to keep growing in the next years[1]. These present new challenges to the grid, in the form of increasing uncertainty and variability. At the same time, they present new opportunities for developing new tools and technologies to improve the security of the power system.

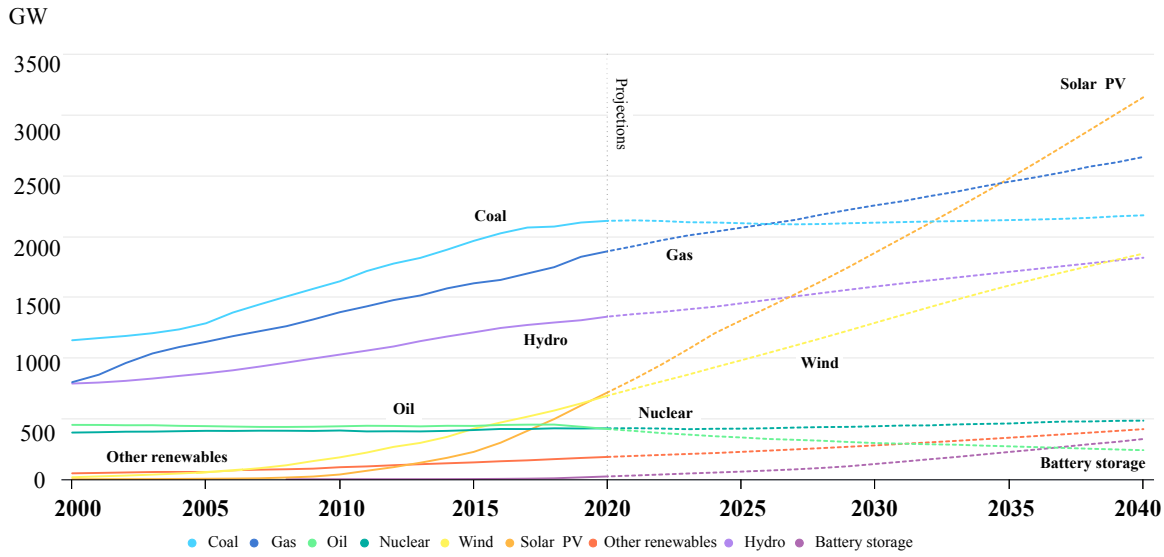

Figure 1: World installed power generation capacity by source between 2000 and 2040 from the International Energy Agency

[1] In Figure 1, BEV stands for Battery Electric Vehicle while PHEV stands for Plug-in Hybrid Electric Vehicle

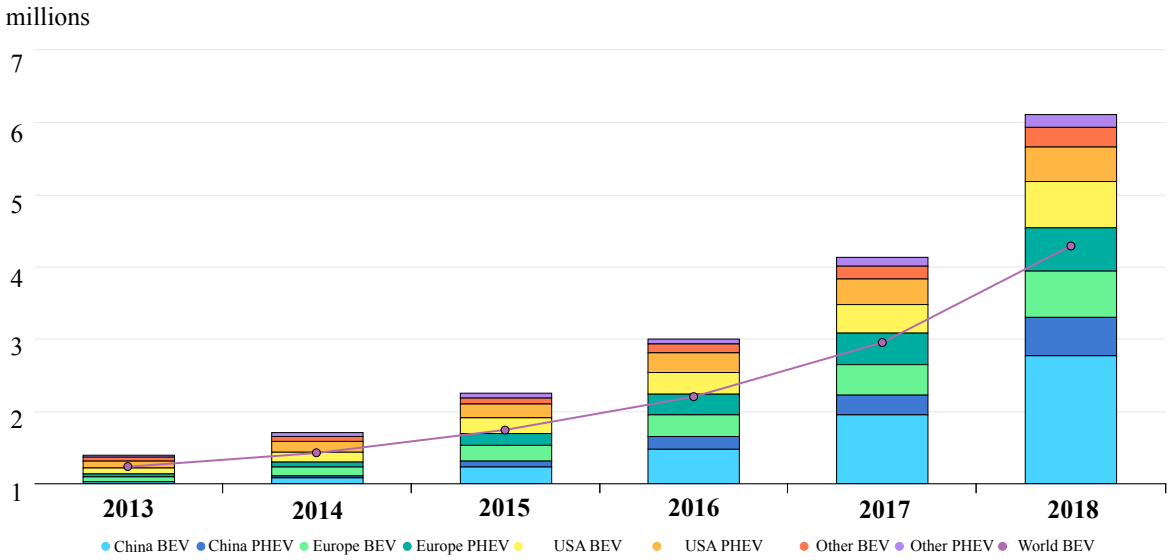

Figure 2: Electric car deployment in indicated countries 2013-2018

Artificial intelligence (AI) has seen tremendous success in recent years, over applications as wide-ranging as software, healthcare, autonomous vehicles, natural language processing, finance, marketing, and within many more industrial sectors. One of the most famous applications of AI took place in 1997, when Deep Blue, a chess-playing computer developed by IBM, defeated for the first time in history, the reigning world chess champion at the time, Garry Kasparov [16]. AI again captured the imagination of the public in 2011, when the supercomputer "Watson" [18], a question-answering computer system capable of answering questions in natural language, defeated two world champions in "Jeopardy", a popular and well-known TV show in which contestants prove their intelligence by answering different quiz questions. If beating the world's best player in chess was not enough, in 2017, AlphaGo, a Google DeepMind project, beat Ke Jie, who at the time was ranked number one in the world. The most impressive aspect of this is that, if chess has around 10^{120} possible chessboard configurations [25], Go presents a shocking estimated number of board positions of around 10^{172}.

The maturing of AI technologies presents a perfect opportunity to address the new challenges we have presented for power systems. As we have seen, AI has offered solutions to very large and complex problems in short computational time and has proven to be able to learn from experience, eventually developing an intuition that is comparable, or in some cases even exceeding, that of human experts. Through this project, we aim to create a tool to help operators analyze a perennial security threat facing power systems: cascading failures. We will develop a specific AI framework called reinforcement learning (RL) which aims to mimic the intuition developed by grid operators from experience and use it to generate information in real-time (on-demand) and assess the risk of cascading failures.

In order to understand how this AI tool could help grid operators in guaranteeing a safe power system operation, we will explain how grid operators work in Section 1 and how a network operates safely in Section 2.

1 HOW GRID OPERATORS WORK

The control room (see Figure 3) is the central nervous system of the electric grid. Here is where operators make decisions round the clock that are critical for maintaining the electric reliability of the system and ensuring the balance of supply and demand power. Each of these operators is structured within different "desks" which have a different role and responsibility with the power system network [14]:

- **The real-time desk** monitors and maintains a frequency of 60 or 50 Hz depending on the country and ensures generation resources are fulfilling their obligations.

- **The transmission and security desk** makes sure the transmission system is operating safely and reliably by monitoring real-time flows on transmission lines and taking action when congestions, outages or voltages issues occur. The AI framework we aim to build in this project is mainly based on their work.

- **The resource operations desk** monitors the performance of generation resources and makes sure that there are sufficient operating reserves available.

- **The reliability unit commitment desk** ensures sufficient generation capacity is committed to reliably serve the forecasted demand.

- **The interconnection desk** is responsible for energy transactions into and out of the grid between interconnected grids.

- **The supervisor desk** monitors and directs actions for all control center desk activities.

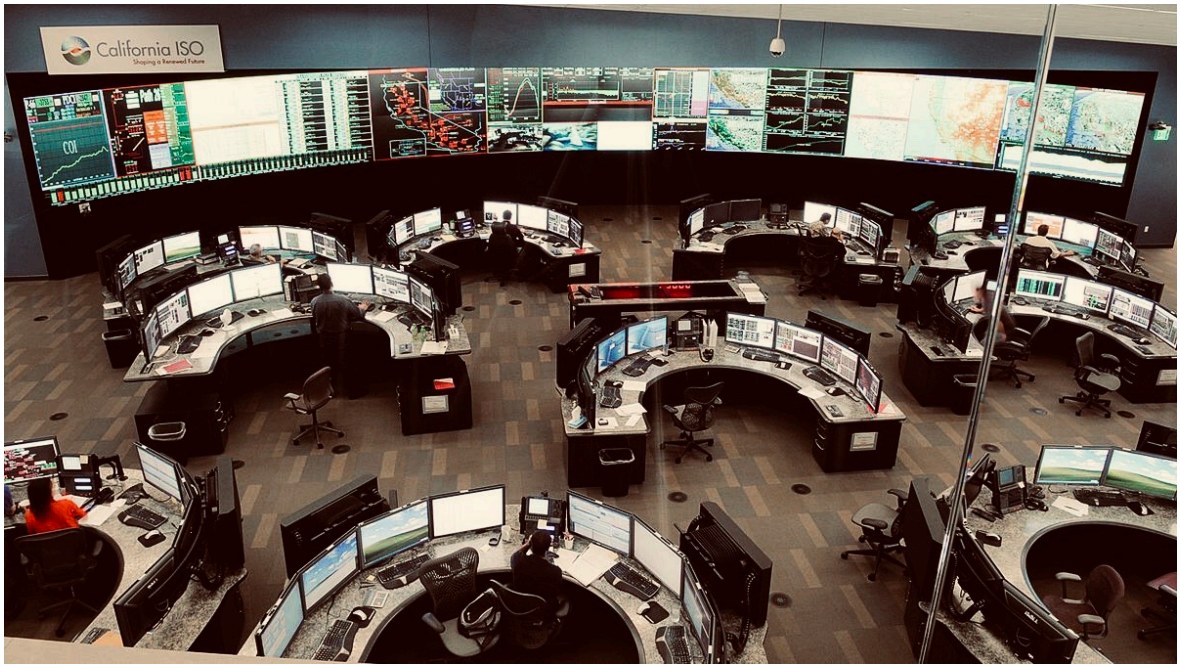

Figure 3: The control room at the California Independent System Operator headquarters in Folsom

Figure 3 shows the control room in the California Independent System Operator. As we can appreciate, grid operators work with a set of computer consoles within a control center with an enormous amount of data and information that allows them to make decisions that are critical to maintaining electric reliability. Here is where our tool could play an important role.

Timely operator action in response to a particular event is the difference between having a major event or it turning into a non-event and so it is important for the operators to have the authority to make decisions quickly without having to check with management. As we can see in Figure 4 in the Electric Reliability Council of Texas they have a big poster displayed on the wall in the control room signed by everybody from the CEO on down that gives them the authority to make whatever decisions they need to make without approval [12]. As a result, if we want to develop a tool able to help these operators, we can see it would be very powerful if it was able to give solutions in real-time. RL is unusual in that it generates data on-demand. This characteristic is one of the key reasons we believe it could become an invaluable tool for grid operators.

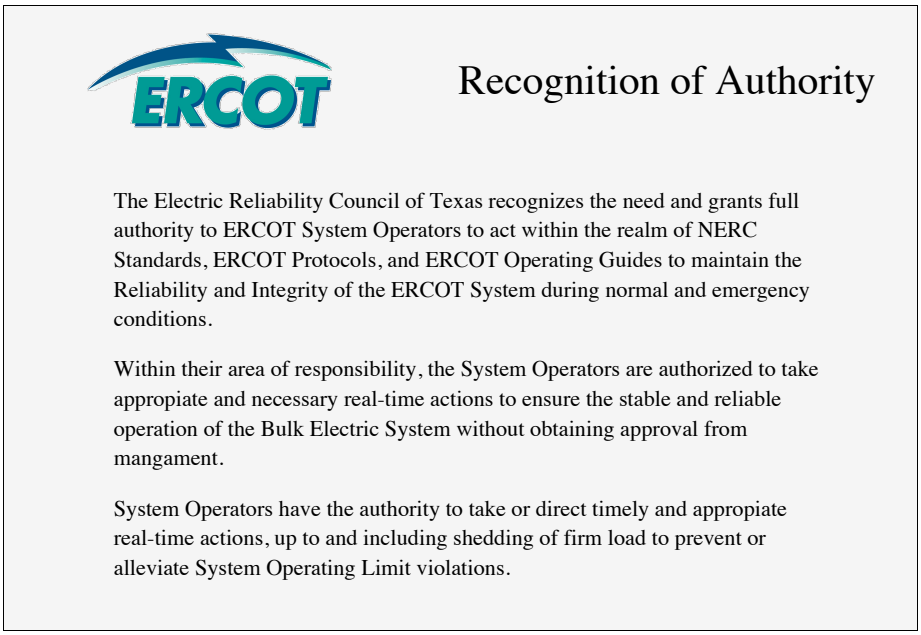

Figure 4: Recognition of Authority for System Operators in the Electric Reliability Council of Texas

Power system operators are on the front line ensuring reliable delivery of electricity to consumers, businesses and industries. With so much at stake, there is no question that a system operator's job is stressful, especially during major system disturbances. Therefore, when not at their desk, system operators are constantly training, improving their skills and using simulation to practice new situations that broaden their expertise. Dispatcher training simulators, also known as operator training simulators, allow grid operators to learn how to properly control an electric grid when they are not working. The presence of simulators in power systems is another key feature that makes RL the perfect tool for security analysis. Through simulation, RL can learn the same way operators do and develop their same expertise and intuition or even improve it.

2 HOW THE NETWORK OPERATES

Even though we have classified six different types of desks within the control room, each one of them will perform its function properly if the rest of them do as well. The tool we are going to develop is mainly focused on the work done by the transmission and security operators. Their goal is to ensure the system operates safely and reliably, but it will only be possible with the coordination and good work of all the desks. They all operate the network by ensuring that the three primary constraints of the electric grid are met at all times in every area of the network [19]:

2.1 THERMAL LIMITS OF TRANSMISSION EQUIPMENT

Power flows through the network from generators to demand nodes through transmission lines. These transmission lines are made of metallic materials, namely, copper or aluminum and therefore they have cooling and heating limits. The power through the lines dissipates energy in the form of heat according to Ohm's law:

$$P_{loss} = IR^2 \tag{1}$$

This power combined with solar radiation and a high ambient temperature can overheat the line and cause it to sag closer to the ground (see Figure 5). The risk becomes greater as the length of the lines increases. When lines start to sag, they can end up touching a tree or the ground causing a short circuit. Due to the voltage difference, a circulation of large currents would originate from the supply sources and also from the loads. To prevent this, grid operators set different power or current limits to every line as we can see in equations (2) and (3) where n is the total number of nodes in a power system:

$$\underline{I}_{i,j} < I_{i,j} < \overline{I}_{i,j} \qquad \forall\, i,j \in \{1,\dots,n\} \backslash i = j \tag{2}$$

$$\underline{P}_{i,j} < P_{i,j} < \overline{P}_{i,j} \qquad \forall\, i,j \in \{1,\dots,n\} \backslash i = j \tag{3}$$

Grid operators install protection mechanisms to automatically disconnect the lines which exceed these limits with a certain margin to prevent potentially dangerous and costly system problems. This causes the power flows to find new routes between the supply and the demand. When this happens, other lines can increase their load and can eventually trigger more overloaded lines that need to be disconnected and so on. This is known as the cascading failure and results in the loss of power supply to the end-user, which is known as a blackout. Examples of these blackouts are spread throughout the history of power systems. The most recent large-scale blackout happened on 16 June 2019 and struck Argentina, Uruguay, and Paraguay, leaving an estimated total of 48 million people without electrical supply [15].

The objective of the RL model in this thesis is to anticipate potentially vulnerable lines that can cause cascading failures. Situational awareness about vulnerable lines within the system could help operators to quickly respond to threats and restore the power grid to safe operating conditions. In the model, due to the significant importance of overloading with respect to underloading from equations (2) and (3), we will just consider the upper-bounds $\overline{P}_{ij}$ and $\overline{I}_{ij}$ and we will ignore $\underline{P}_{ij}$ and $\underline{I}_{ij}$.

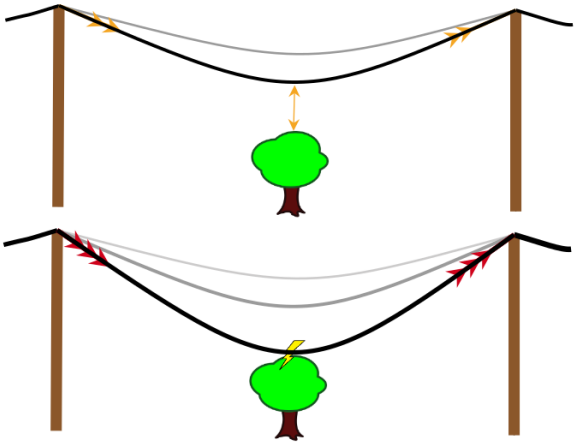

Figure 5: An example of the dangers of overheating power lines by transporting too much current.

2.2 VOLTAGE MAINTENANCE WITHIN A SAFE RANGE

The key challenge of voltage levels is to step them up or down to optimize power flow while maintaining them close to the nominal value. Voltages out of range have caused systems to collapse and eventually have caused blackouts as well. They are mainly controlled by adjusting reactive power: a correct distribution of reactive power between generators that generate it, loads that absorb it and transmission elements that can either generate or consume it are essential for guaranteeing a safe voltage profile in a network. A system-wide voltage collapse, known as a *brownout*, may result from a disruption in an electrical grid, or may occasionally be imposed to reduce load and prevent blackouts. Usual limits are between 110% and 90% of the nominal voltage of each of the elements in the grid. In a grid with a total of n nodes, all the i-nodes must satisfy equation (4) to ensure safe voltage operation:

$$\underline{V_i} < V_i < \overline{V}_i \qquad \forall\, i \in \{1, \dots, n\} \tag{4}$$

Despite its importance for ensuring the reliability of a power system, in our RL framework, we will not consider the voltage limits due to the additional complexity that could introduce into the problem. Nevertheless, the model we will present can easily be modified and adjusted to also satisfy the voltage requirements. Voltage control could be introduced in future versions of this project.

2.3 GENERATION, SUPPLY AND FREQUENCY BALANCE

The third constraint to consider is the power balance between supply and demand. For this purpose, frequency is typically used as a measure of how the power is balanced. Without going into too much detail, when the supply exceeds demand, the frequency increases and when the opposite happens (the demand is larger than supply) it decreases. If the frequency, f, deviates too much from its nominal value (approximately 2 Hz) it will likely collapse the entire grid. To avoid variations in frequency, the grid must supply enough power for compensating both the real power demand and the losses generated according to equation (1). If we assume a network is made up of a total of n nodes where the first m of them are generators we can present equation (5):

$$\sum_{i=1}^{m} P_{gen} = \sum_{i=m+1}^{n} P_{load} + P_{losses} \Leftrightarrow \frac{df}{dt} = 0 \tag{5}$$

Here the real-time desk operators presented in Section 1 play a key role in ensuring that power balance at all times. In order to maintain it, they dispatch the generation sources accordingly to meet the demand by end-users. In our RL framework, we will ensure power balance at all times using a *slack bus*, the details of which we will present in Section 4.2.

Part II. THEORETICAL FRAMEWORK

As we have already stated, we will use a RL framework to create a tool to provide awareness about potential cascading failures in a power system. In order to understand the theory underlying both RL and power systems, we will explain it in detail in Sections 3 and 4 respectively. Our review of RL is mainly based the on the book Reinforcement Learning: An Introduction from Richard S. Sutton and Andrew G. Barto on Chapters 4, 5 and 6.

3 REINFORCEMENT LEARNING

3.1 RL WITHIN ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Artificial Intelligence (AI) is a field within computer science that can be defined as the intelligence demonstrated by machines in contrast to the natural intelligence displayed by humans and animals. Intuitively, it can be understood as the study of "intelligent agents", meaning any device that perceives its environment and takes actions that maximize its chances of successfully achieving its goals [21]. AI is often used to describe machines or computers that mimic functions that we associate with the human mind, such as "learning" and "problem-solving".

Machine Learning (ML) is a set of statistical techniques widely used in AI based around the idea that we should just be able to give machines access to data and let them learn for themselves [24]. The main goal of ML is to develop techniques to allow machines to learn from experience. In order to give some general background to the reader about ML we will shortly present the three different learning techniques used within the field:

- **Supervised learning**: learns a mathematical model for a given dataset to relate input information of data represented as $x^{(i)}$ with output information, also known as labeled data, represented as $y^{(i)}$. A training data with the form shown in (6) is given to training an inferred function used for making predictions about output values. Linear regression, logistic regression, decision trees, neural networks, k-nearest neighbor and support vector machines are examples of this type. Within this project, neural networks will appear as an alternative approach for our RL model.

$$\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N} \tag{6}$$

- **Unsupervised learning**: is used to group unlabeled data based on similar attributes, naturally occurring trends, patterns, or relationships. The system explores the data in the form of (7) and draws inferences to describe hidden structures. The most used techniques are principal component analysis (PCA) and cluster analysis like *k*-means, hierarchical clustering or gaussian mixture models (GMM). This learning approach will not appear within this project and is just presented for setting the background of reinforcement learning.

$$\mathcal{D} = \{\left(x^{(i)}\right)\}_{i=1}^{N} \tag{7}$$

- **Reinforcement learning (RL)**: trains machine models to make a sequence of decisions in order to achieve the desired goal. It is concerned with how agents ought to take actions in an environment to maximize the notion of cumulative reward. An RL agent learns by interacting with its environment, it receives rewards by performing correctly and penalties for performing incorrectly. The agent learns without intervention from a human by maximizing its reward and minimizing its penalties [19]. RL applied to power systems is the main contribution of this project and so it will be explained with more detail in Section 3.2. In comparison with supervised learning and unsupervised learning we will see that RL data is often represented as a Markov Decision Process where data is structured as a tuple of states, actions, transitional probabilities and rewards:

$$\mathcal{D} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}) \tag{8}$$

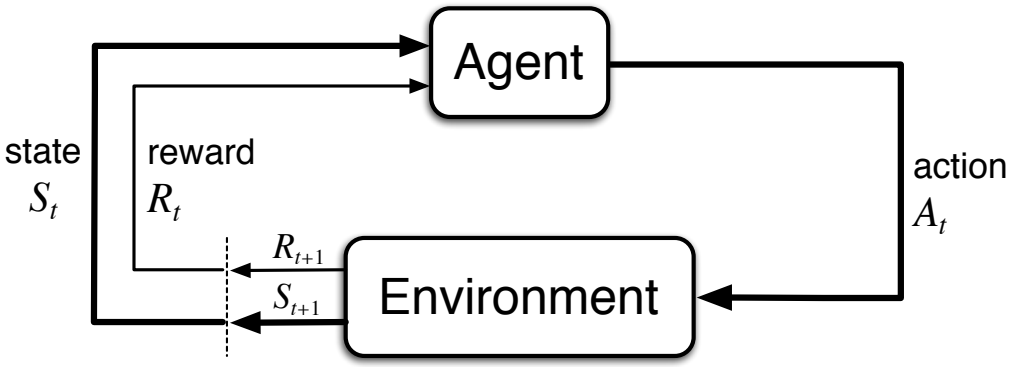

Figure 6: The agent-environment interaction in RL

3.2 EXAMPLES OF RL

As we have stated, RL is structured as an agent that takes actions in a certain environment. This agent decides the actions it takes based on his experience and with the goal of maximizing the sum of the rewards he expects to get in the future. Before explaining in detail what are the elements that make up an RL model let's get some examples of possible applications:

- **Playing chess**: a chess player can be modeled as an agent who decides which movements to make in the chessboard in each turn. The goal of this agent is to checkmate the opponent's king and to do so the agent needs to know which actions will most likely take him to his victory, which check boards are the most valuable to get to and which ones should better be avoided.

