TECHNISCHE UNIVERSITÄT MÜNCHEN

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

Master's Thesis in Information Systems

Clearing Electricity Market of Variable Size with Graph Neural Networks

Nadine Angermeier

TECHNISCHE UNIVERSITÄT MÜNCHEN

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

Master's Thesis in Information Systems

Clearing Electricity Market of Variable Size with Graph Neural Networks

Optimierung des Strommarkts variabler Größe mit Graph Neural Networks

Author:	Nadine Angermeier
Supervisor:	Prof. Martin Bichler
Advisor:	Mete Şeref Ahunbay
Submission Date:	15th of May 2023

I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 15th of May 2023 Nadine Angermeier

Acknowledgments

I want to thank my supervisor Mete Ahunbay for letting me work on this fascinating combination of topics, for his input and support throughout the project. Thanks to Johanna Sommer for her quick insight into her research area. Last but not least, I thank my family and friends for their moral support and feedback during my work on this thesis.

Abstract

The energy market is at the forefront of the climate change debate. Changes in energy sources, demand and distribution pose challenges for market optimization. The energy market clearing and the associated optimal power flow (OPF) problem create energy distribution schedules while balancing generation and demand. The goal is to optimize global welfare under complex technical constraints. As of the present day, solutions are computed via mixed-integer programming (MIP). This optimization problem is very complex, requiring the best techniques and technologies as well as time. In order to ensure energy reliability and give the right impulses in almost real-time, the energy market clearing needs to become faster.

Machine learning (ML) techniques offer the potential to mimic these linear optimizers with almost real-time predictions after training phases. Since the OPF problem is repeatedly solved, the high amount of unused historical data provides the needed conditions for such statistical data-based models. Successful models have already been found on MIP and OPF predictions under academic conditions.

In our paper, we solve an advanced version of the DC-OPF model with a full inclusion of the unit commitment problem. We deep dive into improvement approaches on MIP and OPF with ML so far. Thereby we come across the limit of not including variable grid sizes in the ML approaches. However, the energy grid is continuously changing and growing, and real-life OPF optimization support tools need to be able to handle these changes. This paper inspects and defines approaches to account for this data structure within OPF optimization.

Within the paper, we conduct a data search through publicly available energy data sources and make suggestions on how to create usable data sets for DC-OPF optimization. Then we state the data preparation steps and explain a method to form dynamic grid data. Next, we look at different advanced neural network (NN) structures and discuss general approaches to combat variable grid size in NNs. Finally, we choose dynamic temporal graph neural networks to create a first implementation and test its performance. While the model can handle this data structure and creates good test scores for fixed-size generation prediction (91%), variance in the data and, specifically, different grid sizes pose significant challenges to the model.

Kurzfassung

Der Energiemarkt steht in der Debatte über den Klimawandel an vorderster Front. Veränderungen bei den Energiequellen, der Nachfrage und der höheren Verteilung der Ressourcen stellen eine Herausforderung für die Marktoptimierung dar. Bei der Energiemarktoptimierung (engl. energy market clearing) und dem damit verbundenen Problem der optimalen Leistungsflussberechnung (engl. optimal power flow, OPF) werden Energieverteilungspläne erstellt und gleichzeitig Erzeugung und Nachfrage ausgeglichen. Ziel ist, das optimale Wohlergehen aller Parteien unter komplexen technischen Bedingungen zu erreichen. Bis heute werden die Lösungen mittels gemischt-ganzzahliger Programmierung (engl. mixed-integer programming, MIP) berechnet. Das Optimierungsproblem ist sehr komplex und erfordert die besten Methoden und Technologien sowie einen hohen Zeitaufwand. Um die Zuverlässigkeit der Energieversorgung zu gewährleisten und die richtigen Impulse in nahezu Echtzeit auszusenden, muss die Energiemarktoptimierung schneller werden.

Techniken des maschinellen Lernens (ML) bieten das Potenzial, diese linearen Optimierer zu imitieren und nach Trainingsphasen Vorhersagen fast in Echtzeit zu treffen. Da das OPF-Problem wiederholt gelöst wird, bietet die große Menge ungenutzter historischer Daten die notwendigen Voraussetzungen für solche auf statistischen Daten basierende Modelle. Erfolgreiche Beispiele für die Nutzung von ML für das Bestimmen von MIP und OPF-Lösungen konnten unter akademischen Bedingungen in der Forschung umgesetzt werden.

In unserer Arbeit lösen wir eine fortgeschrittene Version des DC-OPF-Modells unter vollständiger Einbeziehung des binären Einsatzproblems der Ressourcen (eng. unit commitment). Wir inspizieren die bisherigen Verbesserungsansätze für MIP und OPF mit ML. Dabei wird eine Grenze der bisherigen Forschung aufgedeckt, da die variable Größe der Energienetze in die ML-Ansätze nicht miteinbezogen wird. Mit der Zeit wächst und verändert sich das Energienetz jedoch ständig. Daher müssen Hilfsmittel zur Lösung des OPF in der Lage sein, mit diesen Veränderungen umzugehen. In diesem Projekt werden Ansätze zur Berücksichtigung der spezifischen Datenstruktur in der OPF-Optimierung untersucht und definiert.

Im Laufe dieser Arbeit führen wir eine Datensuche in öffentlich zugänglichen Energiedatenquellen durch und machen Vorschläge, wie man daraus Datensätze für die DC-OPF-Optimierung kombinieren und erstellen kann. Anschließend erläutern wir die Schritte der Datenaufbereitung und erklären eine Methode zur Generierung dynamischer Energienetze. Als Nächstes betrachten wir verschiedene erweiterte Strukturen neuronaler Netze (NN) und erörtern allgemeine Ansätze zur Einarbeitung variabler Eingabedaten in NNs. Schließlich wählen wir dynamische temporale Graphen-basierte NNs (engl. dynamic temporal graph neural networks), um eine erste Implementierung zu erstellen und testen ihre Genauigkeit. Während das Modell mit dieser Datenstruktur variabler Größe umgehen kann und gute Testergebnisse für die Vorhersage des Generationsvolumens (91%) in fixierten Netzen erzielt, stellen die Varianz in den Daten und insbesondere die unterschiedlichen Netzgrößen eine Herausforderung für das Modell dar.

Contents

1. Introduction

The energy transition is a central element of a secure and climate-friendly future. It requires a fundamental transformation of the German energy system (and worldwide) in energy sources and energy efficiency, as stated by the German Federal Ministry of Economic Affairs and Climate Action. [1] Alternative energy sources and insufficient energy supply reliability pose new challenges to the energy systems. Energy markets must constantly balance demand and generation under complex system requirements while creating the right incentives for market growth. A central part of the energy market design is the real-time market, which takes in the given bids of energy consumers and offers of generators. It is optimized to maximize global welfare, which refers to the consumers' value for the energy supplied minus the total cost incurred by generators. [2]

The energy market clearing defines the operational schedules and locational marginal prices (LMPs), both optimizations being dependent on each other. The results are binding conditions for the market participants. [2] The sub-problem of creating these schedules (or dispatch), called optimal power flow (OPF), is often discussed in research [3]. Due to technological constraints, such as start-up costs and resource capacities, it is so far solved as a mixed-integer programming (MIP) problem. It includes unit commitment (UC), turning on or off for machines [4], and economic dispatch (ED), generation volume of machines [5], as components [6].

The energy market is growing in size and complexity. In order to ensure the proper incentives and conditions for market participants, the system has to be optimized almost in real-time. A goal within the next years is to reduce the optimization intervals further to a few minutes. Currently, solving this problem requires the newest hardware and software as well as thousands of servers, to run the system. [2] Even sixty years after the first formulation of the OPF model, it is still searched for a more efficient and reliable solution technique [3]. Additional complexity has been created through social and technological changes over the recent years. Especially the increasing amount of renewable energy worsens uncertainty within the market, as their availability is irregular. The growing distribution of the system through the spread of included resources, adds to the system's complexity. [3] Additionally, the demand structure is changing, for example, through smart homes adapting their consumption to real-time prices. These new complexities require even faster technologies and better techniques [7]. [2]

The constant re-solving of the problem has created an immense amount of unused historical data [3][7]. Machine learning (ML) algorithms can utilize this data. They use statistical techniques to extract underlying data structures from old data pools to mimic a target solution variable [8]. Although ML methods might take hours to train, the testing or prediction is made in real-time (mostly less than a second). This creates the opportunity to reduce the optimization time significantly. Research has shown successful examples of ML models for OPF optimization. Early studies focus on predicting OPF results directly. In contrast, newer ones help make the optimization faster by, for instance, predicting a subset of necessary constraints or supplying the solver with better warm starting points. [3] These solutions however, often use artificially generated data that do not fully capture the variability within the real energy market [9]. Other simplifications are also applied.

In this thesis, we primarily focus on the variability of the energy market created by variable sizes of the energy grid. The market optimization takes the bids of participants into account which are connected to different nodes of the energy grid. With changes in market participants, the energy grid continuously grows and changes. So applying a model that can handle variable input data is necessary or it ceases to be applicable with time. This problem is mainly neglected in approaches so far [10]. We take a step away from the neural network (NN) models commonly found in successful recent solutions [3] that cannot handle these variable inputs as they take a fixed-size vector as input. We take a deep look into the data structure to solve the simplified market clearing (SMC) problem adapted from [6] and discuss which changes to the classical NNs are necessary to encompass the complexity.

This is done in three main chapters. First, we examine the literature on what has been done so far to optimize general MIPs and the OPF. Each literature review is divided into traditional linear optimization and experiments to solve the problems with ML. For MIP and OPF, we give an insight into the current optimization approach, including used algorithms, solvers and challenges. Then we conduct a literature review using keyword-based online research with one-step backward research to find papers that optimize the problem with ML. For each paper, we summarize the method, data set and results. At the end of the literature review, a taxonomy is created to categorize the different approaches and give an accessible overview in a mind map. The insights into experiments so far generate a baseline for our next steps.

In the second chapter, we search for realistic data sources and create a data set based on the found databases. There is yet to be a standard public database with realistic data. Some projects that try to solve this lack of data remain hidden or only include parts of the needed data. We thoroughly inspect the SMC model, which encompasses the scheduling logic yet keeps to few conditions, and its variables to identify the needed data points. Simplifications to the real-world setting are also stated. Then we go through openly available sources for energy market data. We provide an overview of energy data sources and energy data generation tools encompassing the provided data types, formats and sources. We also discuss possible data connections and selections, including approaches from the literature review. Data combinations are available for Europe, Germany and parts of the US.

After the data selection step, we explain the data preparation steps and how to generate dynamic energy grids with feasible OPF solutions. The process starts with a definition of data collection and cleaning steps. We define formulas to calculate the needed variables and to normalize the data for optimization feasibility. Pseudo-code to achieve the correct format is also provided. The generation of dynamic data grids is explained by using node splitting and node concatenation. The theory, formulas as well as pseudo-code is defined. For the final step of data generation, the data has to be optimized with a linear MIP optimizer that we want to mimic (yet make faster). We chose the Gurobi optimizer, for which we explain the process and provide code.

In the last chapter, we inspect the hurdles of the data structure in NNs. First, we define the characteristics of the data in the context of NNs, including the spatio-temporal nature and dynamic number of variables. Then, we test different advanced NN structures for applicability to this problem, which hints toward temporal or recursive graph neural networks. To ensure that no more straightforward solutions are possible, we discuss different approaches to incorporate variable input sizes in traditional NNs and give reasons for their inapplicability. Additionally, we inspect possible dynamic approaches within the literature review. A complex combination of layers within a convolutional neural network or approaches utilizing graph neural networks (GNNs) are deemed to provide the needed mathematical expressiveness. We identify one approach to solving OPF with ML in [11] that can handle variable input sizes using a graph convolutional network (GCN) to create embeddings.

Finally, we decide to work with dynamic temporal graph neural networks, which can mimic the full expressiveness of spatio-temporal data [12]. The graph data thereby refers to one energy grid with consumers and generators at each node (bus), for which the phase angle and generation volume are predicted. The number of nodes can change over time. The use of dynamic temporal GNNs enables us to process graph relations of the energy grid and the temporal connections for each node instance, while the implementations can handle variable input sizes with dynamic graphs. This approach was also concluded as the next step in an advanced energy market literature review already discussing GNNs [10].

We provide a strategy to develop a model with these qualities from open-source tools with full instructions and return first performance tests. We choose the Python PyTorch Geometric Temporal library [13] for this task and optimize for the full requirements of the SMC problem. We give a detailed overview of the layers and chosen hyperparameters. Finally, the diffusion convolution recurrent NN (DCRNN) layer by [14] is chosen with some other mechanics to combat overfitting and a linear output layer to return a linear regression of dimension two. The complete code is provided, with additions in our repository [15].

The proposed model can predict generation and phase angles from the OPF data variables for same-size and dynamic energy grid data sets. Our evaluation metric, denoted by R^2, reaches an explainable variance up to 91% with this model to predict generation for same-size data sets. For predicting both targets, the R^2 goes towards 67.5% with same-sized data. While the model can support dynamic graphs, we figured that its prediction accuracy highly depends on the differences within the data sets. The data structure poses some challenges to the model, such as two targets of different dimensions. At last, advice on further use, hurdles and needed improvements to the model are discussed.

Our results provide evidence that handling dynamic grid input data for OPF optimization is becoming possible and might be used as a warm starting point method for now. Dynamic GNNs are a new and hot topic in research with sufficient research efforts, e.g., at Twitter [12][16]. With further advances in this research area, the models should improve and we advise further efforts to test their applicability to diverse real-life use cases.

As the final outcomes of this thesis, we create an overview and categorization of solution approaches of ML to the OPF problem and find a research gap. We give an overview of openly available data sources and their combinations. We then give in-depth instructions to prepare dynamic grid data on all variables needed for the SMC problem that can be fed to the ML network. We discuss approaches to solve OPF with NNs and variable input data. We formulate a dynamic temporal GNN model to encompass this complexity and provide a fully working implementation with performance tests.

2. Literature Review

A literature review on previous research projects was conducted to find which techniques have been applied so far, which data was used, what the results were and what potential problems have been. These insights should create a baseline for further decisions and work within this project. A keyword-based search with selection criteria was applied first, existing literature reviews were regarded and, at last, through a one-step backward search in references missing papers were added. The literature review is divided into two parts. The first inspects general mixed-integer linear programming optimization methods. The second discusses auction-based optimization of optimal power flow (OPF) and the included unit commitment (UC) problem. In each section, we discuss traditional methods first, then applied methods to solve the problems with machine learning.

2.1. General Mixed Integer Programming (MIP)

2.1.1. MIP Optimization

General optimization problems include an objective function to be optimized under constraints with solution variables. If the space of some solution variables is restricted to integer (or binary), special algorithms are needed to solve the problem feasibly. They are discussed in the mathematical theory of *mixed-integer programming (MIP)*. Depending on the nature of the objective function *mixed-integer linear programming (MILP)* optimizes a linear objective function under constraints with some integer or binary solution variables [17]. *Mixed-integer quadratic programming (MIQP)* solves objective functions with a quadratic term and linear constraints respectively [18]. Electricity generation planning, as discussed in research, is a MILP problem [19]. The model used in practice for European energy market clearing is MIQP [20].

Specific mathematical **algorithms** were developed to optimize these problems, which are implemented in software solvers. The traditional algorithm is called *branch-and-bound* (B&B). During processing, the integrality assumption is relaxed. This allows for finding the optimal fractional solution, at which the model is divided into two integer sub-problems by adding a bounding condition ($\leq$ the rounded-down optimal solution and $\geq$ the rounded-up optimal solution). This creates a search tree with a constrained, thus simplified, sub-problem at each node. The algorithm stops once the sub-problem or branch reaches an integral solution, is lower than an already found solution or is infeasible. [21]

A common improvement to this algorithm is *branch-and-cut*, where a cut or constraint is added to the problem with a specific algorithm or heuristic instead of branching through bounds. [17] With so-called *cutting planes*, constraints are found and added to the problem without cutting feasible solutions. Branch-and-cut is the state-of-the-art in commercial integer programming solvers. [22] There are different methods to create cuts. Mixed integer cuts by [23] provide the highest speedup and Knapsack cover cuts reach the highest performance. Cutting planes can reach a speedup factor of 54 compared to branch-and-bound. [17] As the third type of algorithm, with *decomposition algorithms* MIPs are partitioned into sub-problems for tractability. There are various techniques, some of the most common ones being column generation, Blenders' decomposition and Lagrangian relaxation. [24]

There are commercial and noncommercial software **solvers** that implement these algorithms [25]. IBM-CPLEX, Gurobi and FICO Xpress are the most powerful mathematical solvers at the moment [26]. Gurobi will be used in this paper (with access through an academic license). A state-of-the-art noncommercial solver is SCIP [27] and can be found in many research papers, e.g., [19][28][29].