- **Autonomous cars**: car drivers can also be modeled as RL agents that decide which roads to take or avoid moving from one place to another within the shortest possible time. Based on what the agent sees driving, he decides whether to use one path or the other to get to his final destination on time. For example, this agent should learn to avoid traffic jams, closed roads, long highways and red traffic signals and should learn to use empty roads and shortcuts to minimize the traveling time.

- **Portfolio management**: professional investors can be modeled as agents which continuously make decisions on reallocating its resources into a number of different financial investment assets to maximize its profits [28].

- **Play Atari games**: where the agent is modeled as a player that needs to know which are the most convenient moves to get to the goal state, winning the game. For example, in Pacman, the agent's goal is to eat all the dots placed in the maze while avoiding colored ghosts. The agent will move from one scenario to another as he plays the game and, if he is a proper learner, he should learn which directions are most convenient for him to take in each scenario in order to win the game [20].

What all these examples have in common is the structure we can see in Figure 6. We can see what each of the elements is in the shown structure for the given examples in Table 1. The agent is a software program that makes intelligent decisions with its surroundings which we refer to as its environment and they are the learners in RL. As we can appreciate, all the examples have in common an interaction between an agent and an environment. Within this interaction, these agents take different actions transitioning from one state in the environment to another and every time they take an action, they get some feedback from the environment, a reward. This reward is a number that determines which are the good and the bad events for the agent. The goal of an agent in RL is to get the maximum reward possible when interacting with the environment.

Example	Agent	Actions	Environment	States
Chess	chess player	chess move	chess game	chessboard configuration
Autonomous car	autonomous driver	choosing a route	road network	car location
Porfolio management	investor	investments and divestments	stock market	specific portfolio
Atari games (e.g. Pacman)	Atari player	which key to press in the controller	Atari game	screen snaphshot

Table 1: Elements of the agent-environment interaction for each example

In order to explain how RL can be applied to power systems, we will present the problem we aim to solve in this thesis in the form of a third-person attacker. We will use this problem to illustrate all the key concepts of RL.

3.3 ATTACK THREAT PROBLEM

Let's imagine we have an attacker that is trying to turn off a whole power system and, to achieve this goal, he disconnects lines from the system. Let's assume this adversary will not stop damaging the system until all the lines in a system are inoperative. So, using the RL nomenclature we have just presented we could model this attacker as the agent and the power grid as the environment where this agent interacts. The interaction he has with a power system by disconnecting lines would be the actions this agent can take. Every time this agent takes an action over the power system it transitions from one configuration to a new one. In this example, we could model the states as the different configurations that the power systems can have. We can assume, as we have stated in equations (2) and (3), that every time the power systems have overloaded lines, protection mechanisms automatically disconnect these lines as a security procedure. If we model the number of disconnected lines by these mechanisms as the reward signal, then we would have all the elements needed to build a RL framework. This framework is presented in Figure 7.

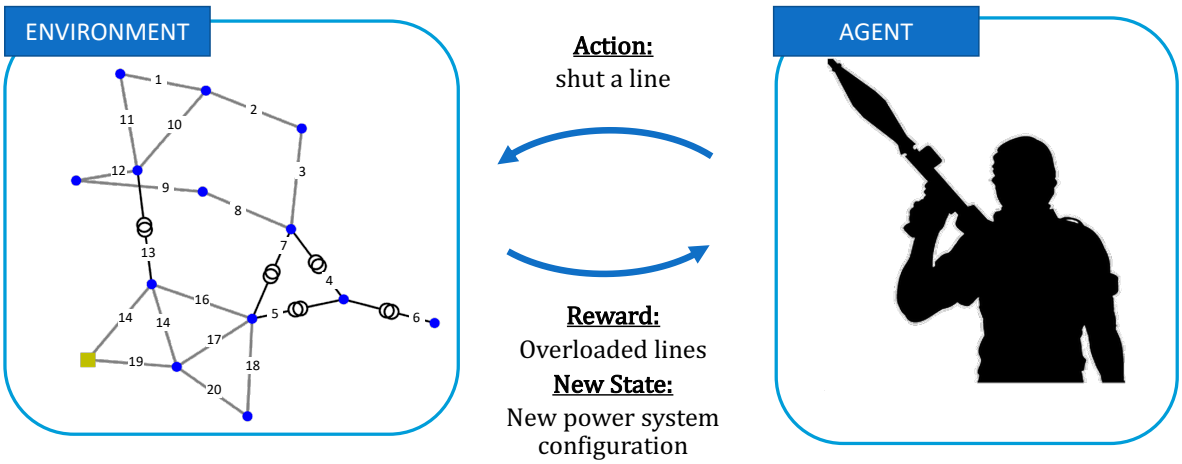

Figure 7: The proposed agent-environment interaction in RL applied to power systems

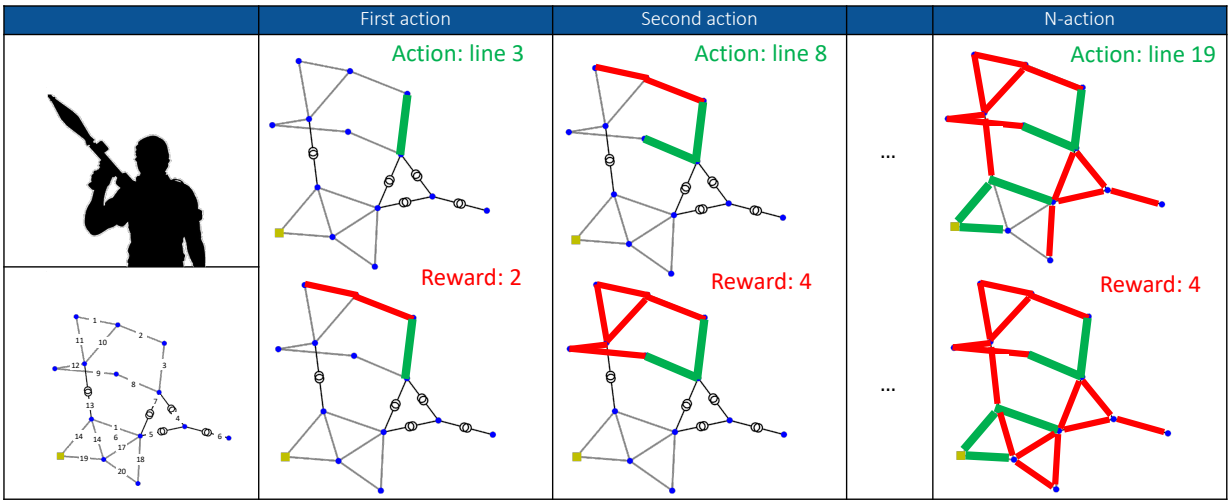

Figure 8: Complete episode in the attack threat problem

To better illustrate how this presented interaction would work, we could present an example as the one shown in Figure 8. Let's imagine this attacker initially decides to disconnect line 3 and after that, lines 1 and 2 overloaded and therefore are switched off. The system would have 3 inoperative lines and the node between lines 2 and 3 completely isolated from the grid. If immediately after this adversary decides to disconnect line 8, which was a very important line for the system, then lines 9, 10, 11 and 12 would be disconnected due to overloading. This agent would continue to disconnect lines until some point where all the lines would be already turned off and so he would have achieved his goal. We call this final state of the system a *terminal state* and we refer to this complete sequence of transitions between an initial step and a terminal state as an *episode*.

If this adversary was smart enough, he would be interested in finding the optimal combination of disconnected lines that guarantee him arriving at a terminal state as soon as possible. Using a decision tree, he could plot all the possible episodes he could imagine and analyze what are the most promising ones. In RL, we call this decision tree a Markov Decision Process and it will be the formal framework we will use in this project to define the interaction between the learning agent and its environment.

3.4 MARKOV DECISION PROCESS

A Markov Decision Process (MDP), is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision making of an agent in situations where outcomes are partly random and partly under the control of a decision-maker. MDPs are particularly useful for studying optimization problems solved via dynamic programming and RL [17]. As we have shown in equation (8), MDPs are defined by a tuple of states ($\mathcal{S}$), actions ($\mathcal{A}$), transitional probabilities ($\mathcal{P}$) and rewards ($\mathcal{R}$). Continuing with our attack threat problem and assuming the attacker has full knowledge of the MDP as shown in Figure 9, we will explain in detail the 4 elements that make up an MDP:

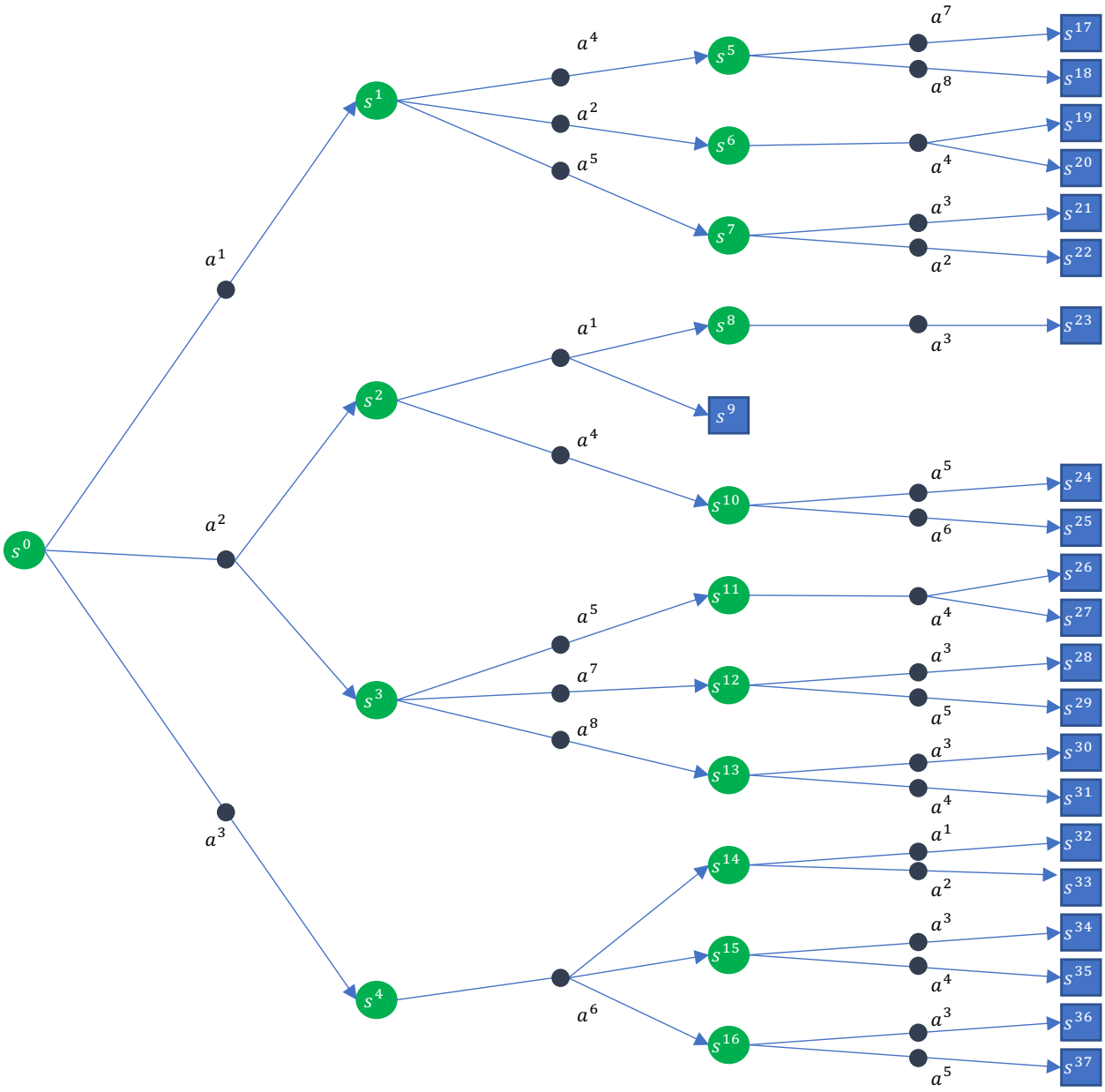

Figure 9: An example of a Markov Decision Process for the Attack threat problem on a small-size power system

- **States:**

The state, denoted by S when presenting it as a variable and by when we refer to it as the realization of S, describes the complete situation of the environment. In this project, there are different possible ways of describing a state and we will explain them elaborately in Part III, for now, we will move on considering that states are a complete characterization of the power system configuration. If we denote by t each of the discrete time-steps in an episode we will use S_t to define the state S at time t. Also, we will use $\mathcal{S}^+$ to refer to the set of all possible states, $\mathcal{S} \subset \mathcal{S}^+$ to refer to all the terminal and $|\mathcal{S}|$ and $|\mathcal{S}^+|$ to refer to the total number of states within each set. So in Figure 9, we would have:

$$\mathcal{S}^+ = \{s^{(i)}\}_{i=0}^{|\mathcal{S}^+|} \; : \; |\mathcal{S}^+| = 38$$

$$\mathcal{S} = \{s^{(i)}\}_{i=0}^{|\mathcal{S}|} \setminus s^{(9)} \; : \; |\mathcal{S}| = 16$$

- **Actions:**

Actions, denoted either as a variable A or as a realization of the variable as a, are deliberated operations executed by the agent. An action taken at time t is modeled as A_t and it defines the transition from state S_t to S_{t+1}. Through actions, the agent can interact with the environment and receive a reward. In the presented problem actions are defined as the operation of shutting down a line. We will denote by $\mathcal{A}$ to the set of all possible actions in an MDP and by $\mathcal{A}(s) \subset \mathcal{A}$ to the set of possible actions from $S_t = s^{(i)}$. Using the MDP in Figure 9:

$$\mathcal{A} = \{a^{(i)}\}_{i=1}^{|\mathcal{A}|} \; : \; |\mathcal{A}| = 8$$

$$\mathcal{A}\left(s^{(0)}\right) = \{a^{(1)}, a^{(2)}, a^{(3)}\},\ \mathcal{A}\left(s^{(1)}\right) = \{a^{(2)}, a^{(4)}, a^{(5)}\},\ \mathcal{A}\left(s^{(2)}\right) = \{a^{(1)}, a^{(4)}\},\ \ldots$$

To describe which actions a learning agent must take from a given state we define a policy. A policy is a mapping function from perceived states of the environment to actions to be taken when in those states. When we use deterministic policies we use the expression (9) which maps which action a to take when we are in state s and when a policy is stochastic we use the more general expression (10) where the output is the probability of taking action a given that we are in state s.

$$\pi(s) \rightarrow a \tag{9}$$

$$\pi(a|s) = \Pr\{A_t = a | S_t = s\} \tag{10}$$

- **Rewards:**

The reward signal, denoted as either R or r, the same way as states and actions, defines the goal of a RL problem. They are the number received by the agent on each time-step t from the environment. It defines what are the good and bad events for the agent in an immediate sense. The agent's objective is to maximize the total reward it receives over the long run. In our example, we defined the rewards as the number of overloaded lines that had to be disconnected automatically. In general, reward signals may be stochastic functions of the state of the environment and the action taken. Therefore we will denote by $r(s,a)$ to the expected immediate reward after taking action a from state s and by $r(s,a,s')$ to the expected immediate reward when transitioning from s to s' taking action a:

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{11}$$

$$r(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] \tag{12}$$

In the episode shown in Figure 8, referring to T as the final time-step before arriving at a terminal state and assuming the initial state is S_0 the reward signals R_t where $r_1 = 2,\ r_2 = 4,\ \ldots, r_T = 4$. In this project, because power systems have finite lines T will also always be finite.

- **Transition model:**

The transition model is the general framework that defines the connections between actions, states and rewards in an environment. In Figure 9, we can see states are modeled as either green circles if they are non-terminal or blue squares if they are terminal and actions modeled as dark circles. The probability of transitioning to a state s' and getting a reward r, given that we move from state s taking action a is what we call the transitional probability $p(s',r|s,a)$ as we can see in equation (13). If we denote by $\mathcal{R} \subset \mathbb{R}$ to the set of all possible rewards in a MDP we can marginalize (13) and get the probability of transitioning from state s taking action a to state s' as we see in equation (14):

$$p(s',r|s,a) = \Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{13}$$

$$p(s'|s,a) = \sum_{r\in\mathcal{R}} p(s',r|s,a) \tag{14}$$

We can formulate this expression due to the memoryless property of the transitioning between states which is known as the Markov property. It states that the conditional probability distribution of future states in a process depends only on the present state, not on the sequence of events that preceded it as we can see in the following equation:

$$\Pr\{S_t = s_t | S_{t-1} = s_{t-1}, \dots, S_0 = s_0\} = \Pr\{S_t = s_t | S_{t-1} = s_{t-1}\} \tag{15}$$

3.5 VALUE OF STATES AND ACTIONS

Once introduced all the essential concepts we will use within our MDP, we still have to figure out how to move from our initial state to a terminal state in an optimal way. In this section, we will show the way of assigning values to all the states and state-action pairs in RL so that the agent can know which are the most promising ones and ultimately to see how to compute the optimal path in our MDP.

Continuing with our example of attacking the grid we could realize that if the agent wants to break the whole system using the least number of actions, he needs to overload the maximum number of lines possible. So if from initial time $t = 0$ he gets a sequence of rewards: $r_{t+1}, r_{t+2}, \dots, r_T$ what he wants is to maximize the sum these rewards, which we will refer to as the **return**.

Nevertheless, imagine there are several possible policies he can take that lead to the same return, then it would make sense for him to take the actions that give the biggest rewards first just in case he has to stop taking actions and never gets to a terminal state. In other words, he would value more the immediate rewards in comparison to the more distant future rewards. In RL this is modeled with the **discount rate** $\gamma \in [0,1]$, which models how less valuable are future states in time-step $t + 1$ in comparison to the more present states in time-step t. Using this discount rate, we define the total discounted return of a given sequence of future rewards as:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= \sum_{k=0}^{T} \gamma^k R_{t+1+k} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{16}$$

As we can see in the last equality of (16), we can express the return using a recursive relationship between G_t and G_{t+1}. The expected value of the return is what we will use for defining the value of states and actions. Because the set of rewards obtained from time-step t to time-step T depends on the policy used, we will define the value of a state s with respect to a policy π as the expected return when starting from state s and following policy π as shown in equation (17).

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\,[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s']] \end{aligned} \tag{17}$$

As we can see in (17), in order to get the value of s given π we need to know the transition model $p(s', r|s, a)$, the discount rate γ and $\mathbb{E}_\pi[G_{t+1}|S_t = s']$ which, if we pay attention to (17), is nothing but $V_\pi(s')$. This recursive iteration is called the Bellman equation for V_π and it expresses the relationship between the value of a state and the value of its successor states:

$$\underbrace{V_\pi(s)}_{\text{Actual state value}} = \sum_a \underbrace{\pi(a|s)}_{\text{Policy}} \sum_{s',r} \underbrace{p(s',r|s,a)}_{\text{Dynamics}} [r + \gamma \underbrace{V_\pi(s')}_{\text{Succesive state value}}] \tag{18}$$

Similarly, using this intuition, we define the value of a state-action pair using a specific policy, as the expected return when starting at state s, taking action a and following π thereafter as shown in equation (19).

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= r(s,a) + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} \pi(a'|s')\, \mathbb{E}_\pi[G_{t+1}|S_t = s', A_t = a'] \end{aligned} \tag{19}$$

This can also be formulated using a recursive relationship as the Bellman equation for Q_π where we can see that actual state-action value function depends on the successive state-action pairs as shown in (20).

$$\underbrace{Q_\pi(s,a)}_{\text{Actual action value}} = \underbrace{r(s,a)}_{\text{Expected reward}} + \gamma \sum_{s'} \underbrace{p(s'|s,a)}_{\text{Dynamics}} \sum_{a'} \underbrace{\pi(a'|s')}_{\text{Policy}} \underbrace{Q_\pi(s',a')}_{\text{Succesive action value}} \tag{20}$$

Following our example, now that we have presented the value functions $V_\pi(s)$ and $Q_\pi(s,a)$ the agent would be able to see how promising are every state and every state-action for a fixed policy π. Nevertheless, he is not interested in this but in finding the optimal policy he needs to take in the MDP to get the maximum total expected return. If we denote by $V^*(s)$ and $Q^*(s,a)$ to the maximum expected discounted return for every state and state action pair, we can define them as:

$$\begin{aligned} V^*(s) &= \max_{\pi} V_\pi(s) \\ &= \max_{a} Q^*(s,a) \\ &= \max_{a} \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a} \sum_{s',r} p(s', r|s, a)[r + \gamma V^*(s')] \qquad \text{s.t. } a \in \mathcal{A}(s) \end{aligned} \tag{21}$$

$$\begin{aligned} Q^*(s,a) &= \max_{\pi} Q_\pi(s,a) \\ &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a)] \\ &= r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q^*(s', a') \qquad \text{s.t. } a \in \mathcal{A}(s) \end{aligned} \tag{22}$$

Once we have defined the optimal value functions, we know that the agent has to take the actions that lead him to the maximum expected discounted return and so we can compute the optimal policy, π^*, as follows:

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_a Q^*(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

If the attacker follows the optimal policy π^*, he is guaranteed to be taking the optimal path towards reaching his goal of maximizing the expected discounted return. It is important to notice that in a stochastic environment where there exist $s \in \mathcal{S}$ and $a \in \mathcal{A}$ such that $p(s', r|s, a) \neq 1$ if this adversary decides to follow π^* he is not guaranteed to get to the maximum return possible due to this stochasticity but to get the maximum expected return a priori, which is different.

Now that we have defined the optimal solutions (21), (22) and (23) to the MDP model we will analyze the different techniques and procedures used in RL to compute them. Within RL three different techniques are the most used for solving MDP: Dynamic Programming, the Monte-Carlo method and Temporal Difference learning. In this project, as a first step towards implementing RL techniques in power systems, we will just describe the two first techniques and encourage the reader to consider Temporal Difference learning for possible future versions of this tool.

3.6 DYNAMIC PROGRAMMING

By Dynamic Programming (DP), we refer to the collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP [27]. These algorithms assume full knowledge of the MDP model which means it knows the 4-tuple presented in (8) where $\mathcal{P}$ is the set of transition probabilities $p(s', r|s, a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$ and $s' \in \mathcal{S}^+$. For the purpose of this project, we will present three algorithms within DP which will prove to be useful within our framework: backward induction, policy iteration and policy iteration.

3.6.1 Backward Induction

Backward induction is the first and most basic algorithm within DP we will present. This algorithm solves the Bellman equation (21) by computing the optimal actions and value functions from the last possible decision at time-step T in a MDP to the initial one $t = 0$. If we take a close look at (21) and we use the value of the terminal states $V(s)$ which are defined by the nature of the MDP we can create a system of $|\mathcal{S}|$ equations (one for every $s \in \mathcal{S}$) with a total of $|\mathcal{S}|$ variables $V^*(s)$. This set of non-linear equations cannot be solved with linear techniques but can be solved by reasoning computing the solutions backward in time, starting from the end step until the initial one. So if we denote by $\mathcal{S}(t)$ to the set of all possible states achieved at time-step t starting from the initial state and $T_{\max}$ to the maximum value of the last possible time-step t in the MDP we can describe this technique as shown in Algorithm 1.

1. Initialization:
   $V^*(s) \in \mathbb{R}$ for all $s \in \mathcal{S}^+, s \notin \mathcal{S}$
2. Backward reasoning:
   Loop for each time-step: $t = T_{\max} - 1, T_{\max} - 2, \ldots, 0$:
   Loop for each $s \in \mathcal{S}(t)$
   $V^*(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V^*(s')]$

Algorithm 1: Backward induction with $V^*(s)$

In this project, we will use this algorithm but applied $Q^*(s,a)$ instead of $V^*(s)$. In the backward induction model we will present in Part III the terminal states will have value zero and therefore we will perform the initialization step in Algorithm 1 as $Q^*(s, \cdot) = 0$ for all terminal states and we will use equation (22) in the last line of code instead of (21).

Backward induction presents the advantage of finding the optimal solution at once, nevertheless over large state-spaces some problems may become really inefficient and therefore other more efficient iterative algorithms are used, namely, policy iteration and value iteration. Even though we will not implement these algorithms we will present them in this project because they could easily be implemented in our tool and because they use a general framework that will help us understand the Monte-Carlo method in Section 3.7.

3.6.2 Policy iteration

This algorithm can compute the optimal policy π^* and optimal value function V^* for a MDP by starting with an initial solution guess V_0 and π_0 and iteratively improving them as v_{π_k} and π_k until convergence is achieved and therefore optimal solution obtained: $V_{\pi_k} = V^*$ and $\pi_k = \pi^*$. We can sketch the basic skeleton of Policy iteration as shown in equation (24) and Figure 10 where we use v_{π_k} as an approximated value function of V_π ($v_{\pi_k} \approx V_{\pi_k}$). As we can see, this algorithm is made up of two parts: policy evaluation and policy improvement.

$$\pi_0 \xrightarrow{\text{Evaluate}} v_{\pi_0} \xrightarrow{\text{Improve}} \pi_1 \xrightarrow{\text{Evaluate}} v_{\pi_1} \xrightarrow{\text{Improve}} \dots \xrightarrow{\text{Evaluate}} \pi^* \xrightarrow{\text{Improve}} v^* \tag{24}$$

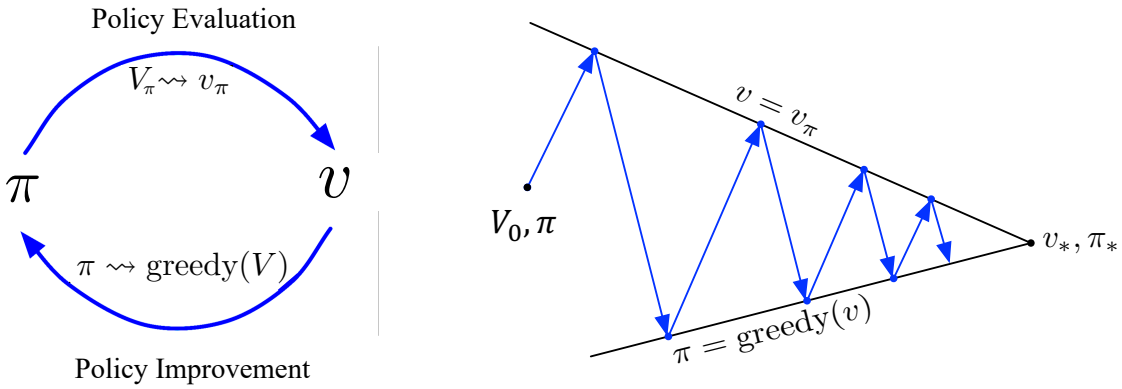

Figure 10: Policy evaluation and policy improvement in Monte-Carlo

1. **Policy Evaluation**: in the first part, as its name suggests, we evaluate all the states with respect to a given policy π_k. Following the reasoning developed with backward induction, this could easily be accomplished by setting a linear system of equations made up of $|\mathcal{S}|$ equations with the form of (18) and $|\mathcal{S}|$ variables $V_\pi(s)$. Nevertheless, this is not very efficient as stated in backward induction and so this procedure uses an iterative approach. Given a policy π, and using an initial value function approximation $v_\pi^{(0)}$, we iteratively improve it by using the Bellman equation (18). Each improvement of $v_\pi^{(k)}$ to $v_\pi^{(k+1)}$ is done by updating the value of a state $s \in \mathcal{S}$ using the values $v_\pi^{(k-1)}$ for states s' in the Bellman equation. Formally this $v_\pi^{(k)}$ is guaranteed to converge as $k \to \infty$, nevertheless in practice, for efficiency reasons, we usually stop before the iteration converges and give an approximate solution $v_\pi \approx v_\pi^{(k)}$ to policy improvement. This is called the truncated policy evaluation.

2. **Policy Improvement**: this second part consists of computing the optimal policy for a specific value function V, which we refer to as the greedy policy. This is done straightforward as shown in line 4 of step 3 in Algorithm 2.

So if we want to compute V^* and π_0 we can use an initial guess V_0 and π_0 and iteratively use policy evaluation and policy improvement to get a sequence of monotonically improving policies π_k and v_{π_k} which is guaranteed to converge to the optimal solution. This is exactly what policy iteration does as we can see in Algorithm 24 where, instead of computing V_π in policy evaluation we use the truncated policy evaluation which stops when $|v - V(s)|$ is less than an input parameter θ.

1. Initialization:
   Algorithm parameter: a small threshold $\theta > 0$ determining the accuracy of estimation.
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}^+$ except $V(\text{terminal}) = 1$
2. **Policy Evaluation**:
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
3. **Policy Improvement**:
   policy-stable← true
   For each $s \in \mathcal{S}$:
   old-action← $\pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If old-action$\neq \pi(s)$, then policy-stable ← false
   If policy-stable, then stop an return $V \approx V^*$ and $\pi \approx \pi^*$, else go to 2.

Algorithm 2: Policy Iteration

As we can see, in the algorithm within the policy evaluation part, the amount of iterations we compute depends on the value of θ. The bigger this value is the fewer iterations needed to finish policy iteration so if this algorithm converges regardless of the value of θ then it must converge regardless of the number of iterations within policy evaluation.

3.6.3 Value iteration

Value iteration takes advantage of the fact policy iteration converges regardless of the number of iterations within policy evaluation to combines policy evaluation and policy improvement into one single step. Value iteration is the same policy iteration but computing just one update to all the states (which we denote by a sweep) in policy evaluation. If instead of updating all the values after every sweep, we update them every time we compute $V(s)$ we would get a version of value iteration which is shown in Algorithm 3.

Like policy iteration, value iteration is also guaranteed to converge to the optimal solution and therefore, both of them are suited for solving a known MDP. Both algorithms are widely used and is not known which, if either, is better in general.

1. Initialization:
   Algorithm parameter: a small threshold $\theta > 0$
   $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}^+$ except $V(\text{terminal}) = 1$
2. Value iteration update:
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$
   Output a deterministic policy, $\pi \approx \pi^*$, such that:
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Algorithm 3: Value Iteration

Despite the fact that value iteration and policy iteration are DP algorithms that may converge to the optimal solution after a long time they present the advantage of being able to offer solutions in real-time with the last best solutions obtained in comparison to backward induction. This characteristic of generating data on-demand is very appealing when applying RL to power systems and we will see it as we move forward again in the Monte-Carlo method. This presented DP techniques can be very useful in many applications but sometimes other techniques for solving MDPs are needed due to three main limitations:

- **Operations over entire state-space of MDP**: with large state-spaces this can either be unfeasible computationally (e.g. Chess or Go) or not efficient. Also, this means that by exploring the entire state-space we lose the chance of focusing on specific areas of the state-space if we know they are relevant a priori.

- **Full knowledge of the environment dynamics**: this assumption is a very strong assumption that limits the possible applications DP considerably. In most real-life applications we do not know the full model and transitional probabilities $p(s', r|s, a)$ ahead of time and therefore methods that assume no known model (model-free) need to be developed for these applications.

- **No learn from experience interaction**: DP techniques plan the optimal solutions or approximations without the need for interacting with the environment. These solutions are computed without any experience in the environment and therefore offer solutions which have never been tried.

To offer solutions to these three limitations of DP other methods like Monte-Carlo and Temporal Difference have been developed. In the next section, we will explain the Monte-Carlo method in RL in detail.

3.7 MONTE-CARLO METHOD

The Monte Carlo method learns from experience, which is modeled as a set of episodes. By episodes, which we will denote as $\boldsymbol{E}$, we refer to any sequence of states, actions and rewards which end in a terminal state as shown in (25). The fact that these episodes can either be generated by an actual environment or by a simulation makes Monte-Carlo especially appealing for applying it to power systems.

$$\boldsymbol{E} = \{(s_i, a_i, r_{i+1})\}_{i=0}^{T-1} \tag{25}$$

This method does not require the full knowledge of the 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ and therefore does not need the four-argument joint distribution p, it just needs sufficient knowledge of the model to compute sample transitions to get episodes. The Monte-Carlo method computes estimates of the expected discounted returns for the value functions Q and V by averaging their sampled returns from episodes. So for example, if we want to evaluate the value function V_π, Monte-Carlo would generate episodes following that policy π and then would compute the average return for each state to compute the value estimations v_π just as shown in Algorithm 4.

1. Initialization:
   Input: a policy π to be evaluated
   Initialize:
   $V(s) \in \mathbb{R}$, arbitrarily, $\forall s \in \mathcal{S}$
   Returns$(s) \leftarrow$ empty list, $\forall s \in \mathcal{S}$

2. Learning from experience:
   Repeat forever (for each episode):
   Choose $S_0 \in \mathcal{S}$ randomly so that all states have probability > 0
   Generate an episode following π: $S_0, A_0, R_1, \dots S_{T-1}, A_{T-1}, R_T$
   $G \leftarrow 0$
   Loop for each step of episode, $t = T-1, T-2, \dots, 0$
   $G \leftarrow \gamma G + R_{t+1}$
   Append G to Returns(S_t, A_t)
   $V(S_t) \leftarrow$average(Returns(S_t))

Algorithm 4: Monte Carlo for estimating $V \approx V_\pi$

This algorithm procedure is guaranteed to converge to the true expected value V_π as the visits to all the states grow to infinity. Nevertheless, visiting every state following a specific policy can only be accomplished either when the episodes start from random states S_0, as is the case of Algorithm 4, or when evaluating stochastic policies which assign a nonzero probability to every action for all the states, which we refer to as soft policies.

Within the Monte-Carlo method, we can distinguish first-visit and every-visit methods depending on whether the estimates of v_π (or q_π) are the average of the returns following the first visit or every visit for every state (or every state-action pair). In our project, in the MDP we will use, there is no possibility of visiting a state several times and therefore first-visit and every-visit methods will give us the same solutions.

The Monte-Carlo algorithm for computing the optimal solution to an MDP has the same structure of policy evaluation and policy improvement shown in Figure 10 but using q instead of v as we can see in (26). It uses q because for computing the greedy policy of v the transitional probabilities $\mathcal{P}$ are needed and they are unknown. On the contrary, computing the greedy policy of q is straightforward as we can see in (23).

$$\pi_0 \xrightarrow{\text{Evaluate}} q_{\pi_0} \xrightarrow{\text{Improve}} \pi_1 \xrightarrow{\text{Evaluate}} q_{\pi_1} \xrightarrow{\text{Improve}} \dots \xrightarrow{\text{Evaluate}} \pi^* \xrightarrow{\text{Improve}} q^* \tag{26}$$

Similarly to DP, in Monte-Carlo, the action value function q_{π_k} is repeatedly updated to a more close approximation $q_{\pi_{k+1}}$ of Q_{π_k}, and the policy is constantly improved with respect to the current value function to approach optimality. Just as we did in DP, we will explain policy evaluation and policy improvement separately and then we will show the pseudocode for the complete algorithm.

1. **Policy evaluation**: when evaluating a policy Q_{π_k} we can use a similar approach to the one shown in Algorithm 4 but with actions instead of states. This algorithm procedure for computing q_π would be guaranteed to converge to Q_π as the visits to all the states-action pairs grow to infinity. Nevertheless, in this case, instead of using a random state-action initialization, we will use a soft policy called ϵ-greedy policy starting from the initial state in the MDP for ensuring visits to all the states when computing the episodes. This ϵ-greedy policy will assign a probability of $1-\epsilon$ to the action we estimate to be optimal and a probability of ϵ to the rest of actions. In this project, we will use equal distribution probabilities for the expected non-optimal actions and so for example if we have a state with $\pi^*\left(s^{(3)}\right) = a$ and $\mathcal{A}(s) = \{a^{(i)}\}_{i=1}^{4}$ and we use $\epsilon = 0.4$ we would get the ϵ-greedy policy $\pi\left(a^{(i)}|s\right)$ as shown in Figure 11.

2. **Policy improvement**: in this step, just as we did in DP, we will update the greedy policy and transform it to a ϵ-greedy policy just as shown in the last 3 lines of code in Algorithm 5.

As we can see while in DP we used the greedy policy for converging to the optimal solution in the Monte-Carlo method we will use ϵ-greedy policies and therefore we will not converge to the optimal policy but to the ϵ-greedy optimal policy. And just as happened with truncated policy evaluation in DP where the solution converged to optimality regardless of the number of iterations used in this step, we will take advantage of this with Monte-Carlo. The version we will use of Monte-Carlo computes the updates q_{π_k} and π_{k+1} for every episode just as shown in Algorithm 5. This algorithm is the one we will apply for our power system framework and will be the key contribution of this thesis.

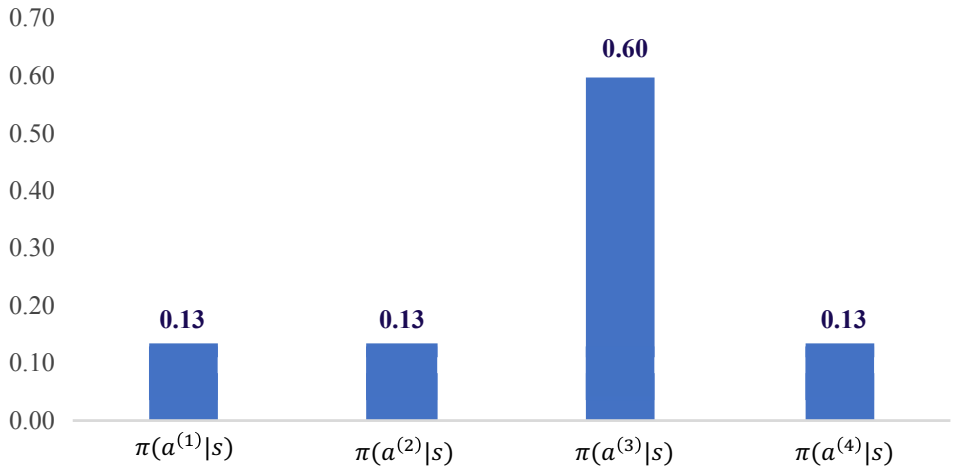

Figure 11: ϵ-greedy policy example for ϵ=0.4, $\pi^*(s) = a^{(3)}$ and $\mathcal{A}(s) = \{a^{(i)}\}_{i=1}^{4}$

1. Initialization:
   Algorithm parameter: small $\epsilon > 0$
   Initialize:
   $\pi \leftarrow$ an arbitrary ϵ-soft policy
   $Q(s,a) \in \mathbb{R}$ (arbitrarily), $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
   Returns$(s,a) \leftarrow$ empty list, $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
2. Learning from experience:
   Repeat forever (for each episode):
   Generate an episode following π: $S_0, A_0, R_1, \ldots S_{T-1}, A_{T-1}, R_T$
   $G \leftarrow 0$
   Loop for each step of episode, $t = T-1, T-2, \ldots, 0$
   $G \leftarrow \gamma G + R_{t+1}$
   Append G to returns(S_t, A_t)
   $Q(S_t, A_t) \leftarrow$ average$\big($Returns$(S_t, A_t)\big)$
   $A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
   For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)|, & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)|, & \text{if } a \neq A^* \end{cases}$$

Algorithm 5: Monte Carlo

4 POWER SYSTEMS

Once we have fully explained the RL framework we will use for this project, we will present the theory underlying this thesis from an electric perspective. All with the goal of combining these two materials in Part III, Part IV and Part V of this thesis. In this section, therefore, we will explain the basics about electric power systems, the power flow equations and ultimately a method for solving them known as Newton Raphson.

4.1 ELECTRIC GRID

An electric power system is a network of electrical components employed to supply, transfer and use electric power. The electric grid is the power system used for providing power to extended areas. As we can see in the electric supply chain shown in Figure 12 it is made up of 6 parts [5]:

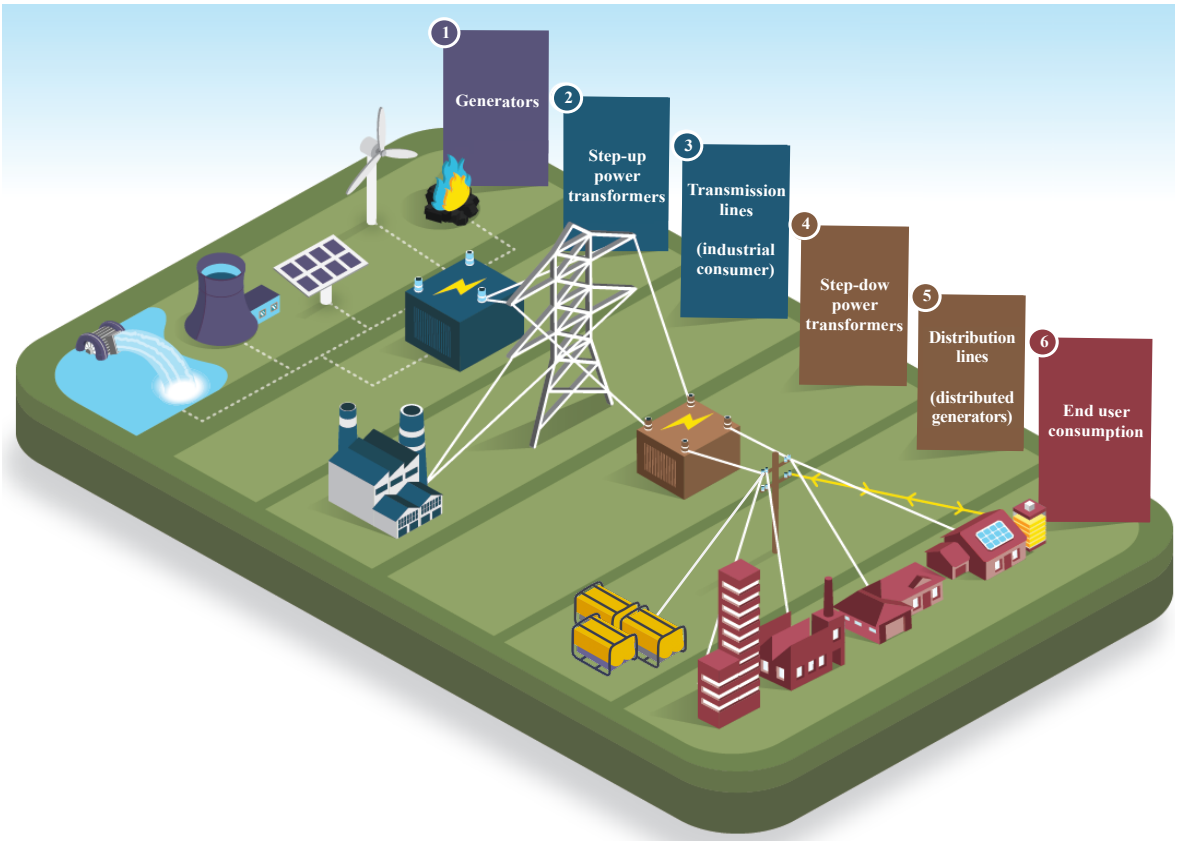

Figure 12: Electric grid power system structure

1. **Generation stations**: produce electric power from different energy sources and technologies. In the United States, the three major energy sources for electricity generation are fossil fuels (coal, natural gas, and petroleum), nuclear energy, and renewable energy sources. According to U.S. Energy Information administration this account for 62%, 20% and 17% respectively in 2019 in the United States [4]. The most used technologies for these sources are steam turbines, gas turbines, hydro turbines, wind turbines, and solar photovoltaics. These generators produce 3-phase AC power with a frequency of 60 Hz in most parts of America including the United States and 50 Hz in Europe and most parts of Africa and Asia.

2. **Step-up power transformers**: these electrical substations use large transformers to convert the generators voltages (in the thousands of volts level) up to extremely high voltages for long-distance transmission in the transmission lines. This way power delivery to the customers is made more efficient as current levels drop and so the power losses we showed in (1). There is a generator transformer for each generating unit and it has a voltage and power rating according to the size of that unit.

3. **Transmission lines**: they travel long distances form the generation locations to the distribution grid located in populated areas with typical voltages in the range of 155 kV to 765 kV [3]. Approximately, transmission lines with a length less than 50 km are considered short, between 50 km and 200 km are considered medium and longer than 200 km are considered long. These transmission lines are supported by transmission towers which height can vary from 15 m to 55 m. Some large industrial consumers take their supply at this point.

4. **Step-down power transformers**: these distribution substations convert high-voltage electricity to low voltage for primary distribution lines. These primary lines have medium voltages ranging from 2.3 kV to 39 kV [26].

5. **Distribution lines**: they carry the electricity from the transmission system to the individual consumers. This is known as the distribution system and is made up of primary lines with medium voltages which, once near the customer, are step down to consumer level voltages in the secondary lines. This low voltage ranges usually from 120 to 240 V in the United States for residential customers [26]. As we can see in Figure 12, small-scaled technologies such as solar panels and combined heat and power may generate in the distribution system. This generation, known as the distribution generation, accounts for approximately one-sixth of the capacity of the nation's existing centralized power plants [2].

6. **End-user consumption**: homes, offices and factories use electricity for lighting, for heating and for power appliances. Some homes as shown in Figure 12 have solar panels and some have their battery storage system. In 2018, the average annual electricity consumption for a US residential utility customer was 10,972 kWh, an average of about 914 kWh per month [6].

4.2 POWER FLOW EQUATIONS

Once we have explained completely how electric systems work and which are their main parts, the first thing we need to understand if we want to see what is happening in the grid is the power flow equations. These will allow us to compute the voltages, currents and power flows in every node and every line in a system and therefore will allow us to check the three primary constraints presented in Section 2. The power flow equations give fundamental information for the operation, exploitation and planification of energy systems and are the previous step for many possible analyses. They will prove to be essential for our security and contingency analysis we aim to build in this project.

In Figure 13 we show an example of a power system made up of 14 buses. These buses are the points where the terminals of two or more elements meet and they are represented with vertical and horizontal lines in bold. These nodes can be connected through lines between them or to other elements such as generators (like buses 2, 12 and 13), transformers (like buses 1 to 7 and 11 to 14), supply nodes (buses 7, 8 and 10) or to a special type of generator known as the slack generator (bus 3).