Improvement techniques for MIP optimization are pre-processing/presolve and heuristics, parallelization and fine-tuning of the algorithm [30]. *Pre-processing* can create a speedup of 11 [17]. Using a *primal heuristic*, so a heuristic applied at the beginning, can lead to double speed and more feasible solutions for hard problems [31]. The most successful heuristics are feasibility pump and crossover for problems solved within one hour [32] (tested for SCIP). Regarding *fine-tuning*, the search, branching and pruning techniques can be adapted. The current state-of-the-art or best benchmark performance was achieved by the following configuration: Reliability branching for the best trade-off in branching/variable selection, the best estimate rule for the search method/node selection, and lastly, duality, fitting lower-bound strategies and dominance relations for pruning. [22]

As for the **performance** of MIP solvers, they have been proven to solve many complex problems (or found a solution within a bound <5%) in feasible computation time [30]. One of the lowest benchmark running times through all the benchmark problems of MIPLIB 2017 [33] was found in 2022 by the best solver at the time (Gurobi). The time was measured in the *shifted geometric mean* (SGM) of the formula: $\sqrt[n]{\prod t_i + s} - s$, which is used to reduce the weight of outliers [28]. It reached the SGM of 91.4 (seconds) with 224 out of 240 instances solved (with eight threads, a maximum running time of two hours per problem and three difficulty levels of the problems). [34] The worst-case running time of B&B would be $\mathcal{O}(Mb^d)$ with b as the branching factor (number of split sub-problems at a node, e.g., 2) and d as the search depth (longest path in the search tree, the number of added constraints necessary). [21] MIP solvers are drastically improving. Over the last 20 years, an average speedup of 1000 times (hardware 20 times, algorithms 20 times) occurred. The benchmark of solvable problems (within six hours) more than doubled in MIPLIB 2017 library, with a mean running time of 104 seconds. [35]

Problems of the traditional MIP optimizers include an exploding time cost [17], which is especially focused on to be improved with this project by using ML. Also, there is no guarantee of tractable optimality and the solvers might lead to near-optimal solutions when given time bounds [24]. There is no universal method for choosing the best strategy [22]. See Table 2.1 for an overview of the stated points on traditional MIP optimization.

General Optimization Problem (MIP)	MILP: Linear objective function under constraints Integer/binary variables Electricity generation planning		MIQP: Quadratic term in objective Energy market clearing
Algorithms	Branch-and-bound (B&B)	Branch-and-cut / Cutting planes	Decomposition algorithms
Solvers	Commercial Solvers: IBM-CPLEX, Gurobi, FICO Xpress		Non-commercial: SCIP
Improvement Techniques	Preprocessing and heuristics	Parallelisation	Fine-tuning of algorithm
Performance	Many complex problems (95% of MIPLIB) in feasible time (100s)		Strong speedup
Problems	Exploding time cost	No tractability guarantee, near-optimal within bound	No universal best strategy

Table 2.1.: Overview of traditional MIP optimization.

2.1.2. MIP and ML

The following literature review subpart discusses previous research projects on ML for MIP. The search string 'Mixed Integer Programming' AND 'Machine Learning' was given to an academic web search tool (Google Scholar) to identify the following papers. The survey of [22] on learning algorithm design strategy for MIP algorithms was used as a baseline. For selection criteria, papers with no ML-specific tests or focusing on a specific use case were excluded. The papers are discussed in chronological order in blogs of similar project contents. The method, data set and results for each paper are discussed. Most discussed papers work on the standard benchmark library MIPLIB 2017 [33].

Different elements of the B&B algorithm (and its improvements) have been subject to optimization through ML over time. Since B&B operates on a tree-like structure of sub-problems, the research here is often connected to ML in tree search. The following elements of the algorithm were learned: Branching strategy, search strategy and pruning, (primal) heuristics and cutting. In **learning the branching strategy**, we learn how to choose which solution variable to branch on (/ which variable the bounding constraints should split next) to create two sub-problems with the highest chance of solving the problem in the shortest time. ML is hereby used to create a branching policy primarily by mimicking the best option in a shorter time or finding the optimal parameters for the best mix. There are different strategies for branching. Choosing the right one can result in better performance, feasibility and efficiency related, as discussed before in improvement techniques under fine-tuning. *Strong branching* is the most exact method but also by far the most time-consuming as it performs various calculations on the variables and problem at hand.

In [36] and [37], they try to leverage the exactness of strong branching while reducing the time cost drastically by estimating the strong branching scores with ML. Imitation learning with regression is applied. So a phase is added to learn the function on randomly generated and optimally solved problems using static (problem-related) and dynamic (based on the state within B&B tree) features. The model results in good branching decisions. Also, the learned approach can improve the performance ratio of solvable problems versus time cost in reliability tests. However, generating training data is difficult as training cannot be abstracted. Only a limited number of samples per variable are available as dynamic features incorporate the state. [36][37] Again mimicking strong branching, in [38] instance-specific on-the-fly training is tested. The framework is usable for other MIP solver strategy decisions, too. It leads to smaller search trees than other heuristics, more solvable instances and in test runs as the fastest for medium/hard problems (compared to CPLEX and CPLEX using strong branching or pseudocost branching). [38]

Further imitation learning of strong branching is applied in [39]. Here, a graph convolutional network (GCN) is used to embed the variable-constraint structure of MILPs. Over all tested benchmarks, it achieves the best performance (most wins, smallest number of nodes and lowest time). Compared to state-of-the-art reliability branching, the approach almost halves the time in medium combinatorial auction-based problems. It provides good generalization to larger instances. However, the training is very computationally expensive (requiring 100,000 instances). [39] To combat the high time costs, they later proposed a hybrid model of graph neural networks and fast multi-layer perceptrons, which resulted in a 26% time reduction of solvers without GPU and solving harder problems. In [19] full strong branching is imitated using a deep neural network that exploits shared structures and works especially well on homogeneous data sets (like electric grid demand). It also incorporates learning primal heuristics to choose the most feasible assignment to branch. The approach significantly outperforms classical MIP solvers (SCIP) and other imitation learning branching approaches. It reaches a 1% duality gap, which measures the difference between the primal and dual solution resulting in the exactness, and is up to ten times faster than tuned SCIP. [19]

In [40], it is claimed that imitating strong branching results in poor performance. This paper uses a reinforcement learning approach to improve previous results with rewards for a few steps. A primal-dual policy network is created. This approach is faster than imitation learning approaches and produces a better dual value for larger problems in their tests. The performance is limited by episode length and performs inefficiently at the beginning. [40]

Rather than mimicking one, different branching strategies can be merged. A dynamic approach for switching heuristics based on pre-learned clustering is proposed in [41]. It outperforms traditional strategy switching methods but conflicts with tree evolution and time efficiency. [41] The tuning of parameters to create an optimal weighting of branching procedures per application can also be learned based on instance distribution. Since it is proven that there is no universally effective parameter choice, the approach is practical and learns a nearly optimal mixture of branching rules per application in the test. [42] The work in [43] steps away from imitating one strategy like strong branching too and focuses on parameterizing the state of the B&B to learn a policy that generalizes across heterogeneous programs. It denotes a test accuracy of 83.7%, which is way higher than GCN on heterogeneous data sets. [43]

Learning the search strategy is discussed in many papers that use machine learning to improve the general tree search mechanism. It chooses which node should be processed next to obtain the solution in the shortest time. They can also be applied to the B&B algorithm.

[22] In papers specifically on MIPs, they often include **learning to prune**, which identifies nodes or sub-problems that do not need to be regarded further. Adaptive node searching was learned by imitation learning in [29]. For a node in a queue, the optimal action is provided by an oracle regarding pruning or search strategy. The project speeds up SCIP with a factor of 4.7 with less than 1% loss in objectives for most problems. More feasible solutions were found given computational limits. It achieves better consistency in performance and acts more problem-independent than comparable solutions at that time. [29] Search and pruning strategy is also connected in [44], which learns node selection policy via imitation learning with a small NN. The best k-solutions are kept, thereby pruning the others. While the solution does not beat optimized SCIP in solving time, it is able to do so with a time limit. It is advised to be seen as a broader approach. [44]

Some papers use machine learning for **learning primal heuristics**, which follows the purpose of finding feasible solutions fast. In [45], it is predicted whether a primal heuristic on a given node would be successful in optimizing the tree search. It boosts primal performance and improves tuned SCIP performance by 6% in primal integral (a measure meant to balance time and quality of solution). The time of finding the first feasible solution lowers by 22%. [45] Some papers focus on a specific type of heuristic such as *large neighborhood search (LNS)*. LNS are used to restrict the search space of feasible solutions. It is chosen between eight powerful LNS primal heuristics during tree search. It creates a time improvement of 2% of solvable problems with primal integral improvement of 18%. [46] In [47], the LNS heuristic is tried to be improved by learning a good neighborhood selector using imitation and reinforcement learning. This produces a decomposition subset that selects and optimizes a subset of constraints. It finds better solutions than Gurobi early on, with Gurobi taking about 2-10 times longer to match solution quality. [47] There are other not MIP-specific works focusing on improving neighborhood search with ML that could be applied. [22]

In [48], they try directly predicting solution values for binary variables. The solution is then used to create a local branching cut. Here a tripartite graph structure is created from the problem and fed to a GCN. This approach improves the primal solution finding performance and accelerates the SCIP solving process by up to 10 times for binary variable-intensive problems. [48]

Lastly, some train ML for **learning to cut**. In [49], the authors use a long short-term memory (LSTM) network to create a reinforcement learning agent creating cuts. It significantly outperforms human-designed heuristics since it provides the fewest number of cuts needed across benchmarks by up to 60% for small problems and solves more with cuts unsolvable/large problems (50%). The model adds meaningful cuts, provides good generalization properties and was advised to be a benefiting subroutine to B&B. It does not lower the time cost of using heuristics. [49] Cuts are not generated but rated in [50]. It ranks the quality of candidate cuts based on the features of the cut and learns the scoring function through feature embeddings and a softmax layer. The function was more effective and had better generalization properties than a heuristic. It creates an average speedup ratio of 12% without accuracy loss, running the fastest in test on benchmarks for small and medium-sized problems (≤ 50 coefficients). [50] An overview of the discussed papers applying ML to MIP, divided into the sub-groups

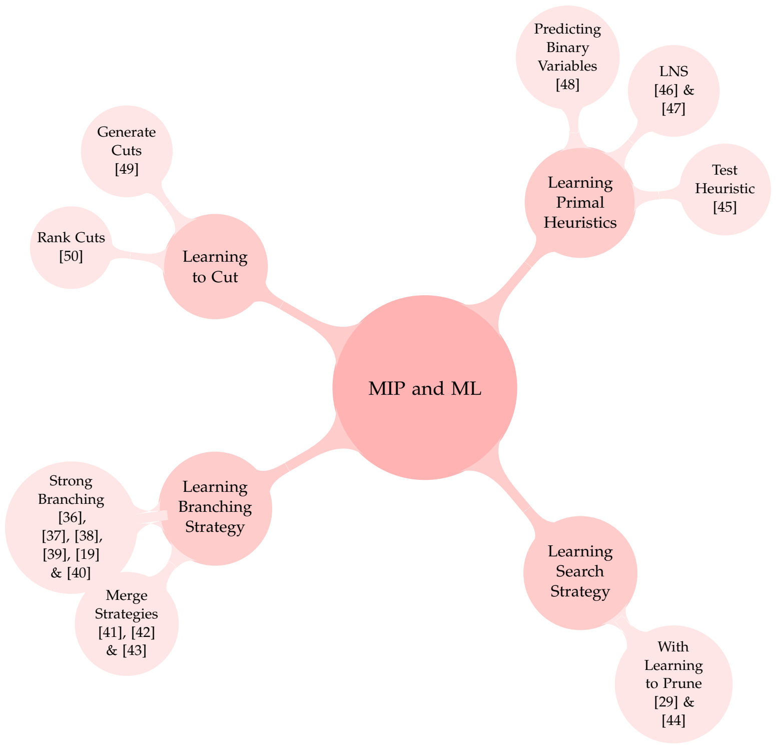
Predicting Binary Variables [48]
LNS [46] & [47]
Generate Cuts [49]
Learning Primal Heuristics
Test Heuristic [45]
Rank Cuts [50]
Learning to Cut
MIP and ML
Strong Branching [36], [37], [38], [39], [19] & [40]
Learning Branching Strategy
Learning Search Strategy
Merge Strategies [41], [42] & [43]
With Learning to Prune [29] & [44]

Figure 2.1.: Overview of Discussed ML for MIP Papers.

of content, is given in Figure 2.1.

2.2. Optimal Power Flow (OPF)

2.2.1. OPF Optimization

Optimal power flow (OPF) is a general optimization problem. Its goal is to find the most economical dispatch of generators that satisfy the demand under the technical constraints of the system. Such constraints would be Ohm's and Kirchoff's laws, loading limits and voltage levels. [51] In practice, this problem is non-linear, non-convex, includes thousands of binary and continuous variables, and thousands of constraints. This is due to the complexity of the energy system and high number of energy generators and consumers. [5]

There are different **models** developed based on needed granularity and included constraints that ultimately influence the solving time cost. In research, usually three algorithms with increasing complexity are being discussed. *Economic dispatch* includes the baseline. The generator cost (cost c times generation volume P_G) is to be minimized under the constraints of generator capacity limits (minimum P^{min} and maximum P^{max}) and power balance (the supply meeting the demand P_D): [5]

$$\begin{gathered} min \sum_i c_i P_{G_i} \\ s.t. \\ P_{G_i}^{min} \leq P_{G_i} \leq P_{G_i}^{max} \\ \sum_i P_{G_i} = P_D \end{gathered}$$

DC-OPF (Direct Current) is mostly used in market clearing and applied in this paper. It considers power flows as well through the admittance matrix B and the phase angle θ. Additionally, transmission line capacity is included with $P_{i,j}^{max}$, compared to the line reactance x. The phase/voltage angles account for the energy flow between the buses, while reactance and admittance (given as susceptance) measure the opposition in the transmission line [6][52]. This algorithm is convex and rather fast solved: [5]

$$\begin{gathered} min \sum_i c_i P_{G_i} \\ s.t. \\ P_{G_i}^{min} \leq P_{G_i} \leq P_{G_i}^{max} \\ B \cdot \theta = P_G - P_D \\ \frac{1}{x_{ij}}(\theta_i - \theta_j) \leq P_{i,j}^{max} \end{gathered}$$

AC-OPF (Alternating Current) is used primarily to optimize operation and control actions. It depicts the full equations by including active and reactive power flow, current, voltage and loss in the constraints. It is non-convex (has no convergence and optimality guarantee), has a high size and high computation time. The full specification of the problem is discussed in [5].

In the **industry**, *EUPHEMIA* is the algorithm to calculate electricity dispatch and prices across Europe in coupled markets. The input for EUPHEMIA is provided by the transmission system operators, which is ENTSO-E in most European countries. [53] In the US, different independent system operators (ISOs) like PJM, MISO and CAISO solve the economic dispatch per region. [5]

The optimization of OPF still faces many **trends** and problems. Even sixty years after the first formulation of OPF, there is still a search for reliable and efficient OPF solving techniques [3]. Internal problems of OPF include incorporating uncertainty, which is explored in some research with security-constrained OPF (SCOPF). There are efforts toward the convexification of the problem and its redesign. Lastly, new solving methods, such as decomposition, influence the implementation of OPF-specific solvers in the industry. [5] Current external trends are also creating hurdles for OPF optimization. Some external problems are the strengthening of renewable resources and the market becoming more distributed with the integration of distributed energy resources, which creates additional complexity in the model. [3] Renewable and distributed resources foster uncertainty and variability in energy realizations and require near real-time forecast updates [7]. Here ML tools can help, as they can also solve complex problems in fast time [3]. See Table 2.2 for an overview of the stated points on traditional OPF optimization.