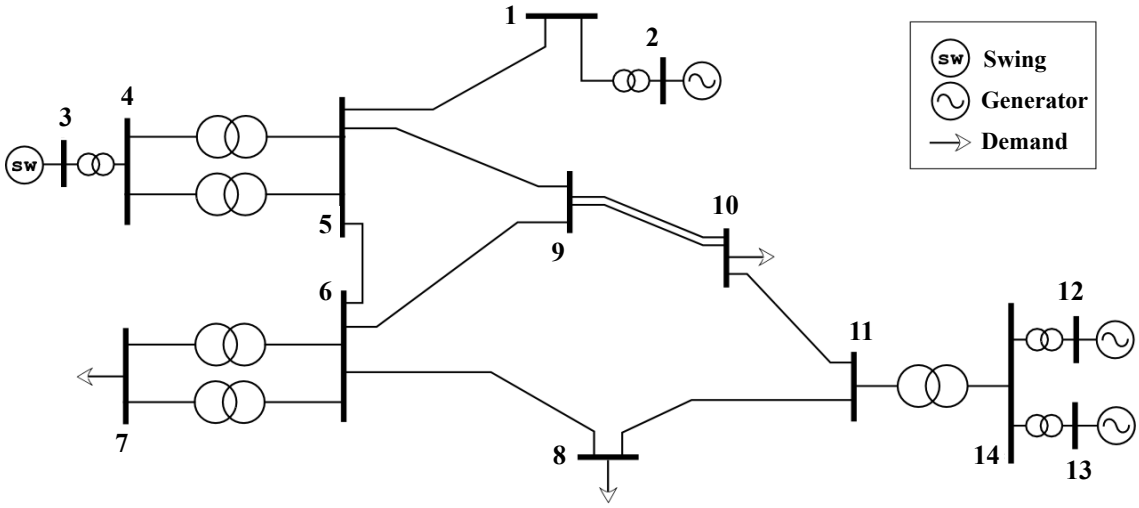

Figure 13: 14-bus power system

For this model we will denote as $\bar{I}_i$ and $\bar{V}_i$ to the total current flowing out of bus i and to the voltage in bus i respectively. Both of these variables are complex numbers that represent phasors, for example, $\bar{I} = Ie^{j\theta}/\sqrt{2}$ represents a sinusoidal function with an amplitude of I, a frequency determined by the grid ($f = 60$ Hz in the United States) and an initial phase θ. Also for referring to the current flowing from bus i to bus j we will use $\bar{I}_{ij}$ and similarly we will denote as $\bar{Z}_{ij}$ and $\bar{Y}_{ij}$ to the impedance and the admittance between this buses. Using this notation and assuming some basic knowledge about phasors and about the Ohm's law applied with phasors we can state equation (27) where $\mathcal{K}$ denotes the set of buses connected to bus i:

$$\bar{I}_i = \sum_{k \in \mathcal{K}} \frac{\bar{V}_i - \bar{V}_k}{\bar{Z}_{ij}} = \sum_{k \in \mathcal{K}} \bar{Y}_{ij} (\bar{V}_i - \bar{V}_k) \tag{27}$$

This simple linear relation between $\bar{I}_i$ and the voltage differences $\bar{V}_i - \bar{V}_k$ could by stated for each bus of a power system with a total of n buses and therefore we could get a system of n equations, relating $\bar{V}_i$ and $\bar{I}_i$ for every $i = 1, \ldots, n$. If we use the following notation $\bar{I} = [\bar{I}_1 \quad \bar{I}_2 \quad \ldots \quad \bar{I}_n]^\top$ and $\bar{V} = [\bar{V}_1 \quad \bar{V}_2 \quad \ldots \quad \bar{V}_n]^\top$ we could define a matrix called the admittance matrix denoted as $\boldsymbol{Y}$ to relate $\bar{I}$ and $\bar{V}$ as:

$$\bar{I} = \boldsymbol{Y}\bar{V} \tag{28}$$

Where:

- $\boldsymbol{Y}_{ij}$ is the negative sum of all the series admittances connected between i and j.
- $\boldsymbol{Y}_{ii}$ is the sum of all the admittances connected to node i.

So, if we use (28) we can compute the current $\bar{I}_i$ for any node in the network as:

$$\bar{I}_i = \sum_{k=1}^{n} \bar{\boldsymbol{Y}}_{ik} \bar{V}_k \tag{29}$$

Also if we denote by $\bar{S}_i$ to the apparent power entering in a network through bus i, P_i to its real part, the real power, and Q_i to its imaginary part, the reactive power, we can get the initial equality shown in (30). If we express $\bar{S}_i$ in terms of $\bar{V}$ and $\bar{I}$ and we use the admittance matrix $\boldsymbol{Y}$ we could develop the following equation:

$$\begin{aligned} \bar{S}_i &= P_i + jQ_i \\ &= \bar{V}_i \bar{I}_i^* \\ &= \bar{V}_i \left(\sum_{k=1}^{n} \bar{\boldsymbol{Y}}_{ik} \bar{V}_k \right)^* \end{aligned} \tag{30}$$

If we express the module and the argument of $\bar{V}_i$ as V_i and θ_i respectively and we separate the real part from the imaginary part for each element in the admittance matrix we have:

$$\begin{cases} \bar{V}_i = V_i e^{j\theta_i} \\ \boldsymbol{Y}_{ik} = \boldsymbol{G}_{ik} + j\boldsymbol{B}_{ik} \end{cases} \tag{31}$$

So, by substituting (30) in (31) we can get:

$$P_i + jQ_i = V_i e^{j\theta_i} \left(\sum_{k=1}^{n} (\boldsymbol{G}_{ik} + j\boldsymbol{B}_{ik}) V_k e^{j\theta_k} \right)^* \tag{32}$$

As we can conclude from this equation, the active and the reactive power are completely determined when we have the complete admittance matrix, which is given by the power system, and the variables V and $\bar{\theta} = [\bar{\theta}_1 \quad \bar{\theta}_2 \quad \dots \quad \bar{\theta}_n]^\top$. So, if we reconfigure (32) and we separate the real part from the imaginary part we can get the functions $p_i(\theta, V)$ and $q_i(\theta, V)$ as:

$$\begin{cases} p_i(\theta, V) = \sum_{k=1}^{n} (\boldsymbol{G}_{ik} \cos(\theta_i - \theta_k) + \boldsymbol{B}_{ik} \sin(\theta_i - \theta_k)) \\ q_i(\theta, V) = \sum_{k=1}^{n} (\boldsymbol{G}_{ik} \sin(\theta_i - \theta_k) - \boldsymbol{B}_{ik} \cos(\theta_i - \theta_k)) \end{cases} \tag{33}$$

We can appreciate in (29) and (33) that when we have V and θ we can get the active power, the reactive power and the currents for every bus in a system and, hence, define the complete state of the system. Power systems are made to satisfy specific power demands by the consumer and therefore they use equations to the relate active and the reactive power of the buses with the variables V and θ. These equations are known as the power flow equations which we can simplify as:

$$\begin{cases} P_i = p_i(\theta, V) \\ Q_i = q_i(\theta, V) \end{cases} \tag{34}$$

These power flow equations can be used as long as the number of equations is equal or smaller to the number of unknown variables. If we classify the types of buses according to the known and unknown variables P_i, Q_i, θ_i and V_i in a system of n buses and m generators, we can get three types of nodes: PV buses, PQ buses and the slack bus, also known as the swing bus of the reference bus.

	Number	$\boldsymbol{P_i}$	$\boldsymbol{Q_i}$	$\boldsymbol{V_i}$	$\boldsymbol{\theta_i}$
PQ buses	$m-1$	✓	✓	×	×
PV buses	$n-m$	×	✓	×	
Slack bus	1	×	×	✓	✓
Total	n	$n-1$	$n-m$	m	1
		$\boldsymbol{2n-m-1}$ equations		$\boldsymbol{2n-m-1}$ variables	

Table 2: Classification of nodes according to the known variables in the power flow equations

- **PQ buses**: these nodes are the ones we would get in parts 3 and 6 in Figure 12 where the energy is supplied to the consumer. They are the load buses and they specify an active and reactive power demand. There is a total of $n - m$ of these buses where P_i and Q_i are known as shown in Table 2.

- **PV buses**: this type of nodes are the generators, the ones we would get in part 1 and 5 in Figure 12. These buses are known as the generation buses or the voltage control buses. They provide a constant real power P_i, controlled through a prime mover, and maintain a constant voltage magnitude by controlling the injection of reactive power. There is a total of m nodes of this type where P_i and V_i are determined.

- **Slack bus**: this is a special node where a generator is modeled as the angle reference for all the buses in a system and also modeled to account for the unknown power losses presented in (1). These power losses, which we will denote as ξ, are equal to the total sum of power injections in a network as we can see in 35. As the PV buses, they maintain a constant voltage by injection of reactive power. There is only 1 bus of this type in a power system where V_i and θ_i are known and P_i and Q_i completely unknown just as shown in Table 2.

$$\xi = \sum_{i=1}^{n} p_i(\theta, V) \tag{35}$$

In conclusion, as shown in Table 2, using the presented notation, we could get a total of $2n - m - 1$ equations for P_i and Q_i, and a total of $2n - m - 1$ variables from the $2n$ voltage and angle variables (only $m + 1$ of them are known). This would make a system of non-linear equations which could give us complete information about the state of a system. Due to the non-linearity, they are usually solved by using iterative numerical methods. In this thesis, we will use the most used one which is the Newton-Raphson method.

4.3 NEWTON-RAPHSON METHOD FOR POWER FLOW

4.3.1 Newton-Raphson for a one-variable single equation $f(x) = 0$

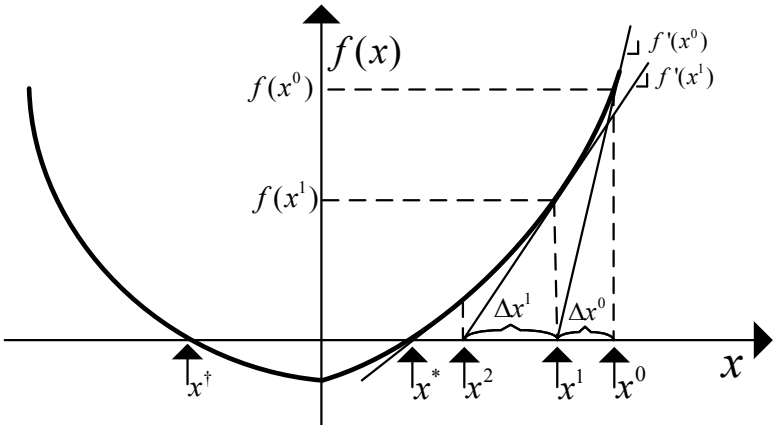

Figure 14: Application of Newton-Raphson to a one-variable single equation $f(x)$

In order to easily understand the Newton-Raphson method we will explain it by first applying it to a simple equation $f: \mathbb{R} \to \mathbb{R}$. This method solves equations stated in the form of (36) and takes advantage of the first-order Taylor approximation of $f(x)$ to try to compute the solution of x.

$$f(x) = 0 \tag{36}$$

Given a point $\left(x^{(k)}, f\left(x^{(k)}\right)\right)$ we can approximate the value of $f\left(x^{(k+1)}\right)$ well when $x^{(k+1)}$is near $x^{(k)}$as shown in (37). Both the function $f(x)$ and its linear approximation in $x^{(0)}$and $x^{(1)}$ are represented in Figure 14.

$$f\left(x^{(k+1)}\right) \approx f\left(x^{(k)}\right) + \left.\frac{\partial f}{\partial x}\right|_{x^{(k)}} \left(x^{(k+1)} - x^{(k)}\right) \tag{37}$$

If we want to compute the root of this linearization, we could get $x^{(k+1)}$as:

$$x^{(k+1)} = x^{(k)} + \left(\left.\frac{\partial f}{\partial x}\right|_{x^{(k)}}\right)^{-1} f\left(x^{(k)}\right)$$

This is how the Newton-Raphson method works, it starts with an initial point $\left(x^{(0)}, f\left(x^{(0)}\right)\right)$ and it iteratively computes the linear approximation of $f(x)$ at point $x^{(k)}$ and calculates it's root $x^{(k+1)}$ for $k = 0,1,2, \ldots$ until it eventually gives a approximate solution. This iteration can be computed indefinitely unless we set a stopping criterion. In this project, we will define a maximum tolerance ϵ for $\left|x^{(k)} - x^{(k-1)}\right|$. If this criterion is not satisfied within a maximum number of iterations n we will state that the Newton-Raphson method has not converged and therefore we will not get any solution. As we can see in Figure 14, this method, depending on the initial point $\left(x^{(0)}, f\left(x^{(0)}\right)\right)$, can converge to different solutions and therefore a proper initial point will be essential for achieving the solution we want.

4.3.2 Newton-Raphson for the power flow equations

Similarly, as we did with the case of one function $f: \mathbb{R} \to \mathbb{R}$, we can apply the Newton-Raphson method to a more general function $f: \mathbb{R}^n \to \mathbb{R}^n$ and ultimately use it for solving the power flow equations. If we define the power flow equations as functions equal to zero, we can define the functions MP_i and MQ_i as mismatches between the specific active and reactive power, P_i and Q_i, and the calculated power $p_i(\theta, V)$ and $q_i(\theta, V)$ dependant on θ and V as:

$$\begin{cases} MP_i = P_i - p_i(\theta, V) & \forall i = 1, \ldots, n \\ MQ_i = Q_i - q_i(\theta, V) & \forall i = 1, \ldots, n \end{cases}$$

Then the solution we are seeking is the one for which $MP_i = 0$ and $MQ_i = 0$ for all $i = 1, \ldots, n$. For notation simplicity, we will use, from now on, bus 1 to refer to the slack bus, buses 2 to m to refer to the PV nodes and buses $m + 1$ to n for the PQ nodes. As we have already seen, the state of a system is completely determined by V and θ, therefore, if we reduce these $2n$ equations for just computing every V_i and θ_i we can compute a reduced system of equations. For this purpose, we will compute the active power mismatches for all the nodes and the reactive power mismatches just for nodes PQ as shown:

$$\begin{cases} MP_i = 0 & \forall i = 1, \ldots, n \\ MQ_i = 0 & \forall i = 1, \ldots, n \end{cases} \tag{38}$$

Using these equations, we would have a total of $2n - m$ equations and a total of $2n - m$ variables as shown in (39) and (40).

- Equations:

$$\underbrace{MP_1}_{\substack{\text{Slack bus}\\(1)}},\underbrace{MP_2,\dots,MP_m}_{\substack{\text{PV buses}\\(m-1)}},\underbrace{MP_{m+1},MQ_{m+1},\dots,MP_n,MQ_n}_{\substack{\text{PQ buses}\\2(n-m)}} \tag{39}$$

- Variables:

$$\underbrace{\xi}_{\substack{\text{Losses}\\(1)}},\underbrace{\theta_2,\dots,\theta_m}_{\substack{\text{PV buses}\\(m-1)}},\underbrace{\theta_{m+1},V_{m+1},\dots,\theta_n,V_n}_{\substack{\text{PQ buses}\\2(n-m)}} \tag{40}$$

We will express this system of equations using a function $F\colon \mathbb{R}^{2n-m} \to \mathbb{R}^{2n-m}$ for the mismatches and a vector variable $X \in \mathbb{R}^{2n-m}$ for the set of unknown variables as:

$$F(\xi,\theta,V) = \begin{bmatrix} f_p(\xi,\theta,V) \\ f_q(\theta,V) \end{bmatrix} = \begin{bmatrix} MP_1 \\ \vdots \\ MP_n \\ MQ_{m+1} \\ \vdots \\ MQ_n \end{bmatrix} \qquad X = \begin{bmatrix} \xi \\ \theta \\ V \end{bmatrix} = \begin{bmatrix} \xi \\ \theta_2 \\ \vdots \\ \theta_n \\ V_{m+1} \\ \vdots \\ V_n \end{bmatrix} \tag{41}$$

It is worth noting that, from now on, we will denote as θ and V to the vector which elements are just the unknown θ_i and V_i for all $i = 1,\dots,n$. Using the presented notation, similarly to the one single equation example, we can express the first-order Taylor approximation of $F(X)$ at $X = X^{(k)}$ as:

$$F\left(X^{(k+1)}\right) \approx F\left(X^{(k)}\right) + \boldsymbol{J}\left(X^{\{(k)\}}\right)\left(X^{\{(k+1)\}} - X^{\{(k)\}}\right) \tag{42}$$

Where $\boldsymbol{J}\left(X^{(k)}\right)$ is the jacobian matrix of $F(X)$ with respect to X computed at $X = X^{(k)}$:

$$\boldsymbol{J}\left(X^{(k)}\right) = \begin{bmatrix} \dfrac{\partial f_p}{\partial \xi} & \dfrac{\partial f_p}{\partial \theta} & \dfrac{\partial f_p}{\partial V} \\ 0 & \dfrac{\partial f_q}{\partial \theta} & \dfrac{\partial f_q}{\partial V} \end{bmatrix}_{X=X^{(k)}}$$

Hence, if we compute the root solution of the linearization we would get:

$$X^{(k+1)} = X^{(k)} - \left(\boldsymbol{J}\left(X^{(k)}\right)\right)^{-1} f\left(X^{(k)}\right)$$

The Newton-Raphson method applied to power systems works just as we have seen with the one-variable single equation $f(x)$. It starts with an initial solution $\left(X^{(0)}, F\left(X^{(0)}\right)\right)$ and iteratively computes $X^{(k)}$ using the Taylor first-order approximation for $k = 0,1,2, \ldots$ until it eventually returns an approximate solution. If we use the same stoping criteria as the one described for $f(x)$, we can present the pseudocode for the Newton-Raphson method applied to power systems in Algorithm 6. It is important to be aware that this method is not guaranteed to converge and that it may give a different solution to the one we are interested in when it converges. To avoid these problems a close initialization $X^{(0)}$ to the solution X^* is essential.

1. Initialization:
   Algorithm parameter: number of iterations N and/or small maximum tolerance $\epsilon > 0$
   Initialize: $X_0 = [\xi^{(0)} \quad \theta^{(0)} \quad V^{(0)}]$ (e.g. flat start)
   $k \leftarrow 0$
2. Taylor first order solutions:
   Loop:
   Compute the Jacobian in $X^{(k)}$: $J\left(X^{(k)}\right)$
   Compute the mismatches in $X^{(k)}$: $f\left(X^{(k)}\right)$
   $X^{(k+1)} \leftarrow X^{(k)} - \left(\boldsymbol{J}\left(X^{(k)}\right)\right)^{-1} f\left(X^{(k)}\right)$
   $k \leftarrow k + 1$
   Until $k = N$ or $\max_{i=1,\ldots,2n-m}\left|X_i^{(k)} - X_i^{(k-1)}\right| < \epsilon$
   Return the solution $X^{(k)}$

Algorithm 6: Newton Raphson method applied to power systems

Algorithm 6 is the one we will use for our security analysis model in this thesis. This presented version of the Newton Raphson method computes the exact jacobian matrix $J\left(X^{(k)}\right)$ and is known as the full Newton-Raphson algorithm. Computing $(2n - m)^2$ derivates for $J\left(X^{(k)}\right)$ may be computationally expensive and therefore there exist other algorithms that use an approximation of the jacobian matrix (i.e. decoupled power flow, fast decoupled power flow or DC approximation). These are more time-efficient but are also less exact. In our model, we will use the full Newton-Raphson algorithm for exactitude purposes but it could easily be adapted to any other solver method of the power flow equations.

Part III. PROBLEM FORMULATION

5 *MODEL DESCRIPTION*

So now that we have presented all the necessary background in both RL in Section 3 and power systems in Section 4 we will combine everything together into a framework able to provide situational awareness to the grid operators of the risk of cascading failure. These cascading failures are a perennial security threat for power systems as they are the usual mechanism by which failures propagate to cause large blackouts. Providing a device able to anticipate potentially dangerous lines that can cause these cascading failures could be an invaluable advisory tool for power grid operators.

Cascading failure is a process in a system of interconnected parts in which the failure of one or a few parts can trigger the failure of other parts and so on. In power transmission, they are normally caused by initial incidents, such as an electrical fault at a particular transmission line, and through fast propagation, they can lead to large-scale blackouts which can cause astronomical financial losses and affect countless people [29]. Through this project, we aim to create a tool able to assess the risk of cascading failure that would entail the loss of any transmission line.

In this manner, we assign an index to every line contingency assessing the risk of cascading failure in the power system. We pose this problem as a MDP, just as we did in Section 3.3, where we model an agent disconnecting lines one by one until all the lines are inoperative and a power system as an environment losing the lines disconnected by the agent plus the ones shut down due to possible cascading failures.

We will define the rewards as the number of cascaded lines after each disconnection by the agent. By doing so, the value of the functions Q^* and V^* will give us information about the maximum discounted sum of future cascading failures that can take place for every power system state and every line disconnection. This way the value functions with the highest values will be those where the cascading failures could be more severe. The value function V^* will assess the risk of cascade for a particular state of the system and Q^* will assess the risk of cascading failure associated with a line contingency in a particular state. These solutions will allow us to sort the lines from the most vulnerable ones to the most secure and also to differentiate between vulnerable power system configurations and more secure ones.

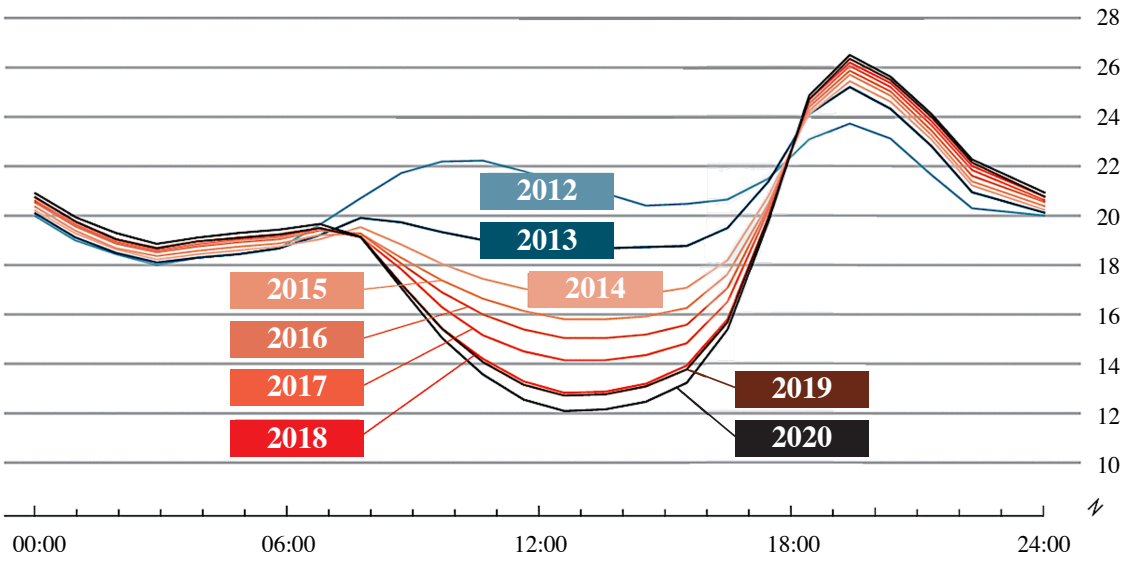

Figure 15: Typical spring day demand in California (GW) by California ISO

To completely set the MDP we first need to establish how we will define the states. As we have seen in Section 4.2, a power system is completely defined when we have V and θ determined. These values are computed by using the power flow equations and by fixing the corresponding known variables for the PQ nodes, the PV nodes and the slack bus. Nevertheless, because the power demand is dynamic, these values change over time. We illustrate this in Figure 15, where we can see the power demand profile in a typical spring day in California between 2012 and 2020 [22]. Considering V and θ in the definition of a state in our MDP model would involve a continuous infinite-state rather than a discrete finite one. For simplicity purposes, in this project, we will assume a constant demand profile and therefore we will not include V and θ in the definition of a state.