General Optimization Problem	Optimal Power Flow (OPF): Most economic dispatch of generators satisfying demand and technical constraints		
Models	Economic Dispatch: Minimize generation cost Generator limits and power balance Convex	DC-OPF: Power flow and line capacity Market clearing Convex	AC-OPF: Reactive power, current, voltage Operation and control Non-convex
Industry	EUPHEMIA Algorithm for clearing the European electricity market		PJM, MISO, CAISO Regional dispatch in US
Trends	External: Renewable resources Distributed markets		Internal: Modelling uncertainty (SCOPF) Convexification and redesign New solving methods

Table 2.2.: Overview of traditional OPF optimization.

2.2.2. OPF and ML

As seen in the problems of OPF optimization just discussed, ML can help to obtain a more cost-effective solution by reducing the time and computational burden of solvers. Since the OPF problem is repeatedly solved, a vast amount of historical data is available on the energy system, which is unused so far and offers an excellent basis for ML training. There are examples of successful applications of ML in power systems utilizing the data and strong computational resources. Early studies focused on predicting OPF results directly; recent ones often predict parts of the solution, for example, initial values. They can be used in fitted ML models, which helps guarantee that all model constraints are satisfied. [3][7]

Since OPF is a MIP, the ML and MIP approaches discussed in the first part of the literature review can also be applied. However, they are not specialized for the OPF problem, its structure and data. In the second literature review, OPF-specific experiments are discussed, which form a baseline for choosing an ML-based approach to solving the DC-OPF problem in this project. The literature review was conducted in the same manner, with an online research for a starting point using the search string 'Optimal Power Flow' AND 'Machine Learning'. Two reviews were regarded for input material. [3] surveyed applications of machine learning for optimal power flow and [4] wrote a paper summarizing ML trends in the unit commitment problem. Connected sources were added in a backward research fashion with one-step depth. We choose only papers with a machine learning project and focus on energy dispatch, other applications of ML and power systems, such as creating a control policy, are shortly mentioned. Papers are discussed chronologically within category subgroups of their subject of optimization. These are predicting generation dispatch, predicting warm starting points, predicting active constraints, and finding stability constraints.

For each paper, the method, used data set and the result are regarded. For the data set an energy grid and historical/market/time series data is needed; more on this in chapter 4 of this paper under section 4.1. The IEEE test cases is commonly used benchmark database in the depicted papers. They can be found as IEEE PES PGLib OPF benchmarks in [54] or university databases such as [55] and [56]. They include grids of different sizes abbreviated "IEEE-(number of buses)" in this paper. If they use multiple sizes, they are trained and tested with separately. The databases and data collection methods in the papers are again inspected in section 4.2 for applicability to our project.

Papers that are **predicting generation dispatch** offer the classical application of ML for OPF. Here the output of the OPF optimization is predicted, which includes energy generation at each generator and transmission at each network point. A fast approximation proxy of OPF cost was generated with different supervised learning approaches that solve the AC-OPF model in [57]. In this work, they trained a naïve Bayes, logistic regression, decision trees and a random forest classifier to classify the problem as feasible. Then they used logistic regression and a neural network (NN) to estimate the cost. They used a 5-bus toy model, IEEE RTS-79 and IEEE RTS-96 as grids. Then they generated the load by scaling average demand with a uniformly drawn random value (within min/max) and did not include cost or generator historical data. They achieved an approximation of less than 1% error on average while running several magnitudes of time (10^{-5}) slower. NN has the highest exactness, lowest prediction time and highest training time. [57] In [58], feasible solutions for AC-OPF with small optimality gaps are predicted with a neural network solving a restricted MIP. IEEE-118,57,39 and load sampled with a Gaussian distribution are used to train each net. They encounter low infeasibility (10^{-8}) and small optimality loss (10^{-3}). The algorithm is quite fast, with about 15x time improvement. [58]

The work in [58] inspired [59]. They build a deep neural network for solving DC-OPF as mapping from load inputs to dispatch and transmission decisions. They use uniform sampling against overfitting. In addition, they propose an approach to reducing the mapping complexity because they leverage the problem structure by predicting the active powers of each bus not individually but through a scaling factor. They use IEEE-30, 57, 118, 300 and scale load randomly for historical data. The model always generates feasible solutions (also without post-processing) within negligible optimality loss. It also sped up computation time by two orders of magnitude. [59]

A slightly different approach is used in [60], where security-constrained generation dispatch is learned. They use a local optimal generation prediction mechanism from local features or measurements and train a random forest on a multi-target regression problem. As a data set GO Competition's test data is used which creates scenarios around IEEE cases, specifically IEEE-14 and IEEE RTS-96. They conclude that the optimal local generation dispatch is predictable with local features with high accuracy, as they achieve the same R^2 for local and global features of around 0.9. R^2 measures the explainable variance by dividing the mean squared error by the variance of the target data. A high sample size (>1000) is needed to combat overfitting. [60]

Since directly predicting the generation dispatch can lead to sub-optimal solutions, some papers proposed using ML for **predicting warm starting points**. *Warm starting points* are (partial) sub-optimal solutions, that can help make the MIP optimization process faster when applied as starting solution in the beginning [61]. In [61], the power generated at each generator is predicted within the DC-OPF model from the demanded load and generation cost. A gradient boosting regression is applied. For data, the IEEE-30 case is used, from which historical generator data is scaled and historical PJM demand data is added. Over

90% of the predictions lay within 5% of the true solution, but 60% of the predictions violated one of the network constraints. So the method was converted to a starting point prediction, reducing computation time by up to 30%. [61] In [62], they train a random forest learner to predict the solution of AC-OPF from historical loads. The solution is used as a better starting point for solvers. The predictions of multiple tree regressions are averaged on generation and voltage. IEEE test networks are used. The method outperforms traditional warm start or flat start methods as it leads to fewer iterations. [62]

Like in ML for MIP papers, some researchers also focused on predictions on constraints when inspecting OPF, especially on **predicting active constraints**. *Active constraints* is a set of constraints (or indexes of the solutions variables) where one side is equal (=) and (not less or equal $\leq$) to the other side of the equation. The solution values can then be derived efficiently. [51] describes a statistical learning approach to learning important bases. They are then given to an ensemble control policy as features to solve DC-OPF. The IEEE-24,75,162 and 300 grids are used in separate tests. They figure that only few bases/constraints are relevant for the system. Policies from ten bases can already obtain the optimal solution with high probability (for 10/14 cases, optimality on bases >0.99). [51]

Classification of active constraints for uncertainty realization in DC-OPF is also predicted in [7]. A NN is trained with multiple fully connected layers, ReLU layers and softmax in the end, as the classifier predicts the top k-active sets. This reduces the complexity and can again be given to an ensemble policy for prediction. IEEE PGLib is used as training data in grid sizes 24, 73, 162 and 300. Through the high computational efficiency, the model is amendable for real-time applications. The top 3-classifier can achieve a 99% accuracy in active set prediction and >95% prediction accuracy for dispatch. More training data and lower complexity models improve model performance. [7] In [63] iterative statistical learning method is applied to find the set of active constraints in the DC-OPF model. A streaming algorithm determines the set, then a proof detects whether the method is applicable and with an ensemble policy, the optimal solution of the reduced problem is derived. IEEE PGLib test cases are used for training. The model requires long running times for complex systems. A normal distribution of parameters is helpful and therefore, the system is applicable to engineered systems with repeated solving for varying input parameters. [63]

A more specific set of active constraints are *umbrella constraints*. Umbrella constraints include the few necessary and sufficient constraints to cover the OPF feasible solution. *Joint chance constraints* satisfy multiple constraints simultaneously. In [64] a method is developed to predict whether a constraint is in the set of umbrella constraints in in SCOPF. They train a wide single-layer NN minimizing the size of the set with a strong penalty for false negatives. Applicability is tested with IEEE-118. They achieve a 0.024 false negative error and an overall 0.33 average test error (meaning 9 extra conditions). [64] Filtering out active constraints in AC-OPF is the goal in [65] with improvement in [66], where they focus on joint chance constraints. They use a support vector classifier to classify active and inactive constraints in the IEEE-37 system. The method is slightly less conservative than Boole's inequality (traditional method) and increases computation efficiency. [65][66]

Instead of finding active constraints, some papers focus on **finding stability constraints**. In

[67], they analyze transient stability constrained OPF (TSOPF), which has additional security constraints for contingencies. They simplify this problem into a conventional OPF through a (multi-layer feedforward) NN that predicts the critical clearing time constraint to eliminate dynamic parameters. The method reduces the computational burden through complexity reduction. It reduces convergence cost by up to 99 %. [67] In [68] they focus on guaranteeing power system security. They create a decision tree for learning decision rules to optimally decide on power flow while guaranteeing security. The decision rules are then integrated into SCOPF formulation as security constraints. The experiments show that $n-1$ security failed to be ensured, but knowledge can be created with this approach. [68] A revisit in [69] shows cost savings when the decision rules were extended to AC-OPF.

Other papers focus on the **unit commitment** component. The *unit commitment (UC)* problem is a sub-problem of OPF. UC regards which generators to start/shut down, so solving for binary solution variables, while the economic dispatch, decides on the actual power output of the generators, a continuous variable. For exactness, power optimization should consider both. [70] In our mathematical problem formulation, we will include both binary and continuous variables, see section 4.1.

UC is often solved by learning clusters. In [71], the nearest neighbor algorithm is applied to approximate market clearing. At test time, the nearest scenario is chosen to approximate the UC solution using the weighted distance of forecasts and topology. IEEE RTS-79 and 96 are used for data. The model requires long initial preparation as much data is needed. It achieves good time reductions but could only be used as a proxy for approximation. The clusters are very season-dependent, possibly because of the availability of renewable resources. The correlation between accurate and predicted costs is 0.96. [71] Another clustering approach for the security-constrained UC case is discussed in [72]. They extract patterns from previously solved instances to predict redundant constraints and initial solutions. The k-nearest neighbor algorithm is used as well as support vector machines (SVMs) to separate the instances. The data gird is self-generated with energy market data from PJM and MISO as market data. The solution is, on average, 4.3x faster with an optimality guarantee. A large number of historical instances is needed to create a fitting distribution. [72]

In [73] likewise, security-constrained UC instances are classified into easy or hard. Then it is chosen whether to solve the problem with a heuristic, which sacrifices solution quality, or trained SVM or NN. Random forest performs best with an average speedup of 1.4x and inaccuracy of 0.07%. [73] An overview of the discussed papers applying ML to OPF, divided into the sub-groups of content, is given in Figure 2.2.

There are other applications of machine learning for power systems discussed in research, but they move further away from OPF. For example, some papers learn a control policy for electronic power inverters, which maintain voltage at a customer, from local historical data. [74][75][76] Others regard forecast errors [77][78][79].

The literature review shows that ML on OPF can result in successful applications, meaning high speedup with low accuracy loss. Mainly ML can be applied to direct solving of the dispatch, simplifications through learning on constraints or algorithm design. When learning to select active constraints, false negatives are problematic. Research on optimizing general

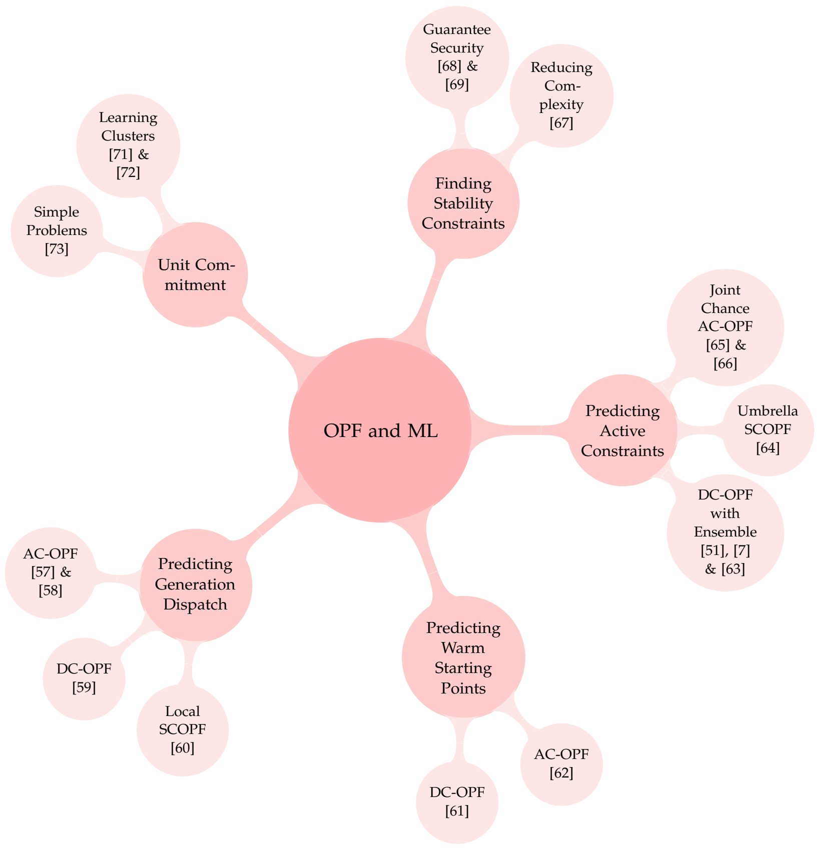
OPF and ML
Finding Stability Constraints
Guarantee Security [68] & [69]
Reducing Com- plexity [67]
Unit Com- mitment
Learning Clusters [71] & [72]
Simple Problems [73]
Predicting Active Constraints
Joint Chance AC-OPF [65] & [66]
Umbrella SCOPF [64]
DC-OPF with Ensemble [51], [7] & [63]
Predicting Warm Starting Points
AC-OPF [62]
DC-OPF [61]
Predicting Generation Dispatch
AC-OPF [57] & [58]
DC-OPF [59]
Local SCOPF [60]

Figure 2.2.: Overview of discussed ML for OPF papers.

MIP solving provides methods for learning optimal algorithm parameters on the search tree, which can be applied to OPF but is not specific to that field. Since security and optimality cannot always be ensured, the review advises using sub-optimal OPF prediction solutions as a warm starting point. The B&B algorithm or post-processing rules can be fed with the ML results to ensure guarantees and constraints are met. They might also improve the results further. This application already leads to high time savings and makes even unreliable ML predictions useful. Different standard ML algorithms can be applied, while neural networks are discussed the most in newer projects and often lead to the best results. Regarding the databases, most papers chose the IEEE grids and, in a few cases, real-life data was extracted from the energy operator databases of PJM or MISO. These insights will be regarded in our future steps.

The papers so far have three main limitations, which leave room for further research before they can be integrated in real-world operations. First, they mostly run numerical tests on small academic test systems. ENTSO-E identifies more than ten-thousand entities in their market [80], while IEEE grid sizes from five to a maximum of 300 nodes were used in tests. Second, the artificially generated data distribution does not capture the variability of real-time operations. [9] Additionally, variable size grids are barely regarded, but power grids are dynamic. [10] In the work of this paper, we will search for realistic data sets, create a data set from real market data and explore the dynamic size of grids thoroughly.

3. Foundations

3.1. Relevant Electricity Market Foundations

3.1.1. Electricity Market

The insights into the electricity market are adapted from the article about the market design [2], which focuses on American markets. The goal of the *electricity market* is to provide electricity to all consumers reliably and at the least cost possible. We have two types of efficiency objectives short and long-term. While the latter discusses the best investment strategy for new resources, we look into the short-run efficiency, which marks the core optimization problem of the energy market. This short-run objective includes the optimal scheduling of resources. There are two types of scheduling markets, a day-ahead market for planning and a real-time market for binding dispatch. The markets are designed to balance supply and demand at all times as well as fit thousands of resource and network constraints. Additionally, the right incentives have to be created for generators and investors.

The specific electricity market design depends on the region. For parts of the US, the market is run by the regional *independent system operator (ISO)*, like PJM and MISO. The integrated market is favored in North America, which follows a central optimization. In Europe, an exchange-based market was established, focusing on day-ahead and real-time trade between companies to clear the market. Electricity markets are artificially designed, and they have improved through good governance and technological progress. Nowadays, electricity markets are spot markets. So the generation, consumption and price are specific to time and location.