Considering this simplification, if we want to analyze the maximum number of cascaded lines, we can achieve by disconnecting lines in a system, a state could be simplified to the set of inoperative lines and a binary variable specifying whether the state is terminal or not. In this project, for clarity and simplicity purposes, we will separate the set of inoperative lines into two different subsets and we will define a state as a tuple of three components:

$$S = (\mathcal{D}, \mathcal{C}, \mathcal{T}) \tag{43}$$

- **The disconnected lines**, denoted by $\mathcal{D}$[2], refer to the set of disconnected lines by the agent to get to a certain state. Initially, $\mathcal{D} = \emptyset$. every time-step the agent is allowed to disconnect one line and, therefore, the total number of lines in $\mathcal{D}$ is equal to the time-step t, $|\mathcal{D}| = t$. If we assign an identification number to each of the lines in a system and denote as d_i to each of the lines disconnected by an agent, we can express $\mathcal{D}$ as:

$$\mathcal{D} = (d_1 \quad d_2 \quad \dots \quad d_t)$$

- **The cascaded lines**, denoted as $\mathcal{C}$, refer to the set of cascaded lines until arriving at the state S. Initially, this value is $\mathcal{C} = \emptyset$. We want to analyze the episodes that lead to the largest cascaded lines possible, meaning the ones with the largest possible $|\mathcal{C}|$. If we assign an identification number to each line, we can express $\mathcal{C}$ as we show in (44). It is important to realize that $\mathcal{D}$ and $\mathcal{C}$ are mutually exclusive sets and therefore $\mathcal{D} \cap \mathcal{C} = \emptyset$. Also, we should appreciate that the maximum number of lines within both sets is n_L: $\max|\mathcal{D}| + |\mathcal{C}| = n_L$.

$$\mathcal{C} = (c_1 \quad c_2 \quad \dots \quad c_{|\mathcal{C}|}) \tag{44}$$

- **The terminal state indicator**, which we will refer to as $\mathcal{T}$, refers to the boolean variable which expresses whether a state is terminal ($\mathcal{T} = 1$) or not ($\mathcal{T} = 0$) as shown in (45). We will consider terminal the states where all the lines are inoperative and therefore $|\mathcal{D}| + |\mathcal{C}| = n_L$. Additionally, because the full Newton-Raphson method is not guaranteed to find a solution, we will classify as terminal the cases where a non-convergence is achieved. This means that we won't explore states where we can't identify the overloaded lines and therefore, we will just focus on the states where V and θ are obtained. This problem of non-convergence could be solved by changing the Newton Raphson stopping criteria or by using another method for the power flow equations. In this project, for accuracy and simplicity, we will classify these cases as terminal. We will assign a value of zero to these states as shown in (46).

$$\mathcal{T} \in \{0,1\} \tag{45}$$

$$V^*(s) = 0 \qquad s \in \mathcal{S}^+, s \notin \mathcal{S} \tag{46}$$

Considering this notation, we will define an action, a_t, as the identification number of the line the agent disconnects after arriving at state s_t. As we express in (47), the set of possible actions to take from a state, denoted as $\mathcal{A}(s_t)$, are the set of operative lines at state s_t.

$$\mathcal{A}(s_t) = \{a_t : a_t \in \{1, \dots, n_L\}, a_t \notin \mathcal{D}_t, a \notin \mathcal{C}_t\} \tag{47}$$

[2] The introduced $\mathcal{D}$ is completely different from the one presented in Section 3.1 which denoted the structure of the information in the different learning techniques used within Machine Learning.

Also, we will denote as $\Delta\mathcal{C}_{t+1}$to the set lines which fail due to the cascade effect after taking action a_t. Therefore, when we take an action a_t from a non-terminal state $s_t = (\mathcal{D}_t, \mathcal{C}_t, \mathcal{T}_t)$, where $\mathcal{T}_t = 0$, we arrive at a new state s_{t+1} where $\mathcal{D}_{t+1} = \mathcal{D}_t \cup a_t$ and $\mathcal{C}_{t+1} = \mathcal{C}_t \cup \Delta\mathcal{C}_{t+1}$. Considering this, the reward function can be formulated as:

$$r_{\{t\}} = |\Delta\mathcal{C}_{t+1}| \tag{48}$$

As we showed in equation (16), to compute the expected return for any state or any state-action pair we need the summation of all the future rewards discounted by a certain factor γ^k. To illustrate an example, in Figure 16 we show the values of this discount factor for $\gamma = 0.9$ and $k = 1, \dots, 20$. As we can see, this parameter allows us to compute different indexes for assessing the cascading risk depending on how much we care about the future cascades relative to the more immediate ones.

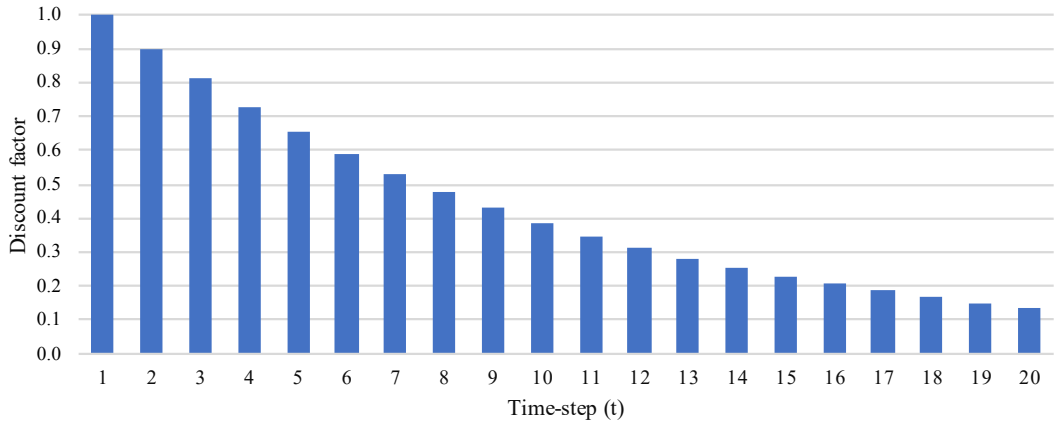

Figure 16: Discount factor γ^k used for computing V(s) and Q(s,a) with $\gamma = 0.9$

In this project, because we will assume a fixed demand profile with no variation, we will get a deterministic environment just as we show in (49). This means that every time the power system is in a specific state s and we disconnect a specific line a, the cascaded lines will always be the same and therefore the new state s' and the reward obtained r.

$$p(s', r|s, a) \in \{1,0\}. \qquad \forall\, s \in \mathcal{S}, a \in \mathcal{A} \tag{49}$$

Hence, the reward function $r(s,a)$ and the next-state function $s'(s,a)$, which we will define as the expected next state S_t after taking action a from state s, change from stochastic functions to deterministic expressions as follows:

$$\begin{cases} r(s,a) = r_{t+1}, & s.t. \quad S_t = s, A_t = a \\ s'(s,a) = s_{t+1}, & s.t. \quad S_t = s, A_t = a \end{cases} \tag{50}$$

Now that we have presented the model we will use to assess the cascading failure risk in a power system, we will first explain in Section 6 how we will simulate these cascading failures and afterward, in Sections 7 and 8, how we will apply Dynamic Programming and the Monte-Carlo Algorithm to solve the presented MDP.

6 *CASCADING FAILURE SIMULATION*

In order to compute (50) for any $s \in \mathcal{S}$ and any $a \in \mathcal{A}(s)$ we will need to create a tool able to compute the possible cascading failures. In Algorithm 7 we present the algorithm we will use to compute cascading failures simulations for our model. In the first step of the algorithm, we initialize the state-action pair as s_t and a_t, where s_t must be a non-terminal state. And in the second and final step, we iteratively run the Newton Raphson algorithm and check every time we run it wether there are overloaded lines or not. This iteration keeps running until either no overloaded lines are found or until we reach a terminal state, in which case we return $\mathcal{T} = 1$. In the algorithm, we can see how we append the set of cascaded lines $\Delta\mathcal{C}_{t+1}$ to $\mathcal{C}_{t+1}$ and we add up to the reward $|\Delta\mathcal{C}_{t+1}|$ every time we run the iteration. This algorithm will allow us to compute the transitions between the different states of the power system and will give us the necessary information for computing the value functions for every state and every state-action pair in the MDP.

1. Initialization:
   Input: $\begin{cases} s_t = (\mathcal{D}_t, \mathcal{C}_t, \mathcal{T}_t) \\ a_t \end{cases}$ with $\mathcal{T}$ = False
   Initialize next state: $\mathcal{D}_{t+1} \leftarrow \mathcal{D}_t$, $\mathcal{C}_{t+1} \leftarrow \mathcal{C}_t$, $\mathcal{T}_{t+1} \leftarrow \mathcal{T}_t$
   $r_{t+1} \leftarrow 0$
2. Cascade simulation:
   Loop:
      Compute Newton-Raphson with lines in $\mathcal{D}'$ and $\mathcal{C}'$ switched off
      If it converges:
         $\Delta\mathcal{C}_{t+1} \leftarrow$ compute overloaded lines in the network
         $r_{t+1} \leftarrow r_{t+1} + |\Delta\mathcal{C}_{t+1}|$
         $\mathcal{D}_{t+1} \leftarrow \mathcal{D}_t \cup \Delta\mathcal{D}_{t+1}$
      If it doesn't converge or all the lines already switched off ($|\mathcal{D}_{(t+1)}| + |\mathcal{C}_{t+1}| = n_L$):
         $\mathcal{T} \leftarrow$True

   Until $\mathcal{T}$ =True or $|\Delta\mathcal{C}_{t+1}| = 0$
   Output: $\begin{cases} s_{t+1} = (\mathcal{D}_{t+1}, \mathcal{C}_{t+1}, \mathcal{T}_{t+1}) \\ r_{t+1} \end{cases}$

Algorithm 7: Cascading failure for computing $s'(s,a)$ and $r(s,a)$

7 DYNAMIC PROGRAMMING

Dynamic programming, as we stated in Section 3.6, is a model-based technique and therefore needs complete knowledge of the MDP. In presented MDP, without any historical information or any simulations of cascading failures, the model is completely unknown. In order to completely define the MDP, considering that it is a completely deterministic model, we will just need to define $r(s,a)$ and $s'(s,a)$ for every $s \in \mathcal{S}$ and every $a \in \mathcal{A}(s)$. Therefore, to be able to apply any technique presented in Section 3.6, we will first need to learn these two functions. To this end, we will use an exhaustive search technique called breadth-first search to completely learn the model.

7.1 EXHAUSTIVE SEARCH

If we exhaustively compute all the possible state-action in the MDP, we would be able to completely define it and therefore apply Dynamic Programming techniques. There are different ways to compute every $s \in \mathcal{S}$ and every $a \in \mathcal{A}(s)$, in this project, we will apply an exhaustive search algorithm known as breadth-first search.

This search technique iteratively computes all the state-action pairs in $s \in \mathcal{S}(t)$ and $a \in \mathcal{A}(s)$ for $t = 0,1,2,\ldots$ until all the non-terminal state actions are computed. Therefore, it computes every state-action pair in the MDP sorted by time-step t. As we show in Algorithm 8, it uses a queue of states for sorting them properly. This queue applies the method first-in-first-out. It picks the initial state in the queue to compute all its possible next states and append these at the end of the queue as they are being visited. Also, iteratively removes the states in which all the actions have been computed until this queue is empty.

As we can see in Algorithm 8, this technique allows us to define $s'(s,a)$ and $r(s,a)$ for every $s \in \mathcal{S}$ and every $a \in \mathcal{A}(s)$ and therefore gives us the complete knowledge of the model needed for using Dynamic Programming.

$Queue =$ list with s_0 as the single component
Loop:
    $s =$ first element in $Queue$
    For $a \in \mathcal{A}(s)$:
        $s'(s,a) \leftarrow$ compute state s' with cascading failure simulation for s and a
        $r'(s,a) \leftarrow$ compute reward with cascading failure simulation for s and a
        If s' is not terminal: append s' to final position in $Queue$
    Delete s from $Queue$
Until $Queue$ is empty

Algorithm 8: Breadth-First Search

7.2 BACKWARD INDUCTION

Once we have the model completely defined, we can compute any algorithm within Dynamic Programming we have presented. In this project, we will apply the backward induction algorithm to get the optimal solution Q^* and π^* and use it to evaluate how well the Monte-Carlo converges to this solution. This algorithm has the drawback of not being very time-efficient and therefore may become infeasible over large state-spaces. Nevertheless, it also presents the advantage of returning the optimal solution, when obtained, at once, and thus we will use it in this thesis when possible.

It is intuitive to realize that the more cascading failures that happen in a system, the fewer states we have to analyze in our model. Therefore, if we assume a MDP where no cascading failures occur at all, we could get the maximum number of states in a power system by simply using permutations. If we denote as $\boldsymbol{P}_k^n$ to the k-permutations without repetition of n elements, we can add up all the k-permutations of n_L lines for $k = 1, \dots, n_L$, and compute the maximum number of states in a MDP as follows:

$$\max|\mathcal{S}^+| = \sum_{i=1}^{n_L} \boldsymbol{P}_k^{n_L} \tag{51}$$

$$= \sum_{i=1}^{n_L} \frac{n_L!}{(n_L - k)!} \tag{52}$$

In order to have a general idea of the maximum size of the state-space depending on the number of lines in a system, we show some evaluations of (51) in Table 3. The factorial function increases faster than all the polynomial and exponential functions and that is why the maximum state-space size increases so rapidly. As we can see, even just with 25 lines we get a state-space with a maximum size of $\approx 4.2 \cdot 10^{25}$ which may become infeasible within reasonable time horizons for many usual computers. As we will see in Part V, this will restrict us from using Dynamic Programming and force us to use iterative methods such as the Monte-Carlo algorithm for medium-size power systems.

Number of lines	Maximum number of states
5	326
10	$\approx 9.9 \cdot 10^{6}$
25	$\approx 4.2 \cdot 10^{25}$
50	$\approx 8.3 \cdot 10^{64}$
100	$\approx 2.5 \cdot 10^{158}$

Table 3: Maximum number of states per number of lines in a power system

In this project, backward induction will allow us to compute Q^*, V^* and π^* for every state and every state-action pair in our MDP. By computing the action values of the initial states $Q(s_0, a) \quad \forall a \in \mathcal{A}(s_0)$ we will be able to asses the cascading risk for every line contingency. Also, $V(s_0)$ will allow us to assess the risk of cascading failure for the system configuration s_0. For our MDP, we will compute the action values and use them afterwards to compute the state values when needed. As we show in case i in Figure 17, we will store the action values in a hash table with two entries: the states and the actions.

In the presented model, the state-space can become so big as the number of lines increases that using a hash table may become a major challenge for memory. As we present in Figure 17 in case ii, neural networks can be used as an alternative approach to hash tables. While they compute an approximation of Q, they present the advantages of only storing its parameters and of being able to compute the action values for infinite state-spaces. In this project, we will focus on obtaining the exact value function and therefore we will use hash tables instead of neural networks. Nevertheless, the use of neural networks is encouraged in future versions of this project.

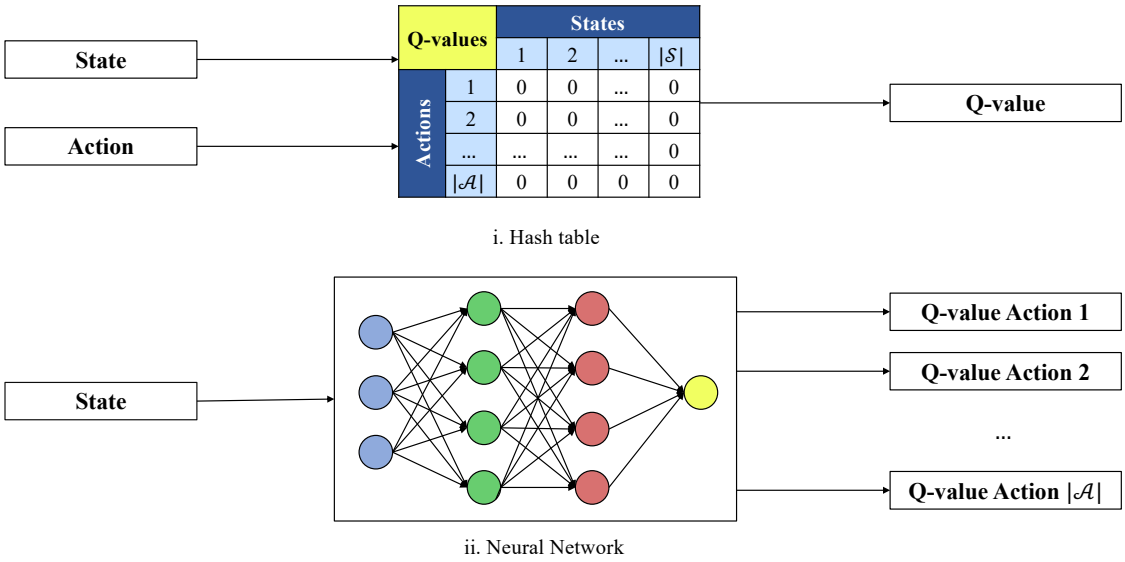

Figure 17: Action value function in RL (case i) and action value function in deep RL (case ii)

8 *MONTE-CARLO ALGORITHM*

Dynamic Programming, as we have stated in the previous section, works under the assumption that the model is completely known. This assumption is not initially true for our MDP and achieving it through exhaustive search can easily become an infeasible option as the number of lines increase in a power system. For this reason, model-free techniques are needed to deal with these large state-spaces. The most well-known methods within model-free techniques are the Monte-Carlo method and Temporal Difference learning. In this project, we will apply the Monte-Carlo Method to initially apply RL to power systems. Nevertheless, we encourage testing the implementation of Temporal Difference learning in future versions of this project.

By using this method, we will compute episodes E using an ϵ-greedy policy as the one we presented in the example in Figure 11. These episodes, in contrast to Dynamic Programming, do not need to traverse the complete MDP diagram and can just compute sections of it. Applying the Monte-Carlo method to power systems presents the advantage of being able to use any episode both from historical data and from simulation to iteratively converge to the optimal solution. This presents a key opportunity for the application of the Monte-Carlo method to power systems.

This method starts from an initial value function Q_0 and an initial ϵ-greedy policy π_0 and, just as we explained in Figure reference 10, iteratively improves them with episodes using policy evaluation and policy improvement. This technique is guaranteed to converge to the optimal ϵ-greedy policy π^* and to its corresponding optimal action value function Q^* as the number of episodes grows to infinity. This method offers the advantage of being able to assess an approximation of the presented cascading failure risk indexes at any time. This possibility of generating data on-demand allows grid operators to consider this tool and quickly respond accordingly to security threats. The solution given may be suboptimal but, considering the infeasibility to compute exact solutions in reasonable time, this technique, among other model-free techniques, becomes the only option when dealing with large state-spaces.

Additionally, the Monte-Carlo method presents the advantage of being able to use previous information about the vulnerable line contingencies for the initialization of Q_0 and π_0. The possibility to choose this initiazation allows us to consider information from other power system security analysis, such as the $N-k$ analysis, to focus on the parts we believe to be more relevant in the MDP. In this project, we will use a random initialization for both the action value function and the policy in order to test the complete and purest version of the Monte-Carlo method. Nevertheless, it is important to realize that the possibility of using previous information makes this method especially appealing when applying it to power system security analysis.

Finally, as we can see, this method offers us a tool to efficiently deal with massive state-spaces. Nevertheless, a possibility to reduce significantly the state space could be to reduce the depth of the MDP by defining a maximum time-step $T_{\max}$ and therefore defining the states $s \in \mathcal{S}(T_{\max})$ as terminal. As we can see in Figure 16, when $\gamma < 1$ the discount factor decreases as the time-step t increases. A possibility when implementing the Monte-Carlo method could be to ignore the cascade rewards which have a discount factor less than a specific value. For example, in the presented Figure 16, we could establish to ignore discount factors less than 0.2 and, therefore, assign a 0 discounting factor to time-steps $t > 16$. In this project we will not use this approximation as we will test the complete version of Monte-Carlo, nevertheless we encourage the reader to consider this as a reasonable approach when dealing with large state-spaces.

Part IV. SOFTWARE IMPLEMENTATION

Once we have explained the necessary theoretical background about RL and power systems in Part II and the model we will use in this thesis in Part III, we will move on explaining the software we will use to build our security analysis tool. As we can see in Figure 18, we will implement our framework using Python, where we will use a special power system module called Pandapower, and we will create two programs: one for Dynamic Programming (`Dynamic_programming.py`) and another one for the Monte Carlo method (`Monte_Carlo.py`).

In order to focus on the theory underlying this project, we will retain from explaining in detail the structure of these two programs and we will content ourselves with the pseudocodes presented in previous parts explaining both techniques. Nevertheless, we will include these programs in the Appendix part of this thesis (pages 91 to 102) and we will compute them in Part V. In this part, we will first explain the basics about Python in Section 9 and then we will explain how the Pandapower module works in Section 10.

Figure 18: Software environment framework

9 PYTHON

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics [13]. It is also an open-source programming language which means that it is freely available. It is becoming increasingly popular and today is ranked as the second most used programming language in the world after JavaScript [10]. According to SlashData, as we can see in Figure 19, there are now 8.2 million developers in the world who code using Python and that population is now larger than those who build in Java, who number 7.6 million [10]. Also, it is ranked as the top programming language by the world's largest technical professional organization: IEEE [11]. This popularity, accomplished due to its advantages over other programming languages, facilitates the extension and implementation of this project for further investigation. The main reasons why this project is implemented in Python are three: its simplicity, its large developer community and its popularity within ML.

- **Simplicity**: Python is powerful because it is very easy and this is a benefit both for beginners like me and for experienced programmers. Its simplicity and improved code readability, in comparison to other programs I am familiarized with like Matlab or C, have allowed me to create very complex programs easily and to waste less time with confusing syntax and gain time thinking about the program itself. This code readability translates into better and faster debugging of the program. These factors have made Python the most popular introductory teaching programming language among the top US universities.

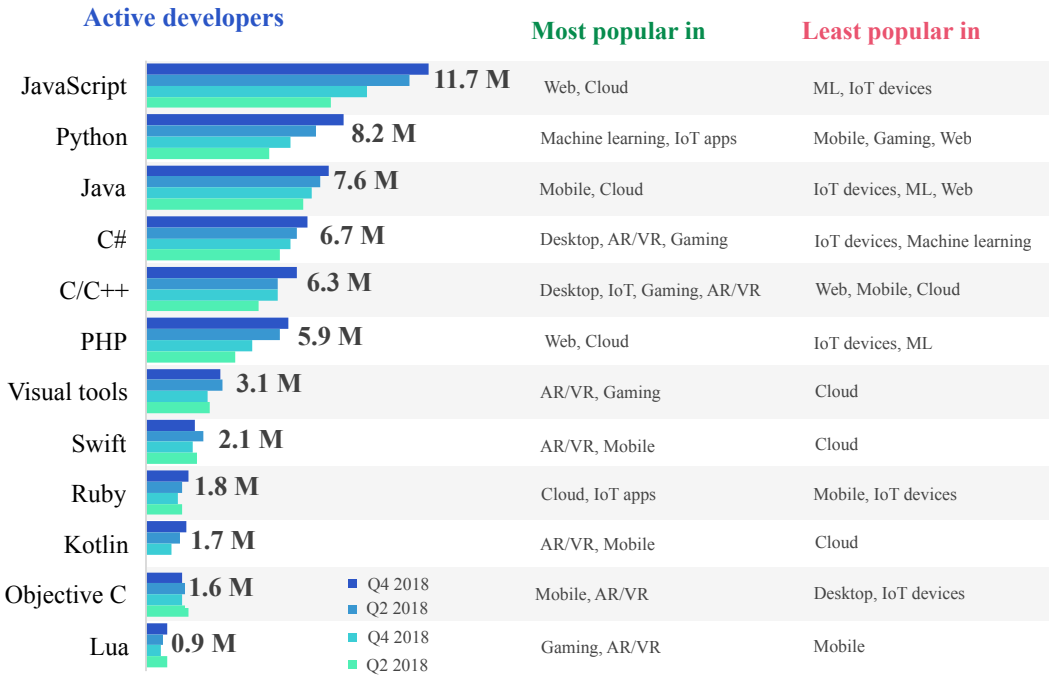

Figure 19: Number of active software developers globally in millions in 2017 and 2018 obtained from SlashData

- **Large developer community**: since it has enjoyed wide dissemination and acceptance across many disciplines, it now has a huge developer community (see Figure 19). There is an abundance of open-source Python libraries available that can most likely achieve what you need to help you solve your problem. In the case of this project where we combine techniques from different fields, there were many modules available for both RL and power systems. Within power systems we found PyPSA, which stands for Python applied to Power System Analysis; PYPOWER, which is a port to the Matlab package MATPOWER; and Pandapower. This last one is the one we will use for this project.