Electricity markets face new challenges. Electricity is at the very center of the climate change debate. The transformation towards handling renewable resources is non-trivial, as they are intermittent sources with zero marginal cost and no inertia. This creates additional uncertainty and might force faster adjustments to the market design. Additionally, the energy grid is getting more and more distributed, which adds technological complexity. Other changes regarding electricity use and innovations, like more price-dependent demand responses through smart houses and battery storage, will impact the electricity market.

3.1.2. Electricity Market Core Problem

There are two core objectives in the electricity market. *Short-run efficiency* is a direct optimization problem for the best use of the existing resources. It includes a unit commitment and a real-time (economic) dispatch problem. The number of resources and technical constraints (of the energy resources and the system) make this problem complex. Given truthful bids and no distortion, optimal global welfare can be achieved. Real-life distortions such as emission externalities, subsidies and regulations might make this approach naive. However, this clear structure helps the other objectives of the electricity market, such as simplicity, transparency and fairness.

To ensure *long-run efficiency*, the market design needs to create the right incentives for long-term investments, primarily through the spot prices. It needs to comply with the reliability requirement, which can be ensured through reserves and scarcity prices.

The market is a *spot market* or nodal market with spot/node-specific decisions. Since the energy prices should reflect the transmission constraints to create good incentives, the price relates to its marginal value at a specific time and location.

The market includes some core players. There is a wholesale market of *generators*, which are independent of electricity utilization and make their own investment decisions. The *transmission providers* will distribute the energy of the wholesale market, ultimately leading to higher competition between generators. Next, the *service providers* or load-serving entities serve and compete for customers creating a retail competition. Their primary goal is to offer minimal energy prices. They may thus utilize price reduction mechanics, such as buying energy when the price is low. Finally, the *distribution companies* distribute electricity to specific customers with low-voltage lines. Distribution companies often hold a monopoly.

3.1.3. Market Clearing and Dispatch

The decision of market participants and market objectives are included in the central optimization of *market clearing and dispatch*. This optimization requires advanced techniques and technologies. Thousands of computer servers are needed to run the systems. Two markets based on time proximity are optimized, the day-ahead and the real-time market. In the *day-ahead market*, the participants submit their *offers* to sell and *bids* to buy energy as well as reserves for the next day. The generators' offers include start-up cost, minimum energy cost (the lowest production level) and the energy offer curve. The offer curve depicts the marginal cost in pairs of price and quantity. Other technical parameters, such as capabilities, create an efficient commitment. Participation is voluntary but financially binding. When the deadline for bids and offers is met, a day-ahead supply schedule is created with spot prices to meet the demand. This schedule helps participants to coordinate production.

The *real-time market* chooses the dispatch and prices during the operating day. Participation is mandatory. It creates a binding physical dispatch and real-time prices within that time interval. The late physical binding might foster gambling between the two markets. Generators need to submit an energy offer curve and consumers submit energy bids or, more often, a bid curve that can show their demand response. The optimization is a security-constrained economic dispatch (SCED) and returns real-time dispatch instructions with prices. It should be executed around every five minutes.

Optimization aims to maximize the total gain as the difference between value and cost while meeting all network and resource constraints. The optimization outcome is a competitive equilibrium with quantities based on shadow prices. These settlement prices are *locational marginal prices (LMP)* as marginal generating cost for megawatt at the respective location and hour. For transmission between different locations, congestion rent is deducted, which creates additional congestion revenue for the transmission providers.

The problem includes many non-convexities, such as start-up costs and capacities. The optimization problem is, in theory, a mixed-integer programming problem. Advanced hardware and software are necessary to solve this problem. Optimization cannot always result in an equilibrium. The solvers will return optimal qualities and approximate prices in this case. After receiving the outcome, participants will submit an adjusted operating plan. The system operator will inspect them to ensure transmission security and reliability in unit commitment.

3.1.4. European Energy Market

The explanations so far focused on the North American market, from where we will also take the data. We will now point out some specifics of the European market, where we are located. The higher fragmentation of the European markets, due to the combination of different national markets, poses some challenges. The LMPs are less exact than in completely centralized markets since a price is often set per country (from a European perspective). This missing granularity also makes the transmission congestion inefficiently priced. Real-time European markets additionally include a third voluntary balancing market. This separation into three markets leads to less reliable real-time prices and higher demand for trading throughout the day. It requires auctions in shorter optimization time spans, while the complexity of the market in general increases. [2]

3.2. Relevant Machine Learning Foundations

3.2.1. Machine Learning (ML)

Machine Learning (ML) is a field in computer science that solves complex problems, which would otherwise be difficult to specify with conventional techniques, using specific algorithms and approaches. Rather than directly defining rules to solve the problem through expert systems, statistical techniques are applied that approximate a *model* from old *data points* within a *training* phase. For this, we require a data set with different data points. In *supervised learning*, these data points are *labeled*, meaning the desired behavior for the *target*/solution (variable) is given. In the *test* phase, we can then predict the solution for a new data point using this model. [8]

There are three problems that the ML model can solve. *Classification* assigns a data point one or more pre-defined categories. *Clustering* puts data points with common properties into a group/cluster. The clusters are not defined beforehand, which makes this an *unsupervised* task. Prediction or *regression* forecasts a future (continuous) value based on historical data. OPF with ML is a regression case. [8]

ML models tend to be more accurate than human-defined rules as the training can consider all data points without human bias. However, how or why the problem is solved, often referred to as *interpretability*, remains to be seen, especially for neural networks. Lifting this black box is an area of active research. Some modern examples of highly successful ML models are language translation or image generation, which raised the general public's attention. ML can be considered a sub-field of artificial intelligence (AI), whose general goal is to make computers intelligent. [8]

3.2.2. Linear Regression and Gradient Descent

The parts on regression are adapted from [8]. With regression, the relationship between the different dimensions of the data onto the target variable is learned. *Linear regression* allows predicting a continuous variable, like the production volume of a machine. *Logistic regression* predicts a binary variable, like whether a machine is turned on or off. In regression, the model is represented as a function $\hat{h}$, where the hat refers to optimal within the capacity of the model. Each data point adds a row, while the input dimensions or variables per data point translate *features* to the columns of X. Over the learnable *parameters* θ, X can predict the target variable Y as $\hat{Y}$. For linear regression in matrix representation, this results in the following:

$$\hat{Y} = h(X) = \theta^T \cdot X$$

The goal is to minimize the *empirical risk function* or *cost function* over selecting the parameters. The function includes the *loss*, which is the difference between the predicted value $\hat{Y}$ and the label Y. Within linear regression, we square the difference to receive a positive error and divide by two for simpler differentiation. In matrix representation, the formula would be:

$$\mathcal{R}(\theta) = (Y - X\theta)^T (Y - X\theta)$$

This function is also referred to as the *mean squared error (MSE)*. It can be solved for the optimal values of the parameters θ by taking the derivative with respect to θ. Since the transpose of X might be quite computationally intensive, especially for many data points, the parameters can also be learned with *Gradient Descent*. Now, the loss is calculated by the total error over all n data points:

$$\mathcal{MSE}(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}^i - y^i)^2$$

Which essentially plots as a convex parabola. θ can be learned as the slope of the parabola via the partial derivative. The derived function resulting in the optimal parameter for all m features transforms to:

$$\theta_j = \frac{\partial \mathcal{R}}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h(x^i) - y^i) \cdot x_j^i \qquad \forall j = 1, ...m$$

Referring to the index, this means the slope and optimal solution of θ can be determined at each data point. For each data point and thereby iteration t, the parameters need to be corrected:

$$\theta_j^t = \theta_j^{t-1} - \alpha \frac{\partial \mathcal{R}}{\partial \theta_j} \qquad \forall j = 1, ...m$$

Since the function is convex, a closer approximation to the optimal parameters will return a decrease in the slope along the parabola, referred to as *gradient descent*. α implies the *learning rate*, which influences divergence and convergence time. *Convergence* measures whether the iterations bring the solution closer to the optimal. α is a *hyperparameter*, which means its value needs to be *tuned* or experimented with for optimal effect.

For faster computations but never fully reaching the optimal solution, *stochastic gradient fescent* can also be applied, which stochastically/randomly selects a single data point to define the slope on:

$$\theta_j = \frac{\partial \mathcal{R}}{\partial \theta_j} = (h(x^i) - y^i) \cdot x_j^i \qquad \forall j = 1, ...m$$

3.2.3. Logistic Regression

For logistic regression, we additionally apply a *sigmoid* function to scale the predicted value within 1 and 0:

$$\hat{Y} = h(X) = g(z) \qquad where \qquad z = \theta^T \cdot X \qquad and \qquad g(z) = \frac{1}{1 + e^{-z}}$$

The loss function is adjusted with the logarithmic function, so the loss is very high when the predicted outcome and label do not match. The loss function for logistic regression is often called *log loss*:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} y^i \cdot log(\hat{y}^i) + (1 - y^i) \cdot log(1 - \hat{y}^i)$$

Through partial differentiation, both loss functions result in the same derivative. However, h and its derivative deviate.

3.2.4. Overfitting and Regularization

The model will not lead to the ground truth but will be matched closely to the given data points. If matched too closely, it may lead to too complicated solutions that incorporate outliers, ultimately reducing the model's prediction quality. This phenomenon is called *overfitting*. The concept of *regularization* is applied to combat overfitting in regression. It introduces the hyperparameter λ, which limits the contributions of outliers. A penalty term is added to the loss function, which includes the squared magnitude of the coefficient scaled by λ. Through derivatives and reordering, it can directly be included in the parameter update:

$$\theta_j^t = \theta_j^{t-1}(1 - \alpha \frac{\lambda}{m}) - \alpha \frac{\partial \mathcal{L}}{\partial \theta_j} \qquad \forall j = 1, ...m$$

This method is called *L2 regularization* or ridge regression. Another regularization method is L1 regularization / Lasso.

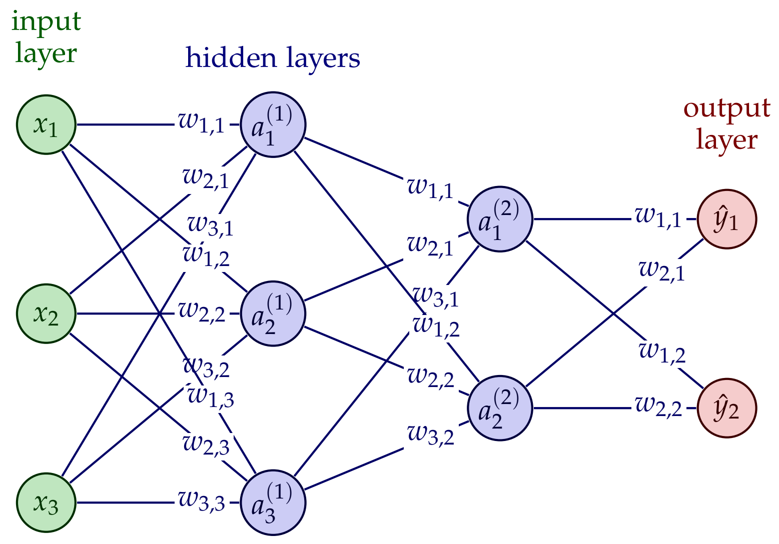

Figure 3.1.: Neural network architecture adapted from [81].

3.2.5. Validation

To validate the model, usually, 70% of the data points are chosen to be in the *training set*. These data points will be used in the learning, so Gradient Descent. The other 30% will be in the *test set*, on which the model is applied to test its correctness. A third *validation set* can be taken from the training set to evaluate the model during training and this set can be used to tune hyperparameters.

3.2.6. Neural Networks (NN)

Neural networks (NN) are networks of interconnected *neurons* or perceptrons within *layers*. The idea was to mimic the working of the brain. From a modern-day standard, however, the workings are quite different. Neural networks can solve more complicated problems by combining different regression models. Each neuron takes in multiple inputs from the layer before and produces one output, which will be forwarded to connected neurons on the next layer. The first layer will be the *input layer*, followed by an ambiguous number of *hidden layers* and the last layer will be the *output layer*. Through this setup, neural networks are also referred to as *multilayer perceptrons*. [8] An outline of an example NN architecture can be found in Figure 3.1 adapted from [81].

The input layer represents the different input variables per node. The hidden layers, create the weighted average $z_j^{(l)}$ from the output of the previous layer $a_i^{(l-1)}$ reaching this neuron weighted by the respective *weights* $w_{i,j}$ and add the neurons bias $b_j^{(l)}$. Then apply an *activation function* $h^{(l)}$ is applied to calculate the output of the neuron $a_j^{(l)}$. This results in the following formulas for a hidden and output layer in scalar notation: [82]

$$z_j^{(l)} = \sum_i w_{i,j}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = h^{(l)}(z_j^{(l)})$$

Activation functions should introduce non-linearity to make the representation of complex (non-linear) models possible. Activation functions include sigmoid or *rectified linear unit (ReLU)*, which is now widely used and combats the vanishing gradient problem. The ReLU function looks like this: [82]

$$h(x) = max\{0, x\}$$

The output layer has the same functions, but instead of an activation function, an output function needs to be applied that shapes the output correctly into y. The same functions h are applied as in regression depending on the nature of the output as well as the same risk function to be minimized. The training works parallel to the regression. Gradient descent is applied to find the optimal values for the weights W and the biases B. Through the *chain rule*, the partial derivatives throughout the system can be calculated. The chain rule combines partial derivatives as follows:

$$\frac{\partial Z}{\partial X} = \frac{\partial Z}{\partial Y} \cdot \frac{\partial Y}{\partial X}$$

Starting from the risk function, the partial derivative is calculated for each parameter through the last until the first layer. The update of each follows the gradient descent principle. This process is called *backpropagation*.

3.2.7. Convolution

Convolution is a technique to extract and aggregate important features needed to identify the target class. Its traditional use case is object detection. Within convolution, a rectangle called *kernel* slides over the image or input sequence, computing the dot product of the filter and the overlapped portion of the image. A high overlap of characteristics implies the detection of the searched object. [8]

Given the input X, our kernel W of size *k x k* and our output or *feature map O*, the convolution or cross-correlation operation can be defined as: [83]

$$O_{i,j} = (X \cdot W)_{i,j} = \sum_m \sum_n X_{i+m,j+n} \cdot W_{m.n}$$

A depiction of one calculation of this function can be taken from Figure 3.2.

The kernel weights W are learned. Kernel size defines how many filter outputs are influenced by a single input. A smaller size introduces sparsity. Kernel parameters are the same for different input layers, which leads to *parameter sharing*, which reduces (space) complexity and causes equivariance to translations (like image rotation). Additional kernel parameters are padding and stride. *Stride* defines how many positions are skipped between calculations. This can reduce computational cost and down-sample the output. With *padding*,

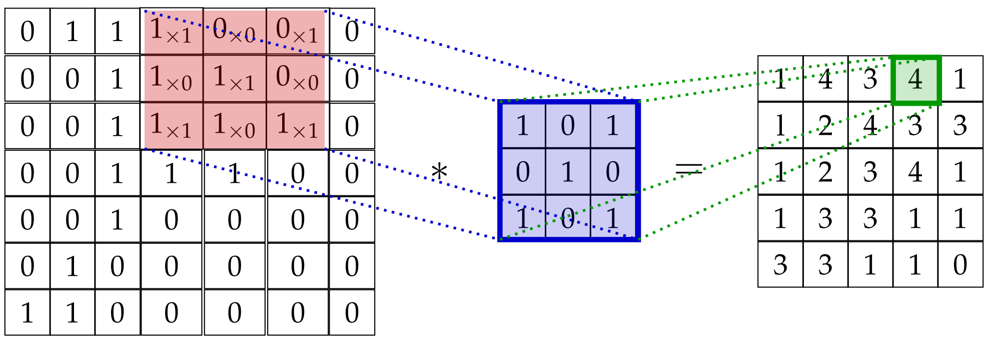

Figure 3.2.: Convolution example adapted from [84].

the input size is increased by, for example, adding zeros to the outer edges. [85] Both parameters influence the output size, which will be for height h (width analogous): [83]

$$h_O = (h_X - h_K + 2p)/s + 1$$

It is also possible to define different kernels per input layer. The collection of kernels would be referred to as *filter*. The output depth after one filter is thereby given by the number of kernels K. The different two-dimensional output matrices per depth are called output *channels*. If the input has depth or multiple two-dimensional inputs, e.g., one for each primary color in images, a different kernel can process each input channel and the resulting matrices can be summed up to be reduced. [85][83]

A convolutional layer should include the convolution filter just discussed, an activation function to introduce non-linearity and a pooling layer. The parameters are learned with the same neural network principles: Chain rule and (stochastic) gradient descent. [85][83]

The objective of *pooling* is to reduce the size of feature maps, thereby decreasing computational costs to train and extract dominant features. They use a summary statistic on the output and work similarly to filters, but there is no parameter to learn. Examples of summary statistics are a *max pooling layer*, where the kernel will select the maximum value within the area, or an *average pooling layer*, where the average of the area would be created. [85] The output height would be (width analogous): [83]

$$h_O = (h_X - h_K)/s + 1$$

There is one pooling kernel per output channel, so the depth remains unchanged.