- **ML applications**: when it comes to AI programming languages Python leads the pack with its unparalleled community support and pre-built libraries: NumPy, Pybrain, SciPy, Pandas and much more. In this project, we will use the Pandas library to compute the corresponding hash tables as we will see in Part V. Its flexibility and predefined libraries and frameworks allow many AI and ML projects to develop. Among the top companies within the field, we see Google and Facebook, which both have created their libraries for this type of application: TensorFlow and PyTorch. Python is becoming so powerful and used that Google has created a free cloud service with free GPU for running computationally expensive Python codes: Google Colab.

10 PANDAPOWER

Pandapower is an easy to use open-source tool for power system modeling, analysis and optimization with a high degree of automation [8]. It works through the Python data analysis library Pandas and the power system analysis toolbox PYPOWER. It started as an instrument around PYPOWER and has evolved into an independent power systems analysis toolbox with an extensive power system model library, an improved power flow solver and many other power system analysis functions [1].

Pandapower was developed to close the gap between commercial power systems like Sincal, PowerFactory or Nepal and open-source power systems analysis tools like MATPOWER and PYPOWER. As we can see in Table 39, Pandapower combines the advantages of commercial tools with the ones of open-source tools [1].

	Electric Models	**Automation**	**Customization**
Commercial tools	✓ Throughly validated and easy to parametrize electric models	✕ Graphical user interface applications difficult to automate.	✕ Restricted possibilities for customization due to propietary codebase.
Open-source tools	✕ Basic models that require expert knowledge for parametrization	✓ Console application designed for automated evaluations.	✓ Free modifiable and customizable code
Pandapower	✓ Throughly validated and easy to parametrize electric models	✓ Console application designed for automated evaluations.	✓ Free modifiable and customizable code

Table 4: Comparison between Pandapower and commercial and open-source tools

Pandapower is all written in Python and, therefore, it offers all the possibilities and advantages this language presents. In the following Sections 10.1 and 10.2 we will show some code computation of Pandapower in order to explain how the elements that make up a power system are modeled and how its power flow solver works.

10.1 NETWORK ELEMENTS

As we have previously stated, Python is an object-oriented language and Pandapower, within this programming language, can be understood as an "object", with its functionalities and its information which can be accessed and modified just as "physical objects". In Python, we call these "objects" modules and they contain functions, classes and variables.

Within Pandapower we will use two modules, one is called pandapower and the other one is called pandapower.networks which is a submodule of the first one. In the `pandapower.networks` module there are grids completely modeled such as CIGRE test grids or IEEE case files. Also, the pandapower module offers the possibility to create and model your own network. Just as we did with `pandapower.networks` if we want to access a module we will just type the name of the module followed by a dot and the name of the class, variable or function we want to access.

In order to show an example and explain the structure of a network within Pandapower we will import the two mentioned modules, which we will rename as pp and nw for code readability, and we will access to a function `case89pegase()` which will return an 89-bus Pandapower network class:

```
>>> import pandapower as pp
>>> import pandapower.networks as nw
>>> net = nw.case89pegase()
>>> net.fn
50
>>> net.sn_mva
1
>>> net
This pandapower network includes the following parameter tables:
- bus (89 elements)
- load (29 elements)
- sgen (6 elements)
- gen (11 elements)
- shunt (44 elements)
- ext_grid (1 element)
- line (160 elements)
- trafo (50 elements)
- poly_cost (12 elements)
- bus_geodata (89 elements)
```

In Python, classes are a type of "object" which can be defined in a code or can be imported from a module, as shown in this example. In this computed simple code `nw.case89pegase()` is a class "object" and net is an individual object of this class, which is known in Python programming language as an instance of the class. In the ran code, we can see that net is made up of a variable for the frequency (`fn`), a variable for the nominal power of the system in MVA (`sn_mva`) and a function that prints the information about the classes we can find within `net`.

As shown in Figure 20, we can see that the structure of a Pandapower network consists of 4 parts: the main parameters defining the network which are its name, its frequency and its nominal power; a type dictionary for modeling networks and two sets of tables: one for storing the information implicit to the system (element tables) and the other one for storing the results after a power flow calculation (result tables) [7].

Within the element tables, we will focus on the following in this project: `bus`, `line`, `load`, `sgen`, `gen` and `ext_grid`. Other tables for elements like: transformers, shunt elements, switches, impedances and some more may be within a Pandapower network and could appear in our model. Nevertheless, because the goal of this thesis is not the technical details of power systems but to present RL applied to power security analysis we will not present these tables. Also, within the result tables, we will present the tables `res_bus` and `res_line`.

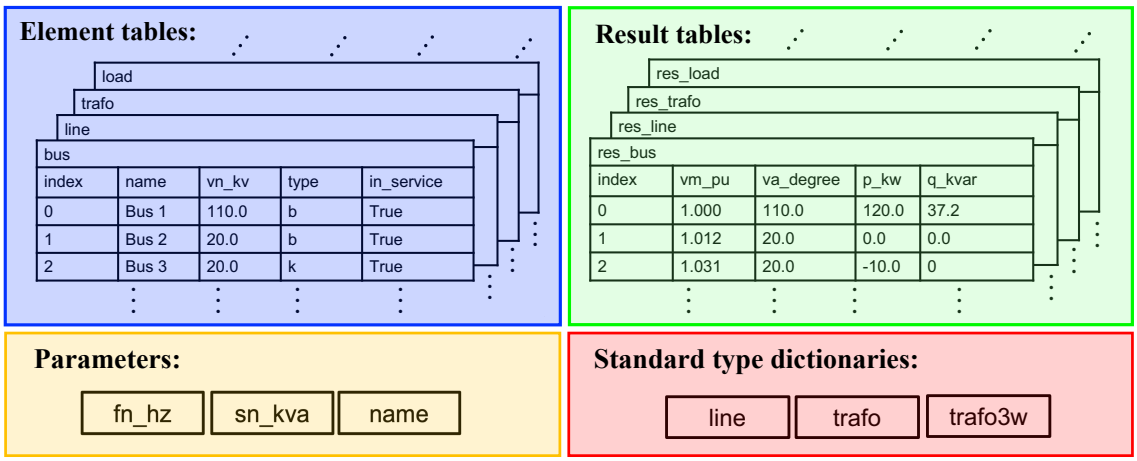

Figure 20: Dataframe structure of network class in Pandapower

If we continue with the previous code, we can compute the mentioned tables present in net to see their parameters and analyze the ones we will use for our model. In the computed tables, we will indicate with an asterisk the variables used for the power flow solution and we will show the numerical values rounded with a maximum of three decimals.

```
>>> net.bus
     in_service*  max_vm_pu  min_vm_pu   name   type   vn_kv*   zone
0          1         1.1        0.9       88      b      380      0
1          1         1.1        0.9      227      b      150      0
2          1         1.1        0.9      270      b      150      0
...      ...         ...        ...      ...    ...      ...    ...
88         1         1.1        0.9     9238      b      380      0
```

- **Bus**: in this table, we will use the boolean variable indicating whether the buses are in service or not and the nominal voltage for each bus for the power flow equations. These nominal voltages, combined with the nominal power of the system, `sn_mva`, will allow us to transform different variables from unit quantities to per-unit quantities and vice versa. Additionally, the table gives us information about the type of each node (n: node, b: busbar, m: muff), the zone it belongs, a string name for identifying each node and the upper and lower-bound voltages for each node. As we stated in Section 2.2, in this project we won't consider these limits.

```
>>> net.line
     from_bus*  to_bus*  r_ohm_per_km*  x_ohm_per_km*  c_nf_per_km*  g_us_per_km*
0       31         4         1.054          13.07           0             0
1       84        43         2.772          33.15           0             0
2       18        57         6.926          28.55           0             0
...    ...       ...          ...            ...           ...           ...
159     15        77           0             0.05           0             0

 length_km*  parallel*  in_service*  max_i_ka*  df*  max_loading_percent  ...
      1          1           1          1.830     1          100.0          ...
      1          1           1          1.381     1          100.0          ...
      1          1           1          99999     1          100.0          ...
     ...        ...         ...           ...    ...          ...          ...
      1          1           1          2.179     1          100.0          ...
```

- **Line**: this table shows the necessary information about the impedance parameters per unit length and the length of the lines so that Pandapower can calculate the admittance matrix ***Y*** presented in (28) by considering (53). Also, this table gives us information about the maximum thermal limits for each line just as we showed in Section 2.1. The derating factor, `df` $\in (0,1]$, is the maximal current of each line in relation to its nominal current. This parameter, along with the number of parallel lines (`parallel`) and the maximal thermal current in kA (`max_i_ka`), will allow us to compute the loading percentage for each line after computing the power flow equations. This result, compared with the parameter `max_loading_percent`, will allow us to compute which lines are overloaded after every line contingency and, therefore, simulate the possible cascading failures in this project.

$$\begin{cases} \bar{Z}_{ij} = (\texttt{r\_ohm\_per\_jm} + j\cdot \texttt{x\_ohm\_per\_km})\cdot \texttt{length\_km}/\texttt{parallel} \\ \bar{Y}_{ij} = (\texttt{g\_us\_per\_km}\cdot 10^{-6} + j\cdot 2\pi f\cdot \texttt{c\_nf\_per\_km}\cdot 10^{-9})\cdot \texttt{length\_km}/\texttt{parallel} \end{cases} \quad (53)$$

```
>>> net.load
    bus  ct_i_percent  ct_z_percent  in_service  p_mw  q_mvar  scaling  ...
     *         *             *            *         *      *        *     .
0    2         0             0            1      96.7   26.8        1  ...
1    5         0             0            1      295.3  41.9        1  ...
2    7         0             0            1      244.2  152.7       1  ...
..  ...       ...           ...          ...     ...    ...      ...  ...
28   87        0             0            1      17.92  23.3        1  ...
```

- **Load (PQ bus)**: this table shows the information about the PQ nodes and allows us to define the active and reactive power demand for the power flow equations as shown in (54) and (55). As we can see, these loads do not show a fixed active and reactive power demand, as stated in Section 4.2, but a more complex expression known as the ZIP model. These loads are represented by a composition of a constant impedance (Z), a constant current (I) and a constant power (P). One can easily account for this ZIP model in the power flow equations by taking some minor adjustments over the $F(x)$ and X presented in Section 4.2. Nevertheless, to avoid getting lost in these finer details we will not explain them in this project.

$$\begin{cases} P_{load} = \texttt{p\_mw}\cdot\texttt{scaling}\cdot(p_{const} + z_{const}\cdot V^2 + i_{const}\cdot V) \\ Q_{load} = \texttt{q\_mw}\cdot\texttt{scaling}\cdot(p_{const} + z_{const}\cdot V^2 + i_{const}\cdot V) \end{cases} \quad (54)$$

$$\begin{cases} z_{const}=ct\_z\_percent/100 \\ i_{const}=ct\_i\_percent/100 \\ p_{const}=(1-z_{const}-i_{const}). \end{cases} \quad (55)$$

```
>>> net.sgen
              bus*    in_service*    p_mw*     q_mvar*    scaling*      ...
    0            1              1    23.43      -57.4           1      ...
    1           21              1   357.45      -33.05          1      ...
    2           54              1   456.66      131.97          1      ...
   ...         ...            ...      ...         ...         ...     ...
    5           79              1  1299.13      140.85          1      ...
```

- **Static generator (PQ bus)**: these elements are used to model generators with a constant active and reactive power feed-in. They can be understood as negative PQ buses where P_i and Q_i are known:

$$\begin{cases} P_i=p\_mw\cdot scaling \\ Q_i=q\_mvar\cdot scaling \end{cases} \quad (56)$$

```
>>> net.gen
              bus*    in_service*    p_mw*    scaling*    vm_pu*      ...
    0           20              1   1269.4           1     1.039      ...
    1           23              1      362           1     1.039      ...
    2           36              1   1097.4           1     1.052      ...
   ...         ...            ...      ...         ...       ...     ...
   10           88              1      419           1     1.052      ...
```

- **Generator (PV bus)**: these are the generators modeled as voltage-controlled PV nodes with the power in kW and the voltage in p.u. specified as shown in (57). From now on, we will use lowercase notation to denote the variables in per-unit quantities.

$$\begin{cases} P_i=p\_mw\cdot scaling \\ v_i=vm\_pu \end{cases} \quad (57)$$

```
>>> net.ext_grid
              bus*     in_service*  va_degree*    vm_pu*       ...
    0           6           1           0         1.031        ...
```

- **External grid (slack bus):** this element represents the slack bus which serves as the angle reference for the rest of the nodes and accounts for the power losses of the system in the power flow equations. Its per-unit voltage, v_i, and its voltage argument, θ_i, are defined as:

$$\begin{cases} v_i = \texttt{vm\_pu} \\ \theta = \texttt{va\_degree} \end{cases} \tag{58}$$

10.2 POWER FLOW SOLVER

Once we have seen the main parts of a network in Pandapower and we have indicated the important variables in these tables for the power flow equations, we will see how the Pandapower power flow solver works.

Apart from the simplicity and intuitiveness behind Pandapower, another main reason why we have decided to use this module for this project is its improved speed when it comes to computing the Newton-Raphson method for solving the power flow equations. This algorithm, as seen in Section 4.3, iteratively computes the Jacobian matrix of the power flow equations and therefore, in order to be time-efficient, requires good performance derivating. As we can see in Figure 21, Pandapower is considerably more time-efficient than other power flow solvers like Matpower or Pypower when using the Newton-Raphson method [23].

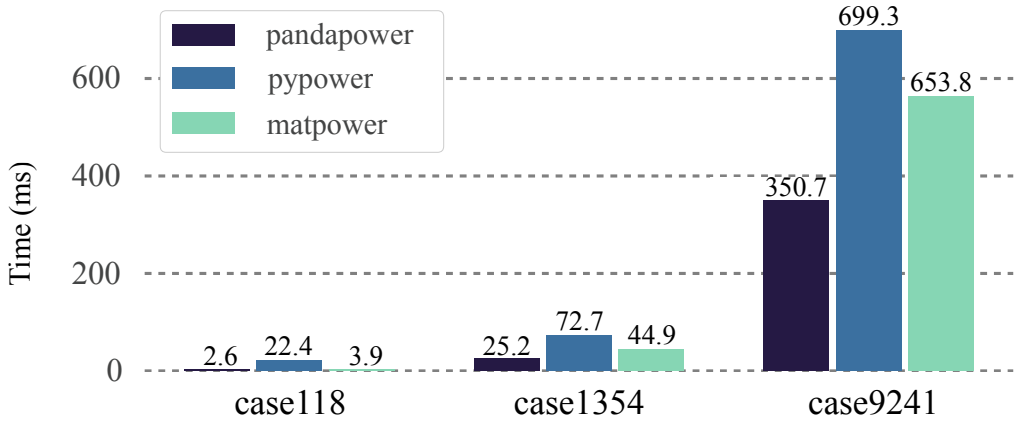

Figure 21: Computational time of the Newton-Raphson solver in total for three different power systems

The power flow solver in Pandapower is a function called runpp within the module pandapower. This function uses as input a Pandapower network and reconfigures it to append or change the results tables in this network as shown in Figure 20. We can compute a Newton-Raphson power flow calculation in Pandapower as:

```
>>> pp.runpp(net, algorithm="nr", init="dc", max_iteration=10,
tolerance_mva=1e-8)
This pandapower network includes the following parameter tables:
- bus (89 elements)
- load (29 elements)
- sgen (6 elements)
- gen (11 elements)
- shunt (44 elements)
- ext_grid (1 element)
- line (160 elements)
- trafo (50 elements)
- poly_cost (12 elements)
- bus_geodata (89 elements)
and the following results tables:
- res_bus (89 elements)
- res_line (160 elements)
- res_trafo (50 elements)
- res_ext_grid (1 element)
- res_load (29 elements)
- res_sgen (6 elements)
- res_shunt (44 elements)
- res_gen (11 elements)
```

As we can see in this execution, we can configure different arguments in the `runpp` function for solving the power flow equations. The executed line really would have been the same as executing `pp.runpp(net)` because in the shown code we have set the default arguments. Nevertheless, for illustrating the possibilities of this solver, we have shown four of its possible arguments. With Pandapower, apart from choosing different algorithms for solving the power flow equations, we can easily customize the initialization for $X^{(0)}$(`init`), the maximum number of iterations N(`max_iteration`), and the maximum mismatch tolerance ϵ (`tolerance_mva`) among other possibilities. If we show the computed results in `res_bus` we can see the computed v_i, θ_i, P_i and Q_i after iteratively solving the power flow equations as:

```
>>> net.res_bus
      vm_pu   va_degree    p_mw    q_mvar
0     0.999      -2.728   0.199    -4.518
1     1.052      -6.262  -23.430   57.400
2     1.039       4.860  149.300   -24.404
...     ...         ...      ...       ...
88    1.052       7.895 -419.000  -154.105
```

Also, by applying equation (29) and with the admittance matrix $\boldsymbol{Y}$ and the voltajes $\bar{V}$, Pandapower can compute the current flowing through every line as shown in table `res_line`. In this table, we can also see the active and reactive power calculated from every bus in every line obtained by applying the Kirchhoff's Current Law and the active and reactive power losses (`pl_mw` and `ql_mvar`) for every line.

```
>>> net.res_line
     p_from_mw  q_from_mvar  p_to_mw  q_to_mvar  pl_mw  ql_mvar  i_from_ka
0     -361.910       6.667  362.780      4.114  0.870   10.781      0.524
1     -250.634      49.821  251.755    -36.422  1.120   13.399      0.367
2       73.686     -38.262  -72.804     41.897  0.882    3.634      0.206
88     -35.019      14.695   37.093     -8.186  2.074    6.509      0.141

 i_to_ka  i_ka  vm_from_pu  va_from_degree  vm_to_pu  va_to_degree  loading_percent
   0.524 0.524       1.049          6.053     1.051         7.758           28.644
   0.367 0.367       1.058          3.598     1.053         6.608           26.576
   0.206 0.206       1.058          1.930     1.070        -0.548            0.000
   0.141 0.141       1.034        -11.211     1.034        -0.892            0.000
```

In this project, the most important information after we have solved the power flow equations will be the loading percentage calculated for every line as we show in (59). When comparing this result with `max_loading_percent` in `net.line` we will be able to see the overloaded lines and change their corresponding parameter `in_service` in `net.line` from true (`1`) to false (`0`). This way we will be able to compute the cascaded simulations as we stated in Section 6 and create out RL framework for assessing the risk of cascading failure.

$$\texttt{loading\_percent} = \frac{\max(\texttt{i\_from}, \texttt{i\_to})}{\texttt{i\_max\_ka} \cdot \texttt{df} \cdot \texttt{parallel}} \cdot 100 \qquad (59)$$

Part V. NUMERICAL RESULTS

Finally, once we have completely explained the model and the software we will use for assessing the risk of cascading failures within a power system, we will present some numerical results for both Dynamic Programming and the Monte-Carlo method. We will test four different IEEE cases extracted from `pandapower.networks`. We will initially test two small-size cases: `case4gs` and `case9`, and then, to see how these methods scale to medium-size cases we will test `case39` and `case89pegase` [9]. We will apply to each of them the programs of `Dynamic_programming.py` and `Monte_Carlo.py` which we have included in the Appendix part of this thesis (pages 91 to 102).

In order to have a general idea of the state-spaces of these cases, we show in Table 5 the total number of lines of each case, n_L, and the maximum possible size of the state-space for each case applying equation (52).

Network	n_L	Maximum number of states
`case4gs`	4	65
`case9`	9	$\approx 9.9 \cdot 10^5$
`case39`	35	$\approx 2.8 \cdot 10^{40}$
`case89pegase`	160	$\approx 1.3 \cdot 10^{285}$

Table 5: Maximum number of states for each network

11 SMALL-SIZE POWER SYSTEM EXAMPLES

11.1 4-BUS POWER SYSTEM

In order to illustrate in detail the presented tool for assessing the risk of cascading failure in power systems, we will start with the simplest possible case: the IEEE network 4-bus network `case4gs`. This power system was imported by Pandapower from PYPOWER and was originally presented in the book *Power System Analysis* (J. J. Grainger and W. D. Stevenson, 1994).

If we apply the exhaustive breadth-first search algorithm within `Dynamic_Programming.py` to this network, in just a few seconds, we can obtain the functions $r(s,a)$ and $s'(s,a)$ and, therefore, have the MDP model completely defined. By doing so, we have observed a total state-space size of 35, which is about half of the maximum possible size we presented in Table 5. Because this state-space is small enough,

we have been able to plot the complete MDP diagram in Figure 22. In this diagram, we can clearly distinguish the terminal states in blue from the non-terminal states in green. As we can see, all the terminal states verify that $|\mathcal{D}| + |\mathcal{C}| = n_L$, and therefore we can conclude that all the cases have converged when using the Newton-Raphson method. Also, in the diagram, we can see in yellow the five only possible rewards achievable in this power system, in the rest of the state-action pairs, we can see no cascading failure propagation at all.

Nevertheless, the program `Dynamic_programming.py` does not compute the rewards and transitions as the diagram shown in Figure 22, but as a hash table which we can see in Table 6. For simplicity, this table shows two hash tables combined together: one for the next-state function $s'(s,a)$ and another one for the reward function $r(s,a)$. This way, if we are interested in checking the state configuration of the 4-bus power system after disconnecting any line a from any state s, we can check the next configuration of the system and the number of cascaded lines by going to the corresponding cell in row s and column a. As we can see, some of the cells in the table are empty. This is because we can't disconnect lines already inoperative in a system and, therefore, $\mathcal{A}(s) \neq \mathcal{A} \quad \forall s \in \mathcal{S}, s \neq s_0$.

As we show, all the states are defined using the 3-tuple $(\mathcal{D}, \mathcal{C}, \mathcal{T})$ we presented in (43) where $\mathcal{D}$ denoted the disconnected lines by the agent, $\mathcal{C}$ the cascaded lines after each action and $\mathcal{T}$ the boolean variable indicating whether a state is terminal (F) or not (T). In this section, when computing the two programs `Dynamic_programming.py` and `Monte_Carlo.py`, we will express the actions as tuples of single elements containing the index of the line to disconnect. In this example, therefore the state space would be expressed as $\mathcal{A} = \{(0), (1), (2), (3)\}$.

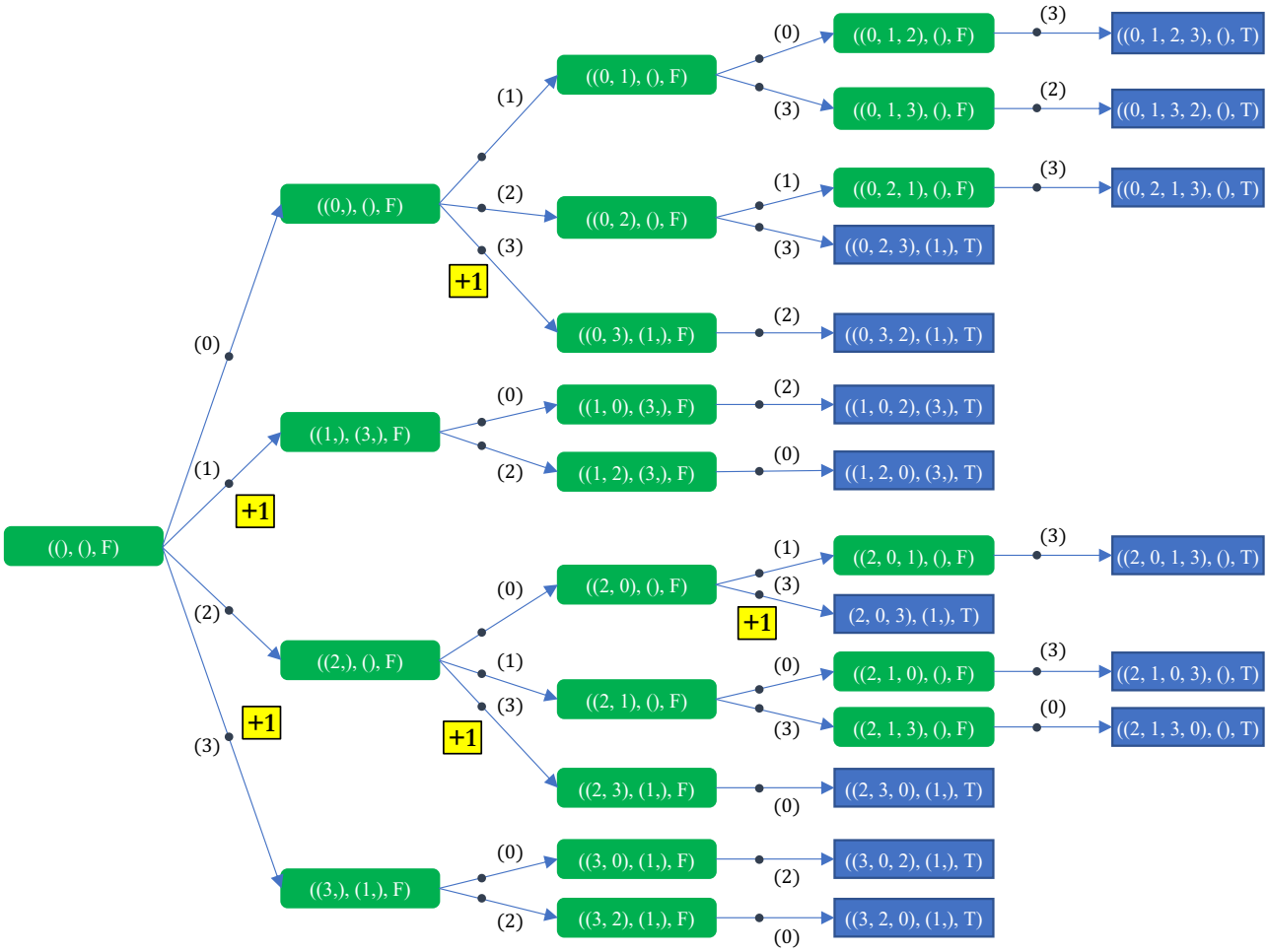

Figure 22: Markov Decision Process diagram for Case4gs

State	Action: (0)		Action: (1)		Action: (2)		Action: (3)	
((), (), False)	((0,), (), False)	0	((1,), (3,), False)	1	((2,), (), False)	0	((3,), (1,), False)	1
((0,), (), False)			((0, 1), (), False)	0	((0, 2), (), False)	0	((0, 3), (1,), False)	1
((1,), (3,), False)	((1, 0), (3,), False)	0			((1, 2), (3,), False)	0		
((2,), (), False)	((2, 0), (), False)	0	((2, 1), (), False)	0			((2, 3), (1,), False)	1
((3,), (1,), False)	((3, 0), (1,), False)	0			((3, 2), (1,), False)	0		
((0, 1), (), False)					((0, 1, 2), (), False)	0	((0, 1, 3), (), False)	0
((0, 2), (), False)			((0, 2, 1), (), False)	0			((0, 2, 3), (1,), True)	1
((0, 3), (1,), False)					((0, 3, 2), (1,), True)	0		
((1, 0), (3,), False)					((1, 0, 2), (3,), True)	0		
((1, 2), (3,), False)	((1, 2, 0), (3,), True)	0						
((2, 0), (), False)			((2, 0, 1), (), False)	0			((2, 0, 3), (1,), True)	1
((2, 1), (), False)	((2, 1, 0), (), False)	0					((2, 1, 3), (), False)	0
((2, 3), (1,), False)	((2, 3, 0), (1,), True)	0						
((3, 0), (1,), False)					((3, 0, 2), (1,), True)	0		
((3, 2), (1,), False)	((3, 2, 0), (1,), True)	0						
((0, 1, 2), (), False)							((0, 1, 2, 3), (), True)	0
((0, 1, 3), (), False)					((0, 1, 3, 2), (), True)	0		
((0, 2, 1), (), False)							((0, 2, 1, 3), (), True)	0
((2, 0, 1), (), False)							((2, 0, 1, 3), (), True)	0
((2, 1, 0), (), False)							((2, 1, 0, 3), (), True)	0
((2, 1, 3), (), False)	((2, 1, 3, 0), (), True)	0						

Table 6: Bread-first search results of $s'(s,a)$ and $r(s,a)$ for Case4gs

Once we have computed the breadth-first search algorithm, we have the MDP completely defined and, therefore, we can apply backward induction to compute Q^* and π^*. In Table 7, we can see these two hash tables for this power system after using a discount rate of $\gamma = 0.9$. The $Q^*(s, a)$ hash table containing the risk associated with each line contingency for every possible state configuration of the system. We can appreciate that at the initial state $s_0 = ((), (), F)$, the most vulnerable lines to the system are lines 1 and 3, with a risk index of 1. Also, we can see that lines 0 and 2 have a risk index of 0.9. This way, by computing expression (60) we can compute a risk index of value 1 to the initial state.

$$V(s_0) = \max_{a \in \mathcal{A}(s_0)} Q(s_0, a) \tag{60}$$

Despite the possibility to use Dynamic Programming in this example, if we apply the Monte-Carlo method presented in Algorithm 5 we can test how this algorithm convergences to the optimal solution. In Figure 23, we show ten different executions of the Monte-Carlo method using $\epsilon = 0.2$, $\gamma = 0.9$ and a total of 50 episodes for each execution. In this figure, we can appreciate how the risk assessment of the initial state of the system converges to the optimal index of 1 for all the executions.

In Table 8, we can compare the risk index of cascading failure associated with each line contingency in the initial state computed by Dynamic Programming with the one obtained using the Monte Carlo method. In this table, Q^* represents the optimal solution computed by Dynamic Programming and $q_{\epsilon,500}$ represents the average return computed by the Monte-Carlo method when improving an ϵ-greedy policy with $\epsilon = 0.2$ after 500 episodes. As we can see, the indexes are the same for lines 1 and 3 but differ by about 0.4 for lines 0 and 2. Nevertheless, this approximation can perform well when sorting the lines from the most vulnerable ones to the most secure ones.

State: s	$\pi^*(s)$
((), (), False)	(1,)
((0,), (), False)	(3,)
((1,), (3,), False)	(0,)
((2,), (), False)	(3,)
((3,), (1,), False)	(0,)
((0, 1), (), False)	(2,)
((0, 2), (), False)	(3,)
((0, 3), (1,), False)	(2,)
((1, 0), (3,), False)	(2,)
((1, 2), (3,), False)	(0,)
((2, 0), (), False)	(3,)
((2, 1), (), False)	(0,)
((2, 3), (1,), False)	(0,)
((3, 0), (1,), False)	(2,)
((3, 2), (1,), False)	(0,)
((0, 1, 2), (), False)	(3,)
((0, 1, 3), (), False)	(2,)
((0, 2, 1), (), False)	(3,)
((2, 0, 1), (), False)	(3,)
((2, 1, 0), (), False)	(3,)
((2, 1, 3), (), False)	(0,)

State\Action	(0)	(1)	(2)	(3)
((), (), False)	0,9	1	0,9	1
((0,), (), False)		0	0,9	1
((1,), (3,), False)	0		0	
((2,), (), False)	0,9	0		1
((3,), (1,), False)	0		0	
((0, 1), (), False)			0	0
((0, 2), (), False)		0		1
((0, 3), (1,), False)			0	
((1, 0), (3,), False)			0	
((1, 2), (3,), False)	0			
((2, 0), (), False)		0		1
((2, 1), (), False)	0			0
((2, 3), (1,), False)	0			
((3, 0), (1,), False)			0	
((3, 2), (1,), False)	0			
((0, 1, 2), (), False)				0
((0, 1, 3), (), False)			0	
((0, 2, 1), (), False)				0
((2, 0, 1), (), False)				0
((2, 1, 0), (), False)				0
((2, 1, 3), (), False)	0			

Table 7: Backward induction results for the optimal policy $\pi^*(s)$ and the optimal $Q^*(s,a)$ for Case4gs with $\gamma = 0.9$

Action	$Q^*(s_0, a)$	$q_{\epsilon,500}(s_0, a)$
(0)	0.9	0.55
(1)	1	1
(2)	0.9	0.54
(3)	1	1

Table 8: Risk assessment of each line contingency at the initial state s_0 using Dynamic Programming and the Monte-Carlo method with 500 episodes for Case4gs with $\gamma = 0.9$ and $\epsilon = 0.2$

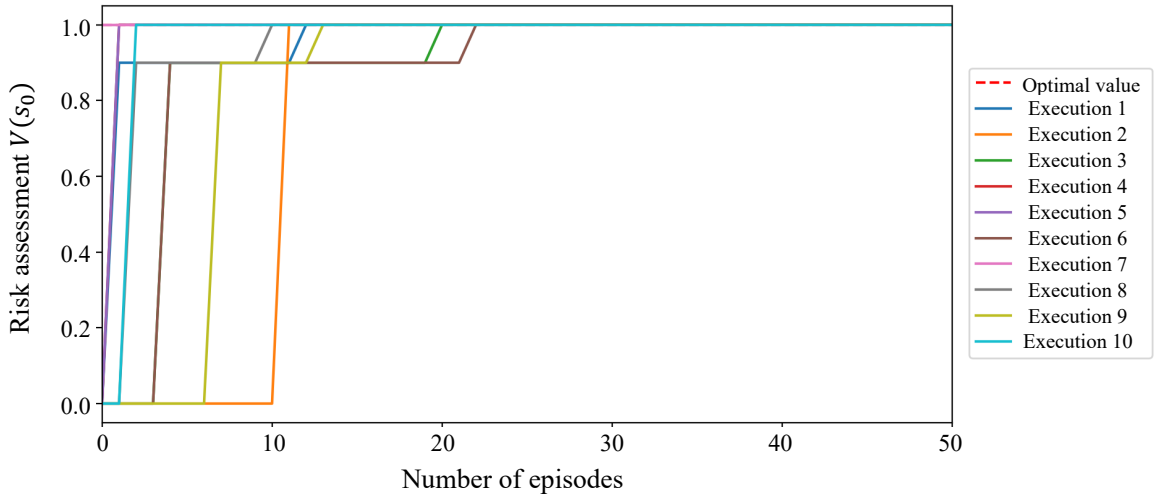

Figure 23: Risk assessment of each line contingency at the initial state s_0 using Dynamic Programming and the Monte-Carlo method with 500 episodes for Case4gs with γ=0.9 and ϵ=0.2

11.2 9-BUS POWER SYSTEM

Once, we have computed our tool for the small network, `case4gs`, we will move on with another small-size power system network made up of 9 buses: the IEEE case9 network. This network was first published in *Power System Control and Stability* (Anderson and Fouad's, 1980) and has been imported from PYPOWER to Pandapower.

As we can see in Table 5, this network approximately has a maximum state-space size of $9.9 \cdot 10^5$, which is a considerably larger size in comparison with the previous network `case4gs`. Using this power system case, we have been able to compute breadth-first search over all the state space in around 3.5 hours and we have observed a state space with a size of 749,756 states. Afterwards, we have computed the backward induction algorithm and we have computed the risk index Q^* in 5.6 hours. We can see in Table 9 the results for Q^* where we can notice that the three most vulnerable lines are 1 and 6 with a risk index of 1.8 followed by line 3 with an index of 0.9. Therefore, using (60) we can see that the initial state has an index risk of 1.8.

If we now compute the Monte-Carlo method, we can see in Figure 24 the evolution of the initial state value function associated with the deterministic version of the ϵ-greedy used for policy which iteratively improves using policy evaluation and policy improvement respectively. In this figure, we can see how, after 500 episodes and using $\gamma = 0.9$ and $\epsilon = 0.2$, nine out of ten executions converge to the optimal risk index of 1.8. Considering that each computation has taken a total of 2 minutes in comparison with a total of 8.1 hours for Dynamic Programming, the Monte-Carlo seems to have the potential to achieve acceptable results in a reasonable amount of time.

Nevertheless, if we analyze the average return from execution 1 in Figure 24, we can see in Table 9 that the average return for 500 episodes, which we will denote as $q_{\epsilon,500}(s_0, a)$, only detects a cascading failure risk in line 6. If we increase the Monte-Carlo experience to 1,000 and 10,000 episodes respectively, we can see in Table 9 how the computed indexes become better and better approximations of the optimal solution.

Action	$Q^*(s_0, a)$	$q_{\epsilon,500}(s_0, a)$	$q_{\epsilon,10^3}(s_0, a)$	$q_{\epsilon,10^4}(s_0, a)$
(0)	0	0	0	0
(1)	1.8	0	1.41	1.37
(2)	0	0	0	0
(3)	0.9	0	0	0.40
(4)	0	0	0	0
(5)	0	0	0	0
(6)	1.8	0.96	0.5	1.51
(7)	0	0	0	0
(8)	0	0	0	0

Table 9: Risk assessment for each line contingency at the initial state s_0 using Dynamic Programming and the Monte-Carlo method with 500, 1,000 and 10,000 episodes for Case9 with $\gamma = 0.9$ and $\epsilon = 0.2$

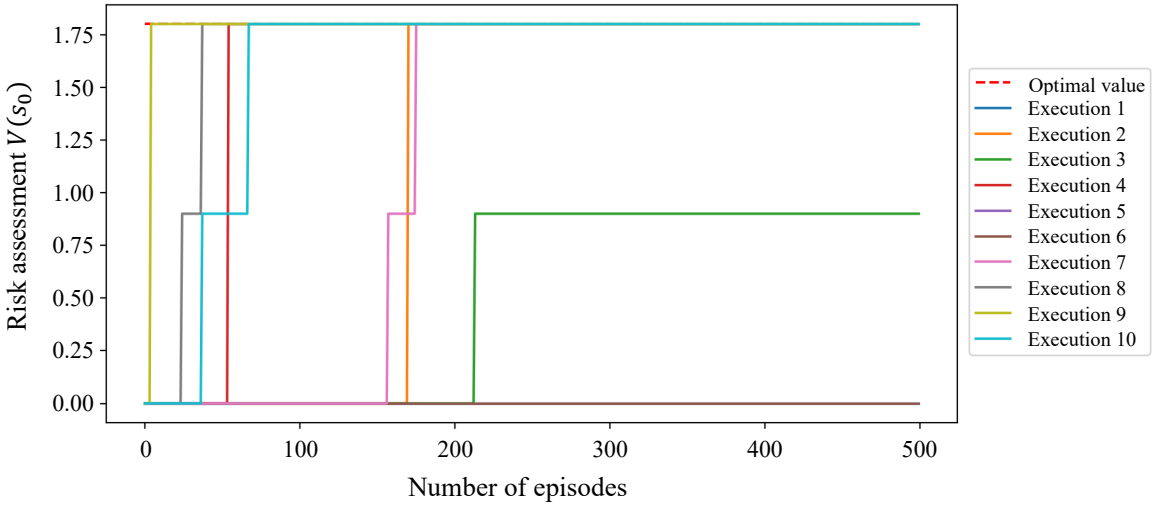

Figure 24: Risk assessment of initial state using the Monte-Carlo method with 500 episodes for Case9 with γ=0.9 and ϵ=0.2

12 MEDIUM-SIZE POWER SYSTEM EXAMPLES

12.1 39-BUS POWER SYSTEM

Continuing increasing the state-space complexity of our MDP model, we will move on to analyze the cascading failure risk in a 39-bus power system. This IEEE network was published for the first time in G. Bills et al., *On-line stability analysis study*, RP 90-1, North American Rockwell Corporation, Edison Electric Institute, Ed. IEEE Press, Oct. 1970 and has been imported to Pandapower from PYPOWER. This network is made of a total of 35 lines and has a maximum state-space of approximately $2.8 \cdot 10^{40}$.

We have not been able to compute Dynamic Programming in this network due to the unreasonable amount of time required for computing the optimal solution. Therefore we can conclude that the use of Dynamic Programming does not scale well to large state spaces and that there is an actual need for other approaches.

As we did with `case4gs` and `case9`, we have computed the Monte-Carlo method using 2,000 episodes and the same parameter values we have been using so far ($\gamma = 0.9$ and $\epsilon = 0.2$). We can show the results in Figure 25, where after just 2,000 episodes we can see no clear convergence among the different techniques and, therefore, we cannot guarantee to have achieved any optimal solution. Nevertheless, we can see that all these executions surely seem to never stop improving and, therefore, in light of the results computed for the small cases, we can expect them to finally converge to optimality as the number of episodes increases.

In Table 10, we show the risk index $q_{\epsilon,2000}$ for execution 3 for all the lines in the initial power system configuration. As we can see, we have been able to sort the most vulnerable lines and we have found a way to achieve the best solution possible given limited experience available despite the high dimensionality of this case.

In light of these results, we can state that this 39-bus power system has a cascading risk index of 14.12 at its initial configuration and that it has a great vulnerability in line 28 which has the potential to become the worst possible cascading failure in this system.

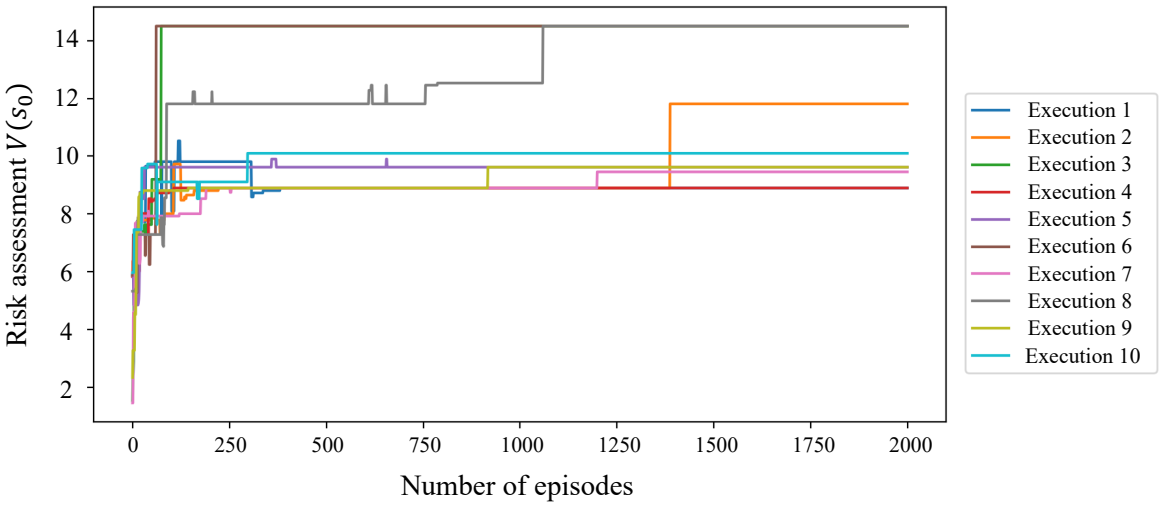

Figure 25: Risk assessment of initial state using the Monte-Carlo method with 2,000 episodes for Case39 with γ=0.9 and ϵ=0.2

	Action: a	$q_{\epsilon,2000}(s_0, a)$
1	(26)	14.12
2	(7)	8.27
3	(28)	7.91
4	(16)	7.91
5	(21)	7.29
6	(24)	7.22
7	(2)	6.90
8	(0)	6.75
9	(27)	5.98
10	(29)	5.95
11	(22)	5.50
12	(11)	4.92
13	(4)	4.75
14	(31)	4.60
15	(18)	4.44
16	(8)	4.38
17	(23)	4.25
18	(25)	4.23
19	(9)	4.00
20	(13)	4.00
21	(19)	3.91
22	(34)	3.86
23	(14)	3.55
24	(10)	3.46
25	(32)	3.35
26	(5)	2.92
27	(30)	2.81
28	(17)	2.73
29	(20)	2.58
30	(3)	2.23
31	(6)	2.18
32	(1)	2.12
33	(15)	2.04
34	(33)	2.04
35	(12)	1.62

Table 10: Sorted risk assessment for each line contingency at the initial state s_0 using the Monte-Carlo method with 2,000 episodes for case Case39 with $\gamma = 0.9$ and $\epsilon = 0.2$

12.2 89-BUS POWER SYSTEM

Finally, to test how far this algorithm can get, we will compute our tool for the IEEE network `case89pegase`. This network was obtained trough MATPOWER from the paper AC Power Flow Data in MATPOWER and QCQP Format: iTesla, RTE Snapshots, and PEGASE (C. Josz, S. Fliscounakis, J. Maenght, P. Panciatici; 2016). As we can see in Table 5, it has a total of 160 lines and a massive maximum state-size of approximately $1.3 \cdot 10^{285}$.

If we compute the Monte-Carlo algorithm with 2,000 episodes, we will visit, considering that the system has 160 lines, a total number of states less than $3.2 \cdot 10^5$ states which is considerably far from $1.3 \cdot 10^{285}$. As we show in Figure 26, despite the insignificant experience it has simulating cascading failures with respect to the total state-space we can see that there is still a process of learning in which we can see the algorithm improves the risk index of the initial state over time. This constant improvement regardless of the state-space size allows us to believe in the potential of RL within power systems. In order to just focus on the top 10 vulnerable lines of this system in terms of cascading failure, we have we show them in Table 11 sorted by risk index using execution 4. In light of this short experience of 2,000 iterations, we could say that this power system seems more prepared for cascading failures than the previous IEEE `case39` which had an index of 14.12 and less than 4 times the number of possible lines to cascade in comparison to `case89pegase`.

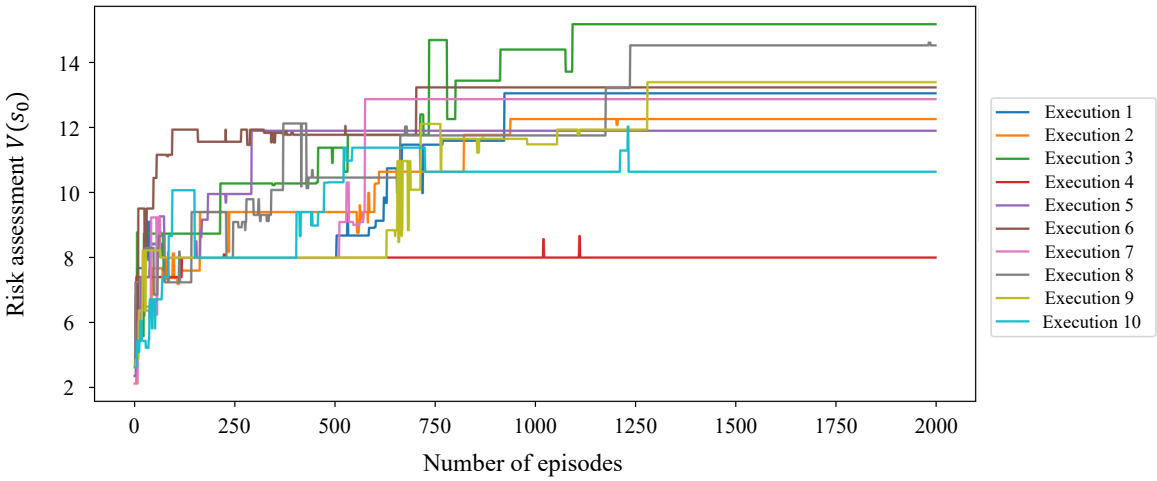

Figure 26: Risk assessment of initial state using the Monte-Carlo method with 2,000 episodes for Case89pegase with γ=0.9 and ϵ=0.2

	Action	Q
1	(61)	3.93
2	(19)	3.26
3	(7)	3.00
4	(17)	3.00
5	(58)	2.81
6	(76)	2.70
7	(102)	2.70
8	(142)	2.70
9	(13)	2.63
10	(98)	2.60

Table 11: Sorted risk assessment for the ten most dangerous line contingencies at the initial state s_0 using the Monte-Carlo method with 2,000 episodes for case Case89pegase with $\gamma = 0.9$ and $\epsilon = 0.2$

For a system made of 160 lines, where any could possibly fail after each disconnection, detecting a maximum discounted cascading failure of 3.93 seems like a too-small threat for even considering it. For cases like this, where the state-space can become a major challenge, reducing the state space by setting a maximum time-step $T_{\max}$ as we proposed in Section 8, becomes necessary. In this example, if we had established a limit of 10 disconnections by the agent, we would have got a maximum state space of approximately $8.3 \cdot 10^{21}$ which could not only be more affordable but also more appropriate for detecting major immediate security threats. As we have stated in the introduction of this thesis, the games of Go and chess present maximum state-spaces of 10^{120} and 10^{172} states respectively, and, despite this dimensionality, RL models have been able to learn enough to improve the human performance in both games.