3.2.8. Graph Neural Networks (GNN)

Machine learning approaches so far assume their data points to be *independent and identically distributed (iid)*. Most real-world data sets, however, show dependencies through *temporal* relations or *spatial*/topology-related dependencies. [86] Graph data can not be depicted in Euclidean space, meaning it cannot be represented in $\mathbb{R}^n$ or as an n-dimensional linear space without information loss. These dependencies challenge traditional neural networks. [10]

A simple *graph* can be defined as $G = (V, E)$ with the set of *nodes* V and the set of *edges* E. Edges are usually given in an *adjacency matrix* $A \subseteq \mathbb{R}^{V \times V}$ with binary values or weights. They are also represented as the *neighbors* of a node $N(v) = \{u_i \in V | (v_i, u_i) \in E\}$. [86][10] An extension for node and graph attributes can be denoted as $G = (V, E, X_V, X_E)$ with $X_V \in \mathbb{R}^{n \times c}$, where n are the number of nodes and f the number of node features, and $X_E \in \mathbb{R}^{m \times c}$, where m are the number or edges and c the number of edge attributes. [10]

In *graph neural networks* essentially, it is learned how to embed the information of the neighboring nodes into the *hidden representation h* of each node. *Differentiable message parsing* is the first method. Within each step/layer k, the information of the neighbor k steps away is included. For each node, the messages from all neighbors are aggregated, for example, in the following averaged fashion. In the formulas, M and U are differentiable functions. The parameters W, Q, L, b and p are trainable parameters at each layer: [86]

$$m_v^{(k)} = \sum_{u \in N(v)} M(h_u^{(k-1)}, X_{v,u}) = \sum_{u \in N(v)} \frac{1}{deg_v} (W^{(k)} h_u^{(k-1)} + Q^{(k)} X_{v,u} + b^{(k)})$$

Then the hidden representation of the node needs to be updated, for example, using a ReLU function for non-linearity: [86]

$$h_v^{(k)} = U(h_v^{(k-1)}, m_v^{(k)}) = relu(L^k h_v^{(k-1)} + m_v^k + p^{(k)})$$

In a usual NN fashion, the output function and risk function fitting the output space must be added to train the network. [86] The trained parameters are reused across nodes and nodes act as data points compared to traditional NNs.

In later research GNNs usually refer to *graph convolutional networks (GCNs)*. It is an alternative view on the message parsing framework as a graph convolution, where the neighboring pixels are the spatial neighbors in the graph. The local features of each node are updated with a convolution of the spatial neighborhood. Given I_N referring to the identity matrix, reformulations to matrices result in the following formula: [86]

$$X_i^{(k+1)} = X_i^{(k)} + \sum_{(i,j) \in E} X_j^{(k)} = (I_N + A) X^{(k)}$$

The impact of the features on the hidden representation will be weighted by the learnable parameters per step $W^{(k)}$. The term can be normalized through the Laplacian $L = D - A$ where D is the degree matrix $D_{i,i} = \sum_j A_{i,j}$ (and *0* otherwise). The normalized Laplacian would be $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$. The update of the hidden representation within a graph convolutional layer can be calculated with the following function, where g is a non-linear (activation) function, $\tilde{A} = A + I_N$ and $\tilde{D}$ is computed with $\tilde{A}$: [87]

$$X^{(k+1)} = g(\tilde{A} X^{(k)} W^{(k)})$$

The normalized function translates to:

$$X^{(k+1)} = g(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X^{(k)} W^{(k)})$$

Multi-dimensional edge features X_E are not a standard in the literature. Through different transformations, they can be incorporated into the embedding of the connected neighbor H_V. We will address this gap in chapter 5.

3.2.9. Recurrent Neural Networks (RNN)

Other non-iid and non-Euclidean data are sequential data, as there are time dependencies within data points. Recurrent neural networks (RNNs) are built to make decisions, set labels or predict forecasts based on historical states. A classical application is natural language processing, where such sequences are sentences and one word is embedded as one data point within the sequence. To keep the context of the embedding, position-based relations between the words must be upheld. RNNs remember past data and process the current embedding on the information before (or after). [8]

RNNs introduce time. So the input data $x^{(t)}$ is relative to t, which could be the time step or generally the place in the sequence. The hidden state h depends on the hidden state before and its external signal x: [8]

$$h^{(t)} = g(h^{(t-1)}, x^{(t)}, \theta)$$

We incorporate learnable parameters θ for the weight of the previous states $h^{(t-1)}$ and the external signal $x^{(t)}$. Different architectures can be implemented depending on the number of input and output data points throughout the time steps. We regard a multiple-input architecture with one output at the end. The architecture and its parameters can be taken from Figure 3.3.

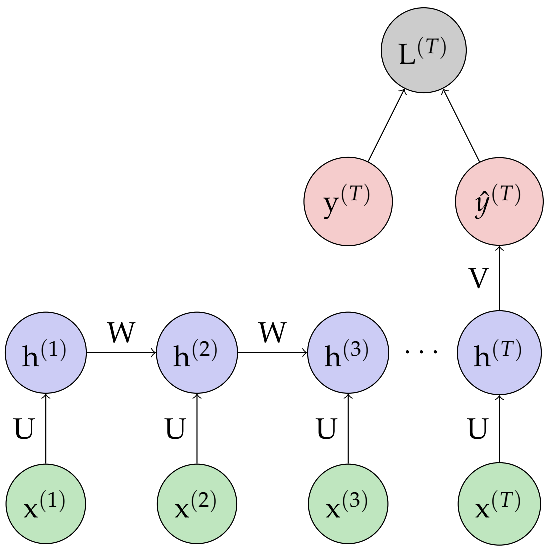

Figure 3.3.: Recurrent neural network architecture adapted from [88].

We use an activation function g like *tanh* and include the weights W and biases b into the computation of the hidden state h and the output o. We apply the proper regression output function g. Then compute the loss with the target variable to start the backpropagation. The internal functions would resemble: [88]

$$h^{(t)} = g(Wh^{(t-1)} + Ux^{(t)} + b) = tanh(Wh^{(t-1)} + Ux^{(t)} + b)$$

$$o^{(t)} = Vh^{(t)} + c$$

$$\hat{y}^{(t)} = g(o^{(t)}) = \sigma(o^{(t)})$$

With this architecture, it is likely to run into the *vanishing gradient* problem and additional efforts are needed to depict *long-term dependencies* accordingly. The *long short-term memory (LSTM)* network adds a *cell* state C_t and *gates*, which define what information should be let through. We have one forget gate f_t, an input gate i_t and an output gate o_t. The formulas are more extensive than traditional RNNs. Let $[\cdot, \cdot]$ be concatenation, σ the sigmoid function and $\times$ piecewise multiplication: [88][89]

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_c[h_{t-1}, x_t] + b_c) \\ C_t &= f_t \times C_{t-1} + i_t \times \tilde{C}_t \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \times tanh(C_t) \end{aligned}$$

The *gated recurrent unit (GRU)* combines forget and input state into a single update state, then merges cell state and hidden state. So it is simpler to train than LSTM. Both LSTM and GRU are widely used. The GRU formulas translate to the following: [88][90]

$$\begin{aligned} z_t &= \sigma(W_z[h_{t-1}, x_t]) \\ r_t &= \sigma(W_r[h_{t-1}, x_t]) \\ \tilde{h}_t &= tanh(W[r_t \times h_{t-1}, x_t]) \\ h_t &= (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t \end{aligned}$$

4. Data

4.1. Data Model

4.1.1. Data in DC-OPF

The motivation of this paper is to test whether the European market clearing, usually solved with EUPHEMIA, can be accelerated by applying ML. Due to increasing uncertainty with more distributed and renewable resources, the clearing should be done every twenty minutes and the interval should be reduced further. The optimization process however, can take longer than that. We will regard the primal problem, to solve the optimal energy dispatch. It can be modeled with the **DC-OPF** problem.

For the OPF problem, we regard the electric grid which essentially is a graph. We have two types of actors: The *consumers* (or buyers), which require energy, and the *generators* (or sellers), which produce energy. Each is connected to a *bus*, a node in the grid. The problem is structured into four equations:

- **Objective:** Global welfare through minimizing cost
- **Generator Operation Limit:** Minumum and maximum generation capacity of each generator
- **Power Flow Balance Equation:** Power input equals power output at each bus
- **Power Injection:** Limit of energy transmission per line

The objective is to minimize the generator cost, which will ultimately lower the energy prices for the consumers, resulting in *global welfare*. This is limited by three constraints. The first is the *generator operation limit*, stating that the generated amount cannot exceed each generator's maximum and minimum generation capacity. Second is the *power flow balance equation*. It requires that the total power injection into one bus equals the power consumption at the bus and power injections into neighboring buses from that bus. Lastly, the *power injection* into each transmission line cannot exceed its capacity. [59]

We have two input data types that need to be given to the optimization. One is the *market data*, which changes for each optimization. It is also referred to as time series or historical data. It includes the power export/demand of each consumer, the incremental cost of each generator's generated power, and the generators' minimum and maximum operation limits. Then there is the *nodal data*, which stays the same until the energy grid eventually changes. Changes are considered in this paper as we work on varying grid sizes to account for real-life conditions. The nodal data includes the electrical properties of the line currents in susceptance (or admittance matrix) and reactance, as well as the overall structure of the net (e.g., number of generators and consumers). Through the optimization process, the DC-OPF model retrieves two solution variables: The voltage angle of each bus, which accounts for the energy transmission between nodes, and the power produced at each generator. [59] How the variables translate into the mathematical model can be extracted from Table 4.1.

DC-OPF model:	Minimizing cost under generator operation limits and power balance
$min \sum_i c_i \cdot P_{G_i}$ $s.t.$ $P_{G_i}^{min} \leq P_{G_i} \leq P_{G_i}^{max}$ $B \cdot \theta = P_G - P_D$ $\frac{1}{x_{ij}}(\theta_i - \theta_j) \leq P_{i,j}^{max}$	Market data: - P_D: Power export/demand - c_i: Cost - $P_{G_i}^{max}, P_{G_i}^{min}$: Generator limits
	Nodal Data: - B: Susceptance/admittance matrix - x: Reactance - $P_{i,j}^{max}$: Line capacity - i: Bus/node
	Output: - θ: Voltage angle - P_G: Power production/generation

Table 4.1.: Overview of DC-OPF and its variables.

4.1.2. Adaption of Simplified Market Clearing Model

In this project, we derive a reduced version of the, we will call it, **simplified market clearing (SMC)** model proposed by [6], which implements an abstracted pricing model for a two-sided market as applied in the European electricity market. It represents an advanced version of the DC-OPF model. The SMC model regards different bids per seller and buyer, the unit commitment in a boolean variable, time relations as well as more detailed energy production costs. The following simplifications were applied, which brings the model closer to the usual DC-OPF formulation seen in the literature review:

- **No Inter-Time Relations:** We clear the market independently, not regarding inter-time relations.
- **One Bid:** We only consider one bid per generator and one bid per consumer.
- **One Type of Actor per Bus:** We work with exactly one consumer and maximum one generator per bus.
- **Inflexible Demand:** We have inflexible demand, meaning $P_D^{max} = P_D^{min} = P_D$.

The simplifications are done due to the structure of the found data as well as to create simple models for the MIP and NN optimizations. Not considering the inter-time relations means the minimum/maximum uptime constraints are not included and the actual start-up cost is distorted. We include the inter-start cost instead without relation to the previous time intervals as an approximation. When we only consider one bid per actor or one type of actor per bus, it is assumed that the bids can be aggregated. Lastly, *inflexible demand* is a condition often applied. The flexibility can easily be re-included with a more detailed data set. Ideas on future research without these simplifications are discussed in the final discussion in chapter 6.

The mathematical formulation was created having further implementation steps in mind. In order to make the input data fit into vectors and matrices with matching dimensions, the dimensions are set to the number of buses (*Nbus*). Empty values in case of no relations or no generators are set to zero. As a result, neighboring relations do not need to be specified, but the whole matrix can be traversed. This is the case for $P_{G_i}^{max}$ and $P_{G_i}^{min}$ of dimension $[Nbus \times 1]$ and $P_{i,j}^{max}$, x and B of dimensions $[Nbus \times Nbus]$. Notice that $\frac{1}{x}$ and B are the same value susceptance [52], which is replaced [6]. We use the common DC-OPF notation, to which two new variables are added. Inter-start cost is denoted as h_i and the boolean unit commitment variable as u_i. This results in the following **Gurobi DC-OPF**, which is given to the MIP solver (Gurobi) as the baseline for our optimization:

$$\begin{array}{cc}
min \sum_i^{Nbus} P_{Gi} \cdot c_i + h_i \cdot u_i & \\
s.t. & \\
P_{Gi} \geq P_{Gi}^{min} \cdot u_i & \forall i = 1, ...Nbus \\
P_{Gi} \leq P_{Gi}^{min} \cdot u_i & \forall i = 1, ...Nbus \\
P_{Gi} - P_{Di} = \sum_j^{Nbus} B_{ij} \cdot (\theta_i - \theta_j) & \forall i = 1, ...Nbus \\
B_{ij} \cdot (\theta_i - \theta_j) \leq P_{ij}^{max} & \forall i, j = 1, ...Nbus
\end{array}$$

4.2. Data Search

4.2.1. Sources for Energy Data Libraries

As part of this thesis, we searched through publicly available energy data sets and data libraries to find realistic and applicable data. As stated in the last model formulation, we are looking for market and nodal data and its variables. Each data set that provides a part of the needed data is discussed as combinations are possible. As a starting point, the data sets mentioned in the literature review and the online libraries in Table 4.2, which summarize energy system databases, were explored. Then with a free online search, some other databases were processed. We divide the energy data sources into three types: Energy operator statistics, energy data libraries and energy system generation tools. The sources are then grouped by area and data type.

4.2.2. Energy Operator Statistics

Databases directly from real-world operators are collected in **energy operator statistics**. ENTSO-E is the network of transmission system operators in Europe. It provides some public data on its Transparency Platform [95] and Power Statistics page [96]. It includes forecast

Name	Source
Open Energy System Models	[91]
Open Energy System Databases	[92]
Transmission Network Data Sets	[93]
ENERGYDATA	[94]

Table 4.2.: Sources that provide overview over energy data libraries.

and realization data on load, generation, prices and transfer capabilities per bidding zone in time intervals per hour or fifteen minutes. The data has many empty points as it is a purely optional collection, making the combination cumbersome. Some data points, such as cost and reactance/susceptance, remain missing. Additional data can be personally requested.

For the US market, the energy system operators PJM and MISO offer comprehensive public data collections. The PJM Data Miner [97] offers all needed market data. It includes different versions of hourly demand bids or load aggregated per different areas as well as hourly energy offers that measure the bid price curve and economic minimum and maximum. Some map visuals can be downloaded from the PJM online library. However, grid data is not provided and no connection can be made between the area codes and the generator identifications. The MISO market reports [98] include tables on cleared bids with the demand and historic offers with minimum, maximum and cost structure. Overall, MISO provides a data basis similar to PJM, lacking grid data. PJM and MISO data are used for time series/market data in different papers of the literature review, like [61][72]. An overview of the operator data is given in Table 4.3.