From these results we can also conclude, that the use of random initialization can postpone significantly the early detection of major threats in a system. As the dimension of the power systems increase it becomes essential to properly initialize the Monte-Carlo method using simple power system considerations like the loading percent of each line of more complex techniques like the N-k security analysis.

Part VI. CONCLUSION

To address new challenges arising to power systems security in the upcoming years, this thesis develops a tool to assess the risk of cascading failures, a major threat to power systems. The tool takes advantage of the maturing of AI in recent years, applying a reinforcement learning framework that has proven to be successful among other fields.

The tool poses the problem of assessing the risk of cascading failure as a Markov Decision Process where we model an agent disconnecting lines from a power system and obtaining rewards in the form of the number of cascaded lines after each line disconnection. This way, we analyze the most critical sequences of cascading failures and assess the risk of cascading failure associated with each individual line contingency and each power system configuration. Our model assumes a constant power demand profile in order to focus our attention on the reinforcement learning aspect of the problem. We consider two methods for solving the problem: Dynamic Programming and the Monte-Carlo algorithm.

Within Dynamic Programming, we consider three different model-based techniques: backward induction, policy iteration and value iteration. Since these techniques require complete information of the Markov Decision Process model, we also present a breadth-first search algorithm to exhaustively learn the 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. We test breadth-first sea and backward induction on a 4-bus power system and a 9-bus power system. For the 4-bus power system with a state-space size of 35, this approach computes the optimal solution in just a few seconds. Nevertheless, the 9-bus power system contains 749,756 states and requires around 9 hours to solve to optimality. Clearly, Dynamic Programming cannot scale to real-world power systems, so we must investigate alternative approaches.

Instead, our tool uses the Monte Carlo method for larger state-spaces that are intractable for Dynamic Programming. This method works by generating random episodes and computing estimates of the optimal action value functions and the optimal policy. For the 4-bus system and the 9-bus system, Monte Carlo converges to the same solution as Dynamic Programming in considerably less time. Using Monte Carlo, we analyze a 39-bus and an 89-bus system with up to $2.8 \cdot 10^{40}$ and $1.28 \cdot 10^{285}$ states respectively, and give numerical evidence that the algorithm learns and improves its estimation over time as it acquires more experience.

In conclusion, reinforcement learning has the potential to be an invaluable advisory tool for grid operators for three main reasons. First, it learns with experience and improves its estimations over time. In power systems where simulation models are completely available, the tool can eventually gain a human-like intuition, similar to that of the grid operators. Second, reinforcement learning uses information from other analyses to enhance its estimations, and therefore stands to benefit from other power security and contingency analysis already in existence. Finally, the ability to adjust the maximum

time-step, the discounting rate, and the epsilon parameter open the door to a great number of options and possibilities for customizing this technique according to the needs of each individual power system.

BIBLIOGRAPHY

[1] About pandapower. http://www.pandapower.org/about/.

[2] Distributed generation electricity and its environmental impacts. https://www.epa.gov/energy/distributed-generation-electricity-and-its-environmental-impacts#ref1.

[3] Does EIA publish the location of electric power plants and transmission lines? https://www.eia.gov/tools/faqs/faq.php?id=567&t=3.

[4] Electricity in the united states. https://www.eia.gov/energyexplained/electricity/electricity-in-the-us.php.

[5] Energy queensland: The facts and myths around electricity prices and consumption. https://www.townsvilleenterprise.com.au/ energy-queensland-the-facts-myths-around-electricity-prices-and-consumption/.

[6] How much electricity does an american home use? https://www.eia.gov/tools/faqs/faq. php?id=97&t=3.

[7] Pandapower: Data structure and elements. https://pandapower.readthedocs.io/en/v2. 2.1/elements.html.

[8] Pandapower official website:. http://www.pandapower.org/.

[9] Power system test cases. https://pandapower.readthedocs.io/en/latest/networks/power_system_test_cases.html.

[10] State of the developer nation. https://slashdata-website-cms.s3.amazonaws.com/sample_reports/ZAamt00SbUZKwB9j.pdfs.

[11] The top programming languages 2019. https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.

[12] What a system operator is http://www.incsys.com/power4vets/what-is-a-system-operator/.

[13] What is Python executive summary. https://www.python.org/doc/essays/blurb/.

[14] Ercot control room video, October 2014. https://www.youtube.com/watch?v=TlsfItQc05o.

[15] Julie Allen. South america power cut leaves 48 million people without electricity. June 2019. https://www.telegraph.co.uk/news/2019/06/17/south-america-power-cut-leaves-48-million-people-without-electricity/.

[16] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[17] Transformational Disruptor and Andreas François Vermeulen. Industrial machine learning.

[18] D. A. Ferrucci. Introduction to "this is watson". *IBM Journal of Research and Development*, 56(3.4):1:1–1:15, 2012.

[19] Adrian Kelly, Aidan O'Sullivan, Patrick de Mars, and Antoine Marot. Reinforcement learning for electricity network operation. *arXiv preprint arXiv:2003.07339*, 2020.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] P Russel Norvig and S Artificial Intelligence. *A modern approach*. Prentice Hall, 2002.

[22] Mark Rothleder. Renewable integration, May 2017. https://www.caiso.com/Documents/RenewableIntegrationUnlockingDividends.pdf. 65

[23] Florian Schäfer and Martin Braun. An efficient open-source implementation to compute the jacobian matrix for the newton-raphson power flow algorithm. In *2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pages 1–6. IEEE, 2018.

[24] Surajbhan Sevda and Anoop Singh. *Mathematical and Statistical Applications in Food Engineering*. CRC Press, 2020.

[25] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[26] Thomas Allen Short. *Electric power distribution handbook*. CRC press, 2014.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] Ruohan Zhan, Tianchang He, and Yunpo Li. Cs221 project final report deep reinforcement learning in portfolio management.

[29] Guidong Zhang, Samson Shenglong Yu, Siyuan Zou, Herbert Ho-Ching Iu, Tyrone Fernando, and Yun Zhang. An investigation into cascading failure in large-scale electric grids: A load- redistribution approach. *Applied Sciences*, 8(7):1033, 2018.

APPENDIX I

1. Code related with the Exhaustive Search Algorithm within Dynamic Programming

- **Function for computing the reward $r(s,a)$ and the next states $s'(s,a)$ from a state s and taking action a**

```
def compute_state_action(state, action):
    global num_lines
    global net
    disconnect = list(action)
    total_cascade = []
    off_lines = list(state[0]) + list(state[1]) + disconnect
    terminal = len(off_lines) == num_lines
    i = 0
    while i == 0 or not terminal and len(disconnect):
        net.line.loc[off_lines, "in_service"] = False
        try:
            pp.runpp(net)
            disconnect =
list(net.res_line.loc[net.res_line.loading_percent >
net.line.max_loading_percent].index)
            total_cascade += disconnect
            off_lines += disconnect
            terminal = len(off_lines) == num_lines
        except pp.powerflow.LoadflowNotConverged:
            terminal = True
        i += 1
    net.line.loc[off_lines, "in_service"] = True
    next_state = (tuple(list(state[0]) + list(action)),
tuple(list(state[1]) + total_cascade), terminal)
    reward = len(total_cascade)
    return next_state, reward
```

- **Function for obtaining all the possible actions from a state ($\mathcal{A}(s)$)**

```
def getactions(state):
    global num_lines
    off_lines = list(state[0]) + list(state[1])
    actions = [tuple([i]) for i in range(num_lines) if i not in
off_lines]
    return actions
```

- **Main code for computing Dynamic Programming (Exhaustive Search)**

```
import pandapower as pp
import pandapower.networks as nw
import pandas as pd
import time
initial_time = time.time()

# Dictionaries:
# state = ((shutted_ines),(cascade lines),(terminal_node))
# reward[state][action]
# next_state[state][action]
# action is a list
# state space is a list with all the possible states


pd.options.display.max_columns = 500
pd.options.display.max_rows = 5

net = nw.case4gs()
num_lines = len(net.line)
non_terminal_states = []
terminal_states = []
reward = dict()
reward[((), (), False)] = dict()
next_state = dict()
next_state[((), (), False)] = dict()

initial_state = compute_state_action(((), (), False), [])[0]
non_terminal_states.append(initial_state)

print("1. Learning model: ")
queue = [initial_state]
while len(queue) != 0:
    leaf_state = queue[0]
    reward[leaf_state] = dict()
    next_state[leaf_state] = dict()
    for action in getactions(leaf_state):
        next_state[leaf_state][action], reward[leaf_state][action]
= compute_state_action(leaf_state, action)
        terminal = next_state[leaf_state][action][2]
        if terminal is False:
            queue.append(next_state[leaf_state][action])

non_terminal_states.append(next_state[leaf_state][action])
        else:
            terminal_states.append(next_state[leaf_state][action])
        if len(next_state[leaf_state][action][0]) !=
len(leaf_state[0]):
            print("- Stage: " +
str(len(next_state[leaf_state][action][0]))+ "/"+str(num_lines),
end ="")
            print("\r", end="")
```

```
    queue.pop(0)

model_time = time.time()
print("Model computed in ", round(model_time-initial_time, 3), "
s.")
print("Non terminal states: ", len(non_terminal_states))
print("Terminal states: ", len(terminal_states),"\n\n")


Q_optimal = dict()
gamma = 0.9
print("2. Computing Q-values: ")
for state in reversed(non_terminal_states):
    Q_optimal[state] = dict()
    for action in getactions(state):
        if next_state[state][action] in terminal_states:
            max_Q = 0
        else:
            max_Q = -1
            for next_action in
getactions(next_state[state][action]):
                if
Q_optimal[next_state[state][action]][next_action] > max_Q:
                    max_Q =
Q_optimal[next_state[state][action]][next_action]
        Q_optimal[state][action] =
reward[state][action]+gamma*max_Q

        print("- Stage: " + str(num_lines-len(state[0])) + "/" +
str(num_lines), end="")
        print("\r", end="")


final_time = time.time()
print("Q-values computed in ", round(final_time-model_time, 3), "
s.\n\n")


# Trajectory: S1;A1;R2;S2;
print("3. Optimal trajectory: ")
optimal_trajectory = [initial_state]
reached = False
state = initial_state

while reached is False:
    max_Q = -1
    for action in getactions(state):
        if Q_optimal[state][action] > max_Q:
            optimal_action = action
            max_Q = Q_optimal[state][action]
    optimal_trajectory += [optimal_action,
reward[state][optimal_action], next_state[state][optimal_action]]
    state = optimal_trajectory[-1]
    if state in terminal_states:
        reached = True

print(optimal_trajectory, "\n\n")
print("4. Value of initial state: ",
```

```
max([Q_optimal[initial_state][action] for action in
getactions(initial_state)]))
```

- **Code for saving the results of the rewards ($\mathcal{R}$), the action value function ($Q(s,a) \forall s \in \mathcal{S}, s \in \mathcal{A}$) and the system dynamics ($s'(s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}$)**

```
# Save Q(s,a), s'(s,a) and r(s,a) into excel files:
# Q-table

import pandas as pd

data = []
all_actions = [tuple([i]) for i in range(num_lines)]
for state in non_terminal_states:
    data.append([state]+[Q_optimal[state].get(tuple([i]), None) for
i in range(num_lines)])
df = pd.DataFrame(data, columns = ['State']+['Action ' + str(i) for
i in range(num_lines)])
df.to_excel("DP Q optimal.xlsx")

# s'(s,a)
data = []
all_actions = [tuple([i]) for i in range(num_lines)]
for state in non_terminal_states:
    data.append([state]+[next_state[state].get(tuple([i]), None)
for i in range(num_lines)])
df = pd.DataFrame(data, columns = ['State']+['Action ' + str(i) for
i in range(num_lines)])
df.to_excel("DP Next states.xlsx")

# r'(s,a)
data = []
all_actions = [tuple([i]) for i in range(num_lines)]
for state in non_terminal_states:
    data.append([state]+[reward[state].get(tuple([i]), None) for i
in range(num_lines)])
df = pd.DataFrame(data, columns = ['State']+['Action ' + str(i) for
i in range(num_lines)])
df.to_excel("DP Reward.xlsx")
```

2. Code related with the Monte-Carlo Algorithm

- **Function for obtaining the optimal action a^* to take from a state s:**

$$a^* = \arg\max_{a \in \mathcal{A}(s)} Q(s, a))$$

```
def pi_optimal(state):
    if state in pi_optimal_dict:
        action = pi_optimal_dict[state]
    else:
        found = False
        off_lines = list(state[0]) + list(state[1])
        k = -1
        while found is False:
            k += 1
            found = k not in off_lines
        action = tuple([k])
    return action
```

- **Function for obtaining the action-value function $Q(s, a)$ given a state s and an action a**

```
def Q(state, action):
    global Q_dict
    if Q_dict.get(state).get(action) is not None:
        result = Q_dict[state][action]
    else:
        result = 0
    return result
```

- **Functions for returning a random action using an ϵ-greedy policy given a state *s* and an action *a***

```
def pi_epsilon(state):
    global epsilon
    optimal_action = pi_optimal(state)
    all_actions = getactions(state)
    num_actions = len(all_actions)
    non_optimal_actions = all_actions+[]
    non_optimal_actions.remove(optimal_action)
    action = random_action([1-
epsilon+epsilon/num_actions]+[epsilon/num_actions]*(num_actions-1),
[optimal_action]+non_optimal_actions)
    return action
```

```
def random_action(pmf, list):
    p = random.uniform(0, 1)
    found = False
    i = 0
    cdf = pmf[0]
    while found is False:
        if p < cdf:
            action = list[i]
            found = True
        else:
            cdf += pmf[i+1]
            i += 1
    return action
```

- **Function for saving the returns after each episode *E* for each state *s* and action *a***

```
def returns_append (G, state, action):
    global returns
    if returns.get(state) is not None:
        if returns.get(state).get(action) is not None:
            returns[state][action].append(G)
        else:
            returns[state][action] = [G]
    else:
        returns[state] = dict()
        returns[state][action] = [G]
```

- **Function for generating a random episode using an ϵ-greedy policy**

$$\boldsymbol{E} = \{(s_i, a_i, r_{i+1})\}_{i=0}^{T-1}$$

```
def generate_episode():
    global pi_optimal_dict
    global initial_state
    state = initial_state
    terminal = state[2]
    episode_states = []
    episode_actions = []
    episode_rewards = []
    while state[2] is False:
        action = pi_epsilon(state)
        next_state, reward = compute_state_action(state, action)
        episode_states.append(state)
        episode_actions.append(action)
        episode_rewards.append(reward)
        state = next_state
    return episode_states, episode_actions, episode_rewards
```

- **Function for defining the value of the action-value function $Q(s,a)$ for a given state s and action a**

```
def set_Q (state, action, value):
    global Q_dict
    if Q_dict.get(state) is not None:
            Q_dict[state][action] = value
    else:
        Q_dict[state] = dict()
        Q_dict[state][action] = value
```

- **Function for returning the average value from all the element of a list**

```
def mean(list):
    return sum(list)/len(list)
```

- **Function for defining the optimal action a^* for a given state s**

```
def set_pi_optimal(state):
    global pi_optimal_dict
    max_Q = -1
    for action in getactions(state):
        if Q(state, action) > max_Q:
            max_Q = Q(state, action)
            optimal_action = action
    pi_optimal_dict[state] = optimal_action
```

- **Function for computing a cascading failure simulation for obtaining the next state $s'(s,a)$ and the reward $r(s,a)$ of a given state s' and action a**

```
def compute_state_action(state, action):
    global num_lines
    global net
    disconnect = list(action)
    total_cascade = []
    off_lines = list(state[0]) + list(state[1]) + disconnect
    terminal = len(off_lines) == num_lines
    i = 0
    while i == 0 or not terminal and len(disconnect):
        net.line.loc[off_lines, "in_service"] = False
        try:
            pp.runpp(net)
            disconnect =
list(net.res_line.loc[net.res_line.loading_percent >
net.line.max_loading_percent].index)
            total_cascade += disconnect
            off_lines += disconnect
            terminal = len(off_lines) == num_lines
        except pp.powerflow.LoadflowNotConverged:
            terminal = True
        i += 1
    net.line.loc[off_lines, "in_service"] = True
    next_state = (tuple(list(state[0]) + list(action)),
tuple(list(state[1]) + total_cascade), terminal)
    reward = len(total_cascade)
    return next_state, reward
```

- **Function for returning list of possible actions $\mathcal{A}(s)$ to take from a state s**

```
def getactions(state):
    global num_lines
    off_lines = list(state[0]) + list(state[1])
    actions = [tuple([i]) for i in range(num_lines) if i not in
off_lines]
    return actions
```

- **Function for returning the list of possible actions $\mathcal{A}(s)$ to take from given state s**

```
def V_optimal_initial_state():
    global initial_state
    state = initial_state
    episode_states = []
    episode_actions = []
    episode_rewards = []
    while state[2] is False:
        action = pi_optimal(state)
        next_state, reward = compute_state_action(state, action)
        episode_rewards.append(reward)
        state = next_state
    discount_vector = [gamma**i for i in
range(len(episode_rewards))]
    return np.dot(episode_rewards, discount_vector)
```

- **Main code for computing the Monte-Carlo Algorithm using a ϵ-greedy policy**

```
import pandapower as pp
import pandapower.networks as nw
import pandas as pd
import time
import random
import numpy as np

initial_time = time.time()

net = nw.case39()
num_episodes = 2000
num_computations = 10
gamma = 0.9
```

```
epsilon = 0.2
num_lines = len(net.line)
initial_state = compute_state_action(((), (), False), [])[0]
ydata = []
for i in range(num_computations):
    pi_optimal_dict = dict()
    Q_dict = dict()
    returns = dict()
    error_G = []
    V_optimal_evolution = []
    print(" Monte Carlo computation: "+ str(i) + " | episodes: " +
str(num_episodes) )
    for i in range(num_episodes):
        episode_states, episode_actions, episode_rewards =
generate_episode()
        G = 0
        for t in reversed(range(len(episode_rewards))):
            print("- Episode: " + str(i) + " | t: " + str(t),
end="")
            print("\r", end="")
            G = G*gamma+episode_rewards[t]
            returns_append(G, episode_states[t],
episode_actions[t])
            set_Q(episode_states[t], episode_actions[t],
mean(returns[episode_states[t]][episode_actions[t]]))
            set_pi_optimal(episode_states[t])
        V_optimal_evolution.append(V_optimal_initial_state())
    ydata.append(V_optimal_evolution)
```

- **Code for plotting the evolution of the risk assessment $V(s_0)$ of the initial state s_0 for each *i*--th iteration**

```
import matplotlib.pyplot as plt
xdata = [i for i in range(num_episodes)]
#yoptimal = [1.8 for i in range(num_episodes)]
#plt.plot(xdata, yoptimal, 'r--')
for i in range(num_computations):
    plt.plot(xdata, ydata[i], label="Line "+str(i))
plt.plot()
#plt.legend(["Optimal value"]+["Execution "+str(i+1) for i in
range(num_computations)], loc="center right")
plt.legend(["Execution "+str(i+1) for i in
range(num_computations)], loc="center right")
plt.xlabel('Number of episodes')
plt.ylabel('Value of initial state following obtained policy')
plt.show()
```

- **Code for saving into an Excel file the optimal policy $\pi^*(s)$ for every state $s \in \mathcal{S}$**

```
"""""
# Save pi_optimal into an excel file:

from DP import non_terminal_states
import pandas as pd

data = []
for state in non_terminal_states:
    data.append([state, pi_optimal(state)])
df = pd.DataFrame(data, columns = ['State', 'PI (state)'])
df.to_excel("MC policy.xlsx")
"""
```

- **Code for saving into an Excel file the action-value function $Q(s_0, a)$ for the initial state s_0 and every possible initial action $a \in \mathcal{A}(s_0)$**

```
# Save Q for initial state and actions into an excel file:
df = pd.DataFrame(columns=['Action', 'Q'])
for a in getactions(initial_state):
    df = df.append({'Action': a,'Q':
Q(initial_state,a)},ignore_index=True)
df.to_excel("MC initial Q values 2000 39.xlsx")
```

APPENDIX II: SUSTAINABLE DEVELOPMENT GOALS

In this appendix we will analyze the relation between this project and the sustainable development goals and how this work alongside other investigations in this area can contribute to these goals.

This project is strongly related with two out of the 17 sustainable development goals:

- **Objective 7**: Ensure access to affordable, reliable, sustainable and modern energy.
- **Objective 9**: Build resilient infrastructure, promote sustainable industrialization and foster innovation.

Objective 7: Energy

The importance of energy nowadays in essential in undeniable and therefore it is essential to support an initiative which guarantees universal access to energy, improves the efficiency y increases the use of renewables to overcome climate change.

The rules which are mostly related with this project are the following:

- 7.1 By 2030, ensure universal access to affordable, reliable and modern energy services.
- 7.2 By 2030, increase substantially the share of renewable energy in the global energy mix.
- 7.3 By 2030, double the global rate of improvement in energy efficiency
- 7.A By 2030, enhance international cooperation to facilitate access to clean energy research and technology, including renewable energy, energy efficiency and advanced and cleaner fossil-fuel technology, and promote investment in energy infrastructure and clean energy technology.

The improvement of the tools used to guarantee safe operation to power system not only is contributing to generation of reliable energy but also allowing investments in new energy sources with high variability specially with distributed systems.

Objective 9: Industries, Innovation and Infrastructure

The power system infrastructure is probably the biggest infrastructure in the world and therefore guaranteeing its safe operation and control is crucial.

- 9.3 Increase the access of small-scale industrial and other enterprises, in particular in developing countries, to financial services, including affordable credit, and their integration into value chains and markets

- 9.5 Enhance scientific research, upgrade the technological capabilities of industrial sectors in all countries, in particular developing countries, including, by 2030, encouraging innovation and substantially increasing the number of research and development workers per 1 million people and public and private research and development spending

This project proposes a situational awareness tool able to assess a major problem in power systems: cascading failures. This can contribute to having a resilient infrastructure more prepared for this type of threats.

Also, this project is specially inspired by the penetration of distributed energies and therefore predicting not only it presence but also the importance of dealing safely with the new formats of power systems.