Area	Data Type	Name	Source	Format
Europe	Grid: Transfer Market: Load, generation	ENTSO-E Transparency Platform	[95]	xml, xlsx, csv
Europe	Market: Load, generation, gen. capacity	ENTSO-E Power Statistics	[96]	csv, xlsx
Part US	Market: Demand, generation max & min, cost	PJM Data Miner	[97]	csv
Part US	Market: Demand, generation max & min, cost	MISO Market Reports	[98]	csv

Table 4.3.: Overview of discussed statistics provided by power operators.

4.2.3. Energy Data Libraries

Next, there are **energy data libraries**, which collect and aggregate this data from operators. Enerdata [99] analyzes global energy trends. Its data includes energy consumption and production by country. Energy consumption data equals load data and could be used as energy demand. Also, from given production values minimum and maximum values could be abstracted. The data is not very detailed. The Open Power System database [100] is a free and public data platform build for power system modeling. It only offers a few data sets. Fitting ones are an overview of power plants in Europe with capacity and location as well as a table of hourly load and price, aggregated from ENTSO-E data. The Open Energy Platform [101] wants to make energy projects transparent and offers some open data tables. It comprises demand data per federal state of Germany and some supply data focusing on renewable resources. These data libraries give us some information on demand and a few points on generation, but not in the detailed format we need.

The IEEE PES Task Force on Benchmarks for Validation of Emerging Power System Algorithms developed a library of power grid benchmarks. It includes a benchmark library for OPF, which we will name IEEE PES PGLib OPF or shortened with "IEE-(number of buses)" as in the literature review. The project aimed to create benchmark data to make research results comparable. It is developed for the AC-OPF problem and can also be applied to DC-OPF. Some of the grids are derived from real-world systems. The files include the needed grid information and one distribution of market data. They are usually given as Matlab files, but some university libraries offer PowerWorld or other format versions [56][55]. This library finds use in many papers like [59] - [69].

The Grid Optimization (GO) competition [102] offers different input data with their challenges. One includes market data for an IEEE PES PGLib grid used in [60]. A summary of the data libraries is shown in Table 4.4.

Area	Data Type	Name	Source	Format
Global	Market: Load, generation	Enerdata Energy Statistics Yearbook	[99]	xlsx
Europe	Grid: Plants (also Germany) Market: Load, gen. capacity	Open Power System	[100]	csv, xlsx
Germany	Market: Load	Open Energy Platform	[101]	csv
Various	Grid: Line capacity, reactance	IEEE PES PGLib OPF benchmarks	[54]	m (pwb, pwd, epc, raw)
Various	Grid	GO Competition	[102]	raw, cop, inl, con

Table 4.4.: Overview of discussed energy data libraries.

4.2.4. Energy System Generation Tools

Lastly, we have **energy system generation tools** which often, over the course of a research project, produce some tool to extract data from real-world sources. They generate grids and other usable data structures from it that are of free use to further optimization projects. They might provide generated ready-to-use data sets. Two research papers focus on creating a full energy grid from open data. In [103], various network and geolocation data, such as light maps and roads, are analyzed to create a complete global energy transmission map. A version of the map, which contains many countries, is available as a geoprocessing file at [94]. Creating a model of the European interconnected system as a benchmark from publicly available data is the content of the paper of [104]. An official version of the data is available from the PowerWorld Corporation [105]. They create the pwd-file format, often employed for energy grid models. The data includes lines with reactance and capacity and is thereby a usable grid for power system optimization. It also contains one distribution of generator minimum, maximum and load per area, essentially the same format as the IEEE cases.

Then, there are projects from the state which share their code and found data sets. SciGRID [106] is a project by the German Federal Ministry of Education and Research to automatically generate models of existing energy grids for research. They create a grid for the European and German markets. The model includes transmission lines with voltage, reactance and maximum line current. For the vertices, only geographical information and voltage are given. ELMOD [107] is a project from the universities DIW Berlin and TU Berlin. The European public model is not very comprehensive. The German model is open source and its data includes a complete grid with line capacity, B and a set of line transmissions. Additionally, time series data of the demand is given as a sum over each time interval that can be split through static shares of each node. The generation capacity is given as a static value for each generator. This data could be used with some scaling steps along a sampling distribution, but it only offers a partial degree of detail of our optimization model.

Some general research projects focus on creating a tool for modeling and optimizing the energy system. Their code is usually downloadable from a public repository. However, the setup might require some reading into the project. The resulting data is usually only given on websites in an unofficial version (such as Zenodo, an open research platform financed by the European Commission). The GridKit project [108] is developed within SciGRID specifically for scientists. It is a data extraction tool using OpenStreetMap and the ENTSO-E map as a basis to build transmission lines and vertices. It therefore would make sense to combine it with the online available ENTSO-E market data. Files are downloadable on GitHub and Zenodo. GridKit is leveraged in [109] and is built to be incorporated into PyPSA. With PyPSA, they develop an energy model simulation and optimization tool, which includes some models. The project can be found on GitHub [110] as well as Zenodo. Their Europe data set includes a definition of the grid, hourly load and some cost information.

Next are two tools to generate synthetic energy grids. Some instance parameters, such as size and voltage levels, can be chosen with these generators. DINGO [111] creates synthetic power grids based on open data. MATPOWER [112] is a Matlab-based open-source tool for electric grid simulation and optimization. MATPOWER [113] used in some energy research papers of the literature review to self-generate grids [72]. An overview of the various energy grid generation tools can be found in Table 4.5.

4.2.5. Data Set Selection

With this analysis of the available data sources, we found no universal or ready-to-use solution. The ELMOD data for Germany might be the closest to it, but the data is not fully in the format we need for cost and demand. Many research papers simply pick some IEEE PES PGLib OPF grids and then sample the historical data. They take the given set of market data from the case and then scale the load with a random value drawn from a uniform or multivariate normal distribution within minimum and maximum. Dynamic cost and generation capacity is regarded as static or factored out [57][58][51][63] or calculated with a cost function from the given parameters [59]. This approach might not capture the underlying patterns of the

Area	Data Type	Name	Source	Format
Global	Grid	Global Transmission Map	[103][94]	gpk
Europe	Grid: Line capacity, x Market: Gen. capacity, load (1 distrib.)	European interconn. system model	[104][105]	pwb, xlsx
Europe & Germany	Grid: Line capacity, x	SciGRID	[106]	xlsx
Germany	Grid: Line capacity, B, line transmission Market: Demand, gen. capacity	ELMOD	[107]	GMS, xlsx
Various	Grid	GridKit	[108]	py (csv)
Various	Grid Market: Load	PyPSA	[110]	py (csv, nc, json)
Synthetic	Grid	DINGO	[111]	py
Synthetic	Grid	MATPOWER	[112]	(m)

Table 4.5.: Overview of discussed energy system generation tools.

data. In [58], a covariance matrix was created to mimic locational patterns within the sampler. Own grids can also be generated with GridKit or MATPOWER, as done in [72].

Market data can be extracted from the public data sources of ENTSO-E for Europe and MISO and PJM for parts of the US. They have many empty data points though, which need to be filled manually or with a script. To this market data, a grid needs to be added. SciGRID or the work of [104] might be an interesting option to be connected to the ENTSO-E data, but this would require manual labor and no research project leveraging this combination has been found. The IEEE PES PGLib data are the most commonly used grids in research and a high number of grid sizes are available, so the most fitting one can be chosen. This connection of data is applied in [61], combining PJM operation data and the IEEE-30 grid, probably resulting in data with the most significant real-world background used in the literature review papers. This data collection strategy has been chosen for our project.

4.3. Test Set Generation with IEEE PES PGLib and PJM

4.3.1. Data Collection

We divide the data set generation process into two parts. The data collection explains the selection process until the download. The data preparation describes which steps need to be applied to the data to receive data sets with feasible optimization solutions.

For the data collection, we inspected different data tables within the PJM data for generator and consumer time series data and different IEEE PES PGLib energy grids, as advised within section 4.2. To select specific data tables from the PJM data set, the variables, number of (grouped) actors and quality of the data points (e.g., few missing data points) were compared. The generator and consumer data were chosen to fit each other's time interval granularity and date. We selected the Hourly Load Metered table at [114] as consumer data and the Energy Market Generation Offer table at [115] as generator data. This consumer data separates the consumers into thirty areas. Therefore, the IEE PES PGLib 30-bus case was chosen. It includes thirty different consumers and six generators. The data selection of IEEE-30 and the consumers table matches [61] as far as explainable from their paper. They do not consider generator time series data but sampled it. The 30-bus case approximates an old version of the American electric grid, fitting the American PJM data.

For the IEEE-30 case, the version from [116] is used. It includes a file in the PowerWorld-format, so an academic test license of the PowerWorld Viewer/Simulator was downloaded to export the data into CSV. Two of the exported data tables are interesting to us. The Branches table includes all power lines connecting buses and the Generators-Grid table connects generators to buses. The PJM data can be exported directly as CSV from the online Data Miner for a chosen time interval. We selected the newest data that offers the needed amount of data points, which complies with June 2021 to July 2022. The Generators table must be downloaded in snippets per month. Six generators need to be chosen, which was done based on the highest number of data points given over all time intervals.

4.3.2. Data Preparation

The downloaded data needs to be adjusted to fit the optimization model and result in feasible solutions. The **data preparation** steps can be subdivided into data cleaning, data conversion and feasibility adjustments.

During **data cleaning**, the following steps were applied (especially to the Generators table):

- Adjust Formats
- Generate Missing Data
- *(Optional)* Adjust Variance
- Match Identifiers

First, data formats must be adjusted, especially dates and numbers. Next, missing values and their patterns need to be identified. The missing data points can be filled in with zero-bids, or be generated from historical data or similar entities. The choice depends on the patterns of the data holes and the similarities re-occurring within the data. If a more extended part was missing, it was filled with data from a same-sized part, so time-related patterns within the data would be kept. For example, suppose one month of data from one generator is missing and it acts similarly to another generator. In that case, the missing part can be filled with the others of that month (adjusted with a fixed value to adjust to its historic generation volume). One more optional step is to scale values of too similar data points with a uniformly sampled value and to reduce outliers to create a more homogenous variance. Lastly, with the year change, PJM creates a new unit (hash) code for their generators (the codes still seem to be alphabetically ordered). So fitting generators have to be identified and then their IDs in the new year need to be interchanged with their old IDs. Overall, this results in 52704 data points for six generators and 8784 time slots. Each step was accomplished using manual inspection in Excel and solved using a specifically created Python script. The Python scripts we wrote during the cleaning steps can be found in our repository [15].

With data inspection, we select the data points which translate to our model variables to create the **data conversion**. For this, we consult [117], which inspects the PJM data thoroughly and gives advice on interpretation. Some data points might require additional conversions, which are applied through Python scripts or manual fixes in Excel. The following conversion was chosen. The number and indices of buses can be taken from the Consumers table from PJM by factorizing the area code. For the data of bus consumption vector P_D, the demanded megawatt (MW) in the Consumers table is used.

The indices for modeling the bus connections are taken from 'From Number' and 'To Number' fields of the Branches table. B refers to susceptance in electrical engineering used to describe a cable's current flow quality. Susceptance is the reciprocal of reactance. [52][6] Points in the matrix where no bus connection is present are set to zero. Also, the flows are bidirectional, so a symmetric matrix is created. [59] P^{max} is created in a similar fashion with the column 'Lim MVA A' of the Branches table (other voltage types are mostly empty).

The indices of P_G, generator cost and capacity vectors, which should be filled with the generators values, are taken from the 'Bus Number' field of the Generators-Grid table from the IEEE-30 case. For the capacities, 'Max Ecomax' and 'Min Ecomin' are used, as advised in [117]. If there is no generator in the bus, these values are set to zero [59]. For the objective, the *incremental costs* are required to be multiplied by the generation volume. The incremental costs can be calculated from the energy generation offer curve, given in different data points of MW, referring to produced volume, and bid, the price at that volume, within the PJM Generators data. We adjust the cost formula from [117] to account for variable indices given (variable start index b^{min}) and choose to set the breaking point to calculate the generation cost as b^{max}, so that $y_{bmax} = 1$ and $z_{bmax} = 1$. This results in the following formula:

$$C_{pw}(x) = \begin{cases} 0 & (x = 0) \\ \sum_{i=b_{min}}^{min(x,b_{max})} bid_i \cdot (mw_i - mw_{i-1}) & x > 1, flag = FALSE \\ bid_{b_{min}} \cdot mw_{b_{min}} + \sum_{i=b_{min}+1}^{min(x,bmax)} bid_i \cdot (mw_i - mw_{i-1}) + \frac{(bid_i - bid_{i-1}) \cdot (mw_i - mw_{i-1})}{2} & x > 1, flag = TRUE \end{cases}$$

The bid flag refers to another column of the Generators table and decides whether the formula should be smoothed out. The formula then merges the data points to the sum of cost at x given the data points so far. We set the input variable x to b^{max}, which is the highest index of the given cost function data points, so every point is considered. We then divide $C_{pw}(b_{max})$ by the maximum load $mw_{b_{max}}$ to get the incremental value. The inter-start cost h per generator are taken from the respective column in the Generators table.

Since we have a feasible market as input and combining different sources might result in infeasibility, we must merge them with some **feasibility adjustments**. Our goal is to predict the solutions of a linear optimizer with ML by using the Gurobi input data and results as training data. Precisely, we do not aim for a NN which can certify infeasibility. Since Gurobi will throw an error or apply substantial simplifications that will obscure the data to overcome infeasibility, we can only include feasible solutions in the Gurobi and eventually the ML data.

The line limits P^{max} are pretty small, probably because the power system abstraction was created in the 1960s when less electricity was needed. So we increase them by one magnitude. Generator capacity limits P_G^{max} have to fit their outgoing branch limits. So we use scaling so each fits within their outgoing branches' limit sum. Lastly, the demand must be normalized to fit into the generator capacity. For normalization, the following formula was applied:

$$P_D^{norm} = P_D / (\sum_i^{N_{bus}} P_{D_i}) \cdot (\sum_i^{N_{bus}} P_{G_i}^{max}) \cdot (1 \pm gap)$$

We divide each demand by the demand sum and multiply by the generation maximum sum to adjust the magnitudes. Then we scale the result with a random gap, which we chose between 0 and 15% generated from a normal distribution, to not always fill the capacity. The variability and patterns of the demand bids remain thereby. An overview of data conversion and feasibility adjustment steps is given in table Table 4.6.

Variable	From	Conversion	Feasibility
i	Area code/index in Consumers table	Factorization	
P_D	MW in Consumers table		Normalize
Indices of B and P^{max}	From Number and To Number in Branches table		
B	Reactance X in Branches table	Reciprocal, Convert to Symmetric Matrix	
P^{max}	Lim MVA A in Buses table		Scale
Index of P_G (P_G^{min}, P_G^{max}, c)	Bus in Generators-Grid table		
P_G^{min}	Min Ecomin in Generators table		
P_G^{max}	Max Ecomax in Generators table		Scale
c	Bid_[i] and MW_[i] in Generators table	Apply cost function	
h	Inter-Start-Cost in Generators table		

Table 4.6.: Overview of translating data tables to variables.

Lastly, we format the existing data tables into two graph-like tables (Buses and Edges) for simple processing and to fit the mathematical vectors. The final Buses table summarizes the needed columns of the consumers, Generators and Generators-Grid tables. For its creation, Generators are (left-)joined with the 'Number of Bus' column from the Generators-Grid column, then the Consumers table is (left-)joined with the resulting tables on the bus ID. Each row is identified by the date and the bus ID. If no generator data is available, the cells are set to zero. The Edges table is created by adding the dates to the Branches table. This state requires a cross-join between all time intervals (unique timestamps from the Buses table) and all branches. The rows are then identifiable by date, source bus and target bus (only one row per connection since source and target are interchangeable). The abstracted algorithm to format the tables into graph-like data is given in algorithm 1. The algorithm creates buses and edges for all time intervals. The date in the Edges table will become important once we generate variable grid sizes in the next step. The Python scripts for data preparation and formatting can be taken from our repository [15].

```
input  : consumers, generators, branches and generatorsGrid tables
output : buses and edges
1 // Create Buses
2 generatorsGrid.genID ← generatorsGrid.index // Adjust other columns or IDs if
   neccessary ;
3 generators ← leftJoin(left: generators, right: generatorsGrid, on: 'genID', fillNA: 0);
4 buses ← leftJoin(left: buses, right: generators, on: ['datetime', 'busID'], fillNA: 0);
5 // Create Edges
6 timestamps ← unique(buses.datetime);
7 edges ← crossJoin(timestamps, branches);
```

Algorithm 1: Formatting data by merging tables to (buses, edges)-format.

4.3.3. Generating Dynamic Grid Data

This paper aims to test whether the model applies to practical use. One of the main hurdles poses the changing size of the grid. The electricity grid is growing continuously, resulting in buses being added or deleted over time. If the model could only process to the same grid, mostly considered in academic research as the standard case, it would eventually cease to be applicable. This situation constitutes the need for a model on dynamic grids. From the data set created so far, node splitting and node contraction, often discussed within graph theory [118], are applied to account for the addition and deletion of nodes. The theoretical formulation and a high-level algorithm for each method will be provided within this sub-chapter.

In **node splitting**, one node is chosen from which an edge to a newly created node is added. These two nodes then evenly distribute the old node's values between each other. This method ensures that the resulting graphs will still result in feasible models. For the splitting process, any node/bus i can be chosen and a new node with the now highest ID $j = Nbus + 1$ is created. The data from i is distributed between both nodes i & j. The generator limits and demand are finally divided into even parts. See formula for the new generation capacity minimum(P_G^{max} and P_D analogical):

$$P_{G_i}^{min,new} = P_{G_j}^{min} = \frac{P_{G_i}^{min,old}}{2}$$

At the same time, costs (incremental and inter-start) remain the same. See formula for c (h

analogical):

$$c_j^{new} = c_j^{old} = c_i$$

A connection between both nodes is added with the values $B_{i,j}$ and $P_{i,j}^{max}$ being sampled around the (non-zero) mean of the others. It can be calculated using a formula like the following. The gap should be a small random percentage that does not lead to a higher number than the total. $P_{i,j}^{max}$ is calculated analogical:

$$B_{i,j} = (\sum_{s,t}^{N_{bus}} B_{s,t}) \cdot (1 \pm gap) \leq 1$$

These adjustments must be made for all time intervals (in our case, hourly for one year), which makes the algorithm more complex. In order to keep order in the time intervals, the timestamps of data sets resulting from node splitting are increased by one year. The method was implemented in Python using *panda dataframes*. An abstracted pseudocode version is given in algorithm 2.

```
input  : buses of size $Nbus \geq 1$, edges, splitAt the chosen node
output : buses and edges including the new node
// Update Values
buses.where('busID' =).mwNorm ← buses.mwNorm/2;
buses.where('busID' =).minEcomin ← buses.minEcomin/2;
buses.where('busID' =).maxEcomax ← buses.maxEcomax/2;
// Create New Node
newBus ← buses.where('busID' = splitAt);
newBusID ← max(buses.busID) +1;
newBus.busID ← newBusID;
buses ← concat(buses, newBus);
buses ← sort(buses, by: ['datetime', 'busID']);
// Create Edge
x ← sampledAverage(edges.x);
capacity ← sampledAverage(edges.capacity);
newEdge ← {'from' : splitAt,'to' : newNodeID,'x' : x,'capacity' : capacity};
timestamps ← crossJoin(edges,datetime);
newEdgeTime ← unique(timestamps, newEdge);
edges ← concat(edges, newEdgeTime);
edges ← sort(edges, by: ['datetime', 'from', 'to']);
// Update Timestamp
buses.datetime ← buses.datetime - dateOffset(years: 1);
edges.datetime ← edges.datetime - dateOffset(years: 1);
```

Algorithm 2: Node splitting for timestamped data.

In **node concatenation**, one edge is chosen and both attached edges are unified. To keep the IDs of the nodes, an edge connected to the highest ID node i is chosen and the node values are aggregated into the other connected node j. For a high chance of preserving feasibility in the new model, the data is aggregated in the following way. The minimum of the lower capacity limits of both nodes is set as the new lower capacity limit:

$$P_{G_j}^{min,new} = min(P_{G_j}^{min,old}, P_{G_i}^{min})$$

For the upper capacity limit and demand, the sum is applied; see formula ($P_{G_j}^{max,new}$ analogical):

$$P_{D_j}^{new} = P_{D_j}^{old} + P_{D_i}$$

Lastly, a weighted average of the inter-start cost h and incremental cost c weighted by demand is created. See formula for h (c analogical):

$$h_j^{new} = h_j^{old} \cdot \frac{P_{D_j}^{old}}{P_{D_j}^{old} + P_{D_i}} + h_i \cdot \frac{P_{D_i}}{P_{D_j}^{old} + P_{D_i}}$$

The edge between i and j is deleted and the edge parameters are disregarded. Other connections from the deleted node i to another node k are redirected into j. If j already has a connection to k the edge values need to be aggregated. For x, we chose the mean and for the edge limit, the maximum. Again, this needs to be done for all time intervals. We propose an algorithm leveraging grouping in algorithm 3.

4.3.4. Optimization with Gurobi

After finishing the data preparation, the data sets must be optimized linearly with a MIP optimizer to create the target variables which should be predicted. Through the optimizer, it is solved for three variables of the dimensions $[NBus \times 1]$. The vector u includes the unit commitment of each bus and their associated generator, whether the generator will be turned on in this time interval. We also solve for P_G the generated amount of each generator in the interval. Both vectors will create values larger than zero only for buses with generators. The maximum generator capacity will be zero otherwise, thus limiting the generation through the generator operation limit constraint. Lastly, the θ per bus is found, representing the phase angles and accounting for the energy flow between the buses. The output variables are overviewed in Table 4.7.

For the optimization, we use Gurobi [119]. Gurobi is a state-of-the-art MIP solver. It is one of the strongest optimizers at the moment [26] and can be downloaded for free with an academic license. After registering and retrieving the academic license key, the Python Gurobi library *gurobipy* can be imported into the workspace for simple use. Additionally, the gurobipy library is available on Google Colab. A Python script needs to be implemented, which makes the solved-for variables and constraints known to Gurobi. Then Gurobi optimizes the model with the input parameters. The documentation of gurobipy can be found on [120], which we followed to implement the script.

```
input  : $buses$ of size $Nbus \geq 2$, $edges$, $concatTo$ which aggregated into
output : $buses$ and $edges$ without the deleted node
1  // Assign the same ID to concatenated nodes and aggregate in a group
2  delNodeID ← buses.max(buses.$busID$);
3  wAvg ← {x ← average($x$, $weights$=buses.$mwNorm$)};
4  aggregatFuncs ← $\{'mwNorm' : sum,' minEcomin' : min,' maxEcomax' :$
     $sum,' interStart' : wAvg,' incPrice' : wAvg\}$;
5  buses ← group(buses, $by$: $[datetime, busID]$);
6  buses ← aggregate(buses, $by$: aggregatFuncs);
7  // Delete connection, set same ids and aggregate for other edges
8  edges ← edges.where($!('from' = delNode\ \ 'to' = concatTo$ || $'from' = concatTo\ \ 'to' =$
     $delNode)$);
9  edges.where($'from'$= delNodeID).$from$ ← concatTo;
10 edges.where($'to'$= delNodeID).$to$ ← concatTo;
11 aggregatFuncs ← $\{'x' : mean,' capacity' : sum\}$;
12 edges ← group(edges, $by$: $[datetime, from, to]$);
13 edges ← aggregate(edges, $by$: aggregatFuncs);
14 // Update Timestamp
15 buses.$datetime$ ← buses.$datetime$ - dateOffset($years$: $1$);
16 edges.$datetime$ ← edges.$datetime$ - dateOffset($years$: $1$);
```

Algorithm 3: Node concatenation for timestamped data.

Variable	Description	Dimension
u	Unit commitment of generator	$Nbus \times 1$ (filled with 0)
P_G	Generated amount per generator	$Nbus \times 1$ (filled with 0)
θ	Phase angle at bus (for power flow)	$Nbus \times 1$

Table 4.7.: Overview of all output variables.

Since the Gurobi optimization and the later ML training require long processing times because more than nine-thousand time intervals and optimization problems are included per data set, we advise sampling from the data set. Applying a sample reduces computation time strongly. For accomplishing steps, the time intervals of the data set need to be identified, a selected amount of intervals (e.g., one-thousand) are randomly non-repetitively sampled from the available ones and then the associated buses and edges are chosen. The pseudocode is given in algorithm 4.

```
input  : buses, edges, Nsample sample size
output: busesSample and edgesSample
// Sample time intervals
timestamps ← unique(edges.datetime);
timeSample ← sample(timestamps, times=Nsample, random);
// Choose buses and edges
busesSample ← buses.where('datetime'.isIn(timeSample));
busesSample ← sort(busesSample, by: ['datetime', 'busID']);
edgesSample ← edges.where('datetime'.isIn(timeSample));
edgesSample ← sort(edgesSample, by: ['datetime', 'From', 'To']);
```

Algorithm 4: Sampling Buses and Edges tables for timestamped data.

From the Buses and Edges tables, matrices for each model need to be created to give to the Gurobi optimization script as input data. In the code, we optimize each time interval individually, thereby looping through the time intervals and cutting the associated buses and edges. Gurobi then includes the resulting market and nodal data into the constraints. Most needed data formatting was performed already. We still need to generate the matrices X and P^{max} of the form $[Nbus \times Nbus]$, as it does not fit the (buses, edges)- format. These formatting steps are implemented as part of the script. See algorithm 5 for the steps.

We implemented the simplified market clearing model, presented in chapter 4.1 before. An abstracted pseudocode of the optimization script is given in algorithm 6. The full version and three resulting data sets can be found on our repository [15]. Additionally, a function was written to write the resulting data into Excel tables again. We advise partially storing the data during the optimization, as the process requires time and might cut between the processing. With a CPU, the script with Gurobi optimization takes about one hour per one-thousand models. With the described feasibility adjustments, the optimization process resulted in all feasible solutions.

```
input  : $buses$ of size $Nbus \geq 1$, $edges$, $time$ the time interval of the model
output: $c$, $h$, $P_G^{min}$, $P_G^{max}$, $P_D$, $B$ and $P^{max}$ matrices for optimization
// Cut values for time interval
busesAct $\leftarrow$ buses.where('datetime' == time);
edgesAct $\leftarrow$ edges.where('datetime' == time);
c $\leftarrow$ busesAct.$incPrice$;
h $\leftarrow$ busesAct.$interStart$;
$P_G^{min}$ $\leftarrow$ busesAct.$minEcomin$;
$P_G^{max}$ $\leftarrow$ busesAct.$maxEcomax$;
$P_D$ $\leftarrow$ busesAct.$mwNorm$;
// Create Nbus x Nbus matrices
B $\leftarrow$ zeros(size: [Nbus, Nbus ]);
$P^{max}$ $\leftarrow$ zeros(size: [Nbus, Nbus ]);
for $i \leftarrow 1$ to Nbus do
    fromEdge $\leftarrow$ edgesAct.at($i$).$from$;
    toEdge $\leftarrow$ edgesAct.at($i$).$to$;
    B.at(fromEdge, toEdge) $\leftarrow$ 1/edgesAct.at($i$).$x$;
    B.at(toEdge, fromEdge) $\leftarrow$ 1/edgesAct.at($i$).$x$;
    $P^{max}$.at(fromEdge, toEdge) $\leftarrow$ edgesAct.at($i$).$capacity$;
    $P^{max}$.at(toEdge, fromEdge) $\leftarrow$ edgesAct.at($i$).$capacity$;
end
```

Algorithm 5: Generation of Gurobi input data for the simplified market clearing model.

```
input  : $c$, $h$, $P_G^{min}$, $P_G^{max}$, $P_D$, $B$ and $P^{max}$ input matrices
output: $solVars$ all output variables
// Create model
model ← GurobiModel();
// Add variables
$P_G$ ← model.addMatrixVar(shape=NBus, type=continuous);
u ← model.addMatrixVar(shape=NBus, type=binary);
$\theta$ ← model.addMatrixVar(shape=NBus, type=continuous);
// Add objective
model.setObjective(sum(c [i] * $P_G[i] + h[i] * u[i]$, for i ← 1 to NBus));
// Add constraints
model.addConstraints($P_G^{min}[i] * u[i] \leq P_G^{max}[i]$, for i ← 1 to NBus);
model.addConstraints($P_G^{max}[i] * u[i] \geq P_G^{min}[i]$, for i ← 1 to NBus);
model.addConstraints($P_G[i]$ - $P_D[i]$ = sum(B [i,j] * ($\theta[i]$ - $\theta[j]$)) , for[i, j] ← 1 to
   NBus);
model.addConstraints(B [i,j] * ($\theta[i]$- $\theta[j]$) $\leq P^{max}[i, j]$, for [i,j] ← 1 to NBus);
model.optimize();
solVars ← model.getSolVariables();

```

Algorithm 6: Optimization with Gurobi for the simplified market clearing model.

5. Network

5.1. Network Design Research

5.1.1. Data Structure within NN

The data structure implied by the problem model and generated within section 4 was inspected in the context of NN research to find fitting solution approaches with ML. The main characteristics of the data structure were extracted, each discussed in papers on this general area of research. From the NN perspective, the **characteristics of the input data** are the following:

- Multivariate
- (Discrete) Time Series
- Spatial Relations
- Dynamic Graph

Multivariate data refers to the fact that we have different input variables given. Each input variable refers to one data value, e.g., the cost of a generator. We more specifically have *multivariate time series (MVTS)* data. Through the historical data of previous optimization intervals, a time perspective is added. For each time snapshot, values of all the variables are given. [121] With the so-called *snapshot* character, the data points are given in *discrete time* steps (precisely every hour in this case). In contrast to continuous time, the data can be updated for each variable (or node) at a different, somewhat random point in time. [12] The data up to these two characteristics could be learned with a normal NN (each snapshot as an independent data point) or a recurrent NN to account for the time relation.

The variables in our data retain not only a temporal relation but also a spatial relation through the graph structure of each input data set. This combination is often referred to as *spatio-temporal* data. It is essentially MVTS data with graph-relationships among one time step. [121] Both spatial and temporal relations cause the data to deviate further from being iid, which is assumed in normal NNs. [10] The spatio-temporal data requires structures that handle temporal graphs. Each time instance of the graph produces its own version of a graph. The mathematical formulation for the graph becomes essentially $G^t = (V, E, X_v^t, X_e^t)$, where t refers to the time stamp, X_v to the node features and X_e to the edge features [122]. The graph thereby changes the feature values over time for each snapshot. Standard graph neural networks can handle spatial relations but are built to make predictions within one specific graph. In the process, it compares nodes within the graph to each other. So multiple different graphs are not considered in the usual case. *Multi-graph learning*, on the other hand, regards different graphs but is only used to create a prediction on the graph level (not at the node level as needed). Multi-graph learning aggregates all hidden presentations within one graph instance into one state. From these graph-level states, the cost function is generated. [86] In more recent research, *recurrent graph neural network (RGNN)* models are discussed, such as those mentioned in [14] and [121]. As a combination of RNN and GNN, they can account for temporal and spatial relations in the data.

At this stage, if the iid assumption is neglected, the temporal graph snapshots could be used as data points for traditional NN. Such an NN takes a one-dimensional input vector with one value as one input node. These values are combined through a weighted sum into another value at the following hidden layer. From intuition, the values of the same nodes would be at the same input point for each data point. The NN should learn to weigh connections higher that influence each other, thereby learning the spatial relations. As a downside, this would require much data and long training. Alternatively, the layers would not need to be fully connected. So efforts could be applied to only create connections concerning spatial relations. However, one relevant aspect of realistic data cannot be regarded with this method. The graph is *dynamic*. Dynamic means that nodes and their connections can be added and deleted over time. All parts of the graph become adjustable $G^t = (V^t, E^t, X_v^t, X_e^t)$. It requires the model to be able to deal with varying input sizes. In the following subsection 5.1.2, the challenges of varying input sizes in traditional NNs are analyzed.

Since in GNNs the neighbors of a node are regarded rather than all input variables, the changing input size of a graph does not matter. Previous states of a specific node would only be missed when being included in its own hidden representation at a new time step, as done in RGNNs. In the implementation of these models, the historic state of a new node would be set to zero. So there is simply no influence of historic states on a node created at that time step. When nodes are deleted, they are not regarded in the calculation anymore. These insights are taken from the PyTorch Geometric Temporal code [13] and the mathematical formulations behind it, e.g., [14]. We will discuss the network design in detail in subsection 5.2.2.

The specific **output data characteristics** include the following points:

- Node-based Prediction
- Multiple Attributes

The generated information is predicted on the node level within GNN, compared to edge or graph-level predictions. Node-level is difficult for some models, like RNN and Multi-GNN, which would predict on the entire graph if a graph is given as input. We also predict multiple node-level values. Thereby phase angle and power generation are linear variables, while unit commitment would be logistic regression. Multiple outputs can be covered with multiple output nodes of the output layer. Depending on the implementations, they might require adjustments if the output functions are of a different kind, as different output functions or output layers need to be applied. In practice, the unit commitment is contained within the generation volume, so the complete insight is created by only predicting the two linear outcomes; the importance of unit commitment rather lies within the constraints.

In Table 5.1, the discussed NN models are compared by whether they can fulfill each data characteristic requirement. The multivariate and multiple output characteristics are cut as they are 'ok' for all models.

Model / Characteristic	Temporal	Spatial	Dynamic	Node-based prediction
NN	ok, if assumed independent	no (learnable)	problematic	ok
RNN	ok (one graph as x)	no	no	no
GNN	no	ok	ok	ok
Multi-GNN	ok, if assumed independent	ok	ok	no
RGNN	ok	ok	ok (in code)	ok, but prediction-focus

Table 5.1.: Comparison of NN models over fit to data structure characteristics. The characteristics multivariate and multiple outputs are 'ok' for all models, consequently cut.

From table Table 5.1, we can see RGNN would cover all characteristics. It is interesting to mention that the temporal data is not needed to generate the output and all the needed information is included in one graph when using linear programming-based optimization. However, the historical data gives us information on previously solved instances. More historical data points can be helpful to improve the accuracy of the model, like in traditional NN-based solutions. These insights, at least in theory, prioritize the data for the prediction in the following order: The nodes' own values, spatial relations, and lastly historical relations.

Many RGNNs are developed in the context of traffic prediction and therefore strongly focus on the time aspect [14][121]. In our problem, prediction is not relevant. The problem does not require the model to predict but how to best integrate all information into its state while learning historically. This divergence in goals does not need to compromise accuracy but raises the question of the applicability of more straightforward solutions. GNNs have reached a high interest in research since 2020. An even more reduced method that was concentrated on before could be traditional NNs, which are applied in many rather successful studies of OPF and ML. [10] As we can see from the table, NN would be a more simple solution if the data size would not be dynamic and the assumption of independence between snapshots could be applied. Efforts to overcome varying input sizes in NNs are discussed in the following.

5.1.2. Varying Input Sizes in Traditional NN

Dealing with varying input sizes in NN specifically is relevant, as otherwise, the data of new variables cannot be included. Varying input size of this kind is usually not considered in traditional NN applications. Next, the **restrictions of the input dimensions** will be addressed, due to the input of a NN being a one-dimensional vector. In the NN architecture, one node in the layer represents one value $a_i^{(l)}$, starting with the input values sent to the nodes in the next layer or used as a network output value. One layer (input, hidden or output) is therefore represented in a one-dimensional vector $a^{(l)}$. Weights $w_{i,j}^{(l)}$ for each node input value (from input node i into present node j in layer l) are learned and included into the calculation of the output value at every node. This is done by creating a weighted sum with bias ($z_i^{(l)}$) and then applying the activation function for non-linear relations. See subsection 3.2.6 for reference of the mathematical formulations.

So for new variables, the weight would be undefined (at that index) and the model would need to be adapted to integrate the new values into the calculation. The initial weights of new node values could be set to random or a similar weight, but for accuracy, they would need to be learned during test phases. In order to integrate the values into the network and its functions, the calculation of the aggregated inputs at each NN layer node ($z_j^{(l)}$) would have to be expanded. A completely new node could be added and integrated into the model. However, the model would need to be re-trained and adapted every time the network changes. For deleted nodes, the old indices could be set to zero, which would not be clean but also not influence the hidden states in the network.

We collected some **methods to include varying input sizes into NNs**, their usage and problems in our case. Methods were taken from standard literature on NNs [8] and online blog posts on that topic. An overview of the methods, its workings and usual applications is summarized in Table 5.2. We will discuss the methods, possible solution approaches and problems to make them fit into our data structure.

Method	Combat Varying size	Usual Application
Zero Padding	Upper bound in input size and pad 0	Empty values in data point
Special embeddings	Transform input data into fixed-length sequence	Words into embedding in NLP
Convolution	Layer to adjust input size by aggregating input data with learned parameters	Image Recognition (extracting underlying features)
Pooling	Layer to downsize feature map through summarizing function	Image Recognition (connecting underlying features)
Sequential data / RNN	Process variable length sequences	NLP, e.g., Translation

Table 5.2.: Overview of methods to combat varying input size methods in NN.

For NNs, the input data would essentially have to be linearized into a one-dimensional vector. Using *zero padding*, some spaces could be filled with zero within this vector to account for empty data points. This way, the input size could change to an extent without impacting the outcome. In order to account for increasing input data size with new nodes, the vector length could be increased and all the not (yet) needed points set to zero. This way, the size could be expanded to a certain threshold and decreased by filling these spaces with zero. This approach requires a fixed maximum length that can only be increased later by creating a new model. Also, the approach could be more space-efficient as it enlarges the data unnecessarily.

Again affecting the input data representations, *special fixed-length embeddings* could be created from the input data. A popular example are word embeddings in NLP that create an identifiable fixed-length vector for each word. For this, a method needs to be developed that creates the same vector size for a different number of nodes. Since we want to generate the output in relation to the node number, it seems illogical to lose the node's identity in the process. The addition of a new node adds its node attributes, as well as new connections to the adjacency matrix and edge feature values. This information would need to be merged with other nodes to create a fixed length, e.g., combining the same variables of the nodes. The only logical way would be to create embeddings per node, like in a GNN.

GCNs create an embedding for a node in the calculation process, that includes its neighbor's information and the connecting edges. A GCN trains within one graph, while the prediction would need to happen for each embedding individually. The challenge lies in the fact that in order to achieve optimal results for computing embeddings, the GCN must be trained to generate the embeddings, while a separate model is required for making predictions. Having two separate models is not part of the typical training process. So it would be necessary to create a respective training pipeline. Within the paper [11], an application of this method is given, which uses GCN embeddings and reinforcement learning predictions. It will be further discussed in the next subsection 5.1.3.

Convolutions within NNs, presented in 3.2.7, are used to adjust the input size by aggregating input data with learned parameters. They are mostly employed within image recognition or image-related tasks. Their aggregation of a set of input values can extract underlying features. Through the dynamic size of the aggregation window, the input size can be variable and influence the output size. However, they are limited to two-dimensional input matrices. Multiple matrices can be regarded through input channels, but their dimensions need to be equal, at least at some point, to be aggregated. Additionally, the input shapes in all channels are assumed to be equal in most implementations and even square in some. It is questionable whether a meaningful mathematical relation of the input dimensions with our data can be found through the frame of convolutions (and pooling). The issue arises from the different shapes of the input. Consider N for the number of nodes, A for the number of node attributes, E for the number of edges and F for the number of edge features.

We provide a mathematical example of the possible input and output dimensions. Intuitively, our data would fit into the following three two-dimensional matrices: Node-attribute matrix ($[N \times A]$), adjacency matrix (different formats possible: $[N \times N]$ or $[E \times 2]$) and edge-feature matrix $[E \times F]$. Input channels being numbered with prefix "c" and kernels with "k" for all three input channels, calculating the height of the output of the node channel $c1$ and edge channel $c3$ would be: [83]

$$h_{c1} = (N - h_{k1} + 2p_{c1})/s + 1 \neq (E - h_{k3} + 2p_{c3})/s + 1 = h_{c3}$$

The sizes of the matrices are not the same. The use of padding seems illogical, as node values would be aggregated with values not connected to them. Since the different dimensions are in no mathematical relation to each other and their size can only be influenced by the convolution parameters (kernel dimensions, padding and stride), they cannot become equal. The parameters would need to be dynamically adjusted to the dimensions, but they need to stay the same for every model use.

If the number of features of the edge is fixed and rather low, which is the case in our tests, the matrices can also be built like this: Node-attribute matrix $[N \times A]$, adjacency matrix with first feature value $[N \times N]$ and adjacency matrix with second feature value $[N \times N]$. N defines the height of all input channels, which is the same.

$$h_{c1} = (N - h_{k1} + 2p_{c1})/s + 1 = h_{c2} = h_{c3}$$

However, A and N are in no relation but determine the width of the output channels.

$$w_{c1} = (A - h_{k1} + 2p_{c1})/s + 1 \neq (N - h_{k1} + 2p_{c1})/s + 1 = h_{c2} = h_{c3}$$

We could create different one-dimensional $[N \times 1]$ matrices for each node attribute, then the height and width of the output channels would finally fit. In that case, the related information is entirely cut apart. Some further inspection is necessary to find logical combinations and configurations of convolution and pooling layers. The application can be successful if these combinations result in meaningful underlying structures. Similar to GCN, all attribute vectors could be aggregated and multiplied to the adjacency vectors. However, the weights of this impact should be trained. There is no proper operation for this within convolutional neural networks, resulting in part of the model being just matrix multiplication. Even with a representation of fitting dimensions, it is questionable whether the kernel would produce good underlying features within this order of values.

Pooling is part of a convolutional NN. It aggregates information with a function. Pooling does not include learnable parameters but uses a summarizing metric like average. The features could be summarized this way, but no logical summary can be found that is independent of the node number, similar to special embeddings.

The last option to include varying input size is with *RNNs*. RNNs take sequential data of any length and create single or multiple predictions. Each point in the sequential data has to be the same length. So we cannot give a graph, but we could set each node as one data point within the sequential data. However, this model could not encompass the complex temporal or sequential relations between given nodes. Each snapshot's nodes would have to be given within a sequence without a clear indicator of the changed time frame or node identities. Temporal and spatial identification variables could be added to the embedding. However, a NN does not handle non-linear variables and their influence is unclear in combination with the actual linear input values. Finally, incorporating edge features would pose a challenge. Including them in the current node embedding, the spatial relations between the nodes and edge features would be lost. Alternatively, if the edge features were added as individual data points, the vector sizes would need to be identical. The problems of varying input size methods with our data structure can be found in Table 5.3.

5.1.3. Review Varying Grid Sizes in OPF

Throughout the research on OPF and ML, the problem of varying grid sizes is not discussed in the papers of the literature review. Taken from their formulations, the proposed NN models work with fixed-size one-dimensional inputs. Only data pre-processing steps are mentioned

Method	Problem
Zero Padding	Requires maximum size, enlarges input data
Special embeddings	No linear embeddings independent of node number, training of embedding not included in model
Convolution	Limited to multiple 2D input matrices fitting in size, combination of input matrices difficult
Pooling	No learnable parameters, no logical summaries independent of node number
Sequential data / RNN	Spatial and temporal relation between nodes unidentified, difficult including edge features

Table 5.3.: Overview of problems regarding our data structure and methods in NN to work with varying input sizes.

before these fixed embeddings as well as a few simplifications to the model not to include all data points. In the research of **ML and general MIPs**, three papers use approaches that could include non-iid data with varying input sizes. Sequential data is processed in a LSTM by [49], to find the next cut and remember the choices before. Graph convolutional networks are used in two papers. A GCN is learned for encoding branching policies, utilizing a bipartite representation from MIP models essentially as feature engineering, in [39]. Imitation learning is then applied to predict an expert rule (strong branching). MIP instances are transformed into a tripartite graph processed by a GCN in [48]. The transformation process is given in Figure 5.1. The graphic is taken directly from the paper. All three papers' targets differ strongly from our target.

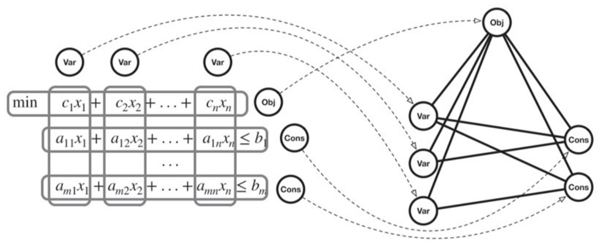
Var
Var
Var
min
c_1x_1 + c_2x_2 + ... + c_nx_n
Obj
$a_{11}x_1$ + $a_{12}x_2$ + ... + $a_{1n}x_n \leq b_1$
Cons
...
$a_{m1}x_1$ + $a_{m2}x_2$ + ... + $a_{mn}x_n \leq b_n$
Cons
Obj
Var
Var
Var
Cons
Cons

Figure 5.1.: Embedding weights, constants and objective value of the MIP instances into a tripartite graph for a GNN, taken from [48].

Liao [10] discussed the applications of **GNNs in general power systems**. The existing methods assume the network topology to be fixed [10]. They mention two papers applying GNN to the optimal power flow problem. In their own research, they predict the status codes for electrical equipment and based on that predict the reactive power load. This is achieved within the electric grid (encoded in an adjacency matrix) using a GCN. Their architecture includes two convolutional layers, ReLU activation functions, L2 regularization and the IEEE-30 data set. [123] The paper is not publicly available but was provided by the authors and translated using DeepL.

The paper closest to our approach is by [11]. Like our approach, they set the buses as nodes and edges as power lines with edge weights. They deal with the AC-OPF model. For the node vector, they initially set the embedding with voltage magnitude and angle for each active and reactive power, resulting in a matrix of size $[N \times 4]$. Through a selection matrix $[M \times N]$, they pick the generator nodes from the buses, for which the generating active power is encoded in another matrix. For the GNN, they input node features $[N \times F]$ and the adjacency matrix $[N \times N]$. They apply a few steps of a GCN, thereby encoding each node's neighbor's feature vector to its own (X) through the adjacency matrix (A) and learnable weights (W).

$$X_i^{(k+1)} = \sum_{k=0}^{K-1} \tilde{A}^{(k)} X_i^{(k)} W^k$$

The formulation is adapted from the paper to the notation used in this paper and translates to a normal GCN without normalization described in subsection 3.2.8. The model is trained through imitation learning, similarly to [39] for predicting branching policies. The GCN is therefore used to create a scalable embedding that can be fed to an imitation learning model. Even though not explicitly mentioned in the paper, this model could therefore deal with varying grid sizes. The problem of GCN having a different target is circumvented using an imitation learning model for prediction. The imitation learning model and the combination of learning both models is not further discussed in the paper. Papers on other areas can be found that apply a similar strategy, e.g., in traffic operation [124] or for synchronizing decentralized controllers [125].

In the model, only the load for active and reactive power is included, edge weights are ignored, and only the (active and reactive) generated power at each node is predicted. The IEEE-30 and IEEE-118 data sets are utilized to create sample data. The loads are sampled and the generated power is solved for via linear optimization of AC-OPF. To evaluate, they use the measure root mean squared error (RMSE) on the generation power prediction and only include the generator nodes in the calculation. Their presented value is very good, with a discrepancy between 0.14 and 0.06. Our approach's differences are that they test with both grid sizes separately and therefore do not acknowledge varying grid sizes. Additionally, we use only one complete temporal GNN model, use different variables and can compute phase angles that are needed for the MIP computations.

5.1.4. Network Selection

Through the research on possibilities to include varying grid sizes in our data structure in neural networks and machine learning optimization of OPF, we identified four NN-based solution approaches with the mathematical capacity for this task. It is possible to divide the data into fitting matrices, e.g., $A \times [N \times 1]$ node attribute matrices and $F \times [N \times N]$ edge feature matrices as shown before, and give these to a convolutional neural network. A fitting combination of convolution and pooling layers as well as input data representation and combination needs to be found, while the potential for meaningful extractions remains unclear.

One big GCN can be utilized when providing it with one comprehensive graph composing all training data point graphs. The temporal relations could be included as specific edges. An approach similar to this was also discussed in [121], where they learn spatial and temporal relations and then feed the graph to a GCN. This approach refines the weights to include neighboring information over many training graphs. It could cover temporal relations when the respective edge weights are adjusted or trained correctly. However, this approach may not have enough complexity in its trainable weights. The to-be-tested graph would also need to be included, so the graph would have to be re-learned, not resulting in time savings compared to MIP optimization. A temporal part is needed to supply new graphs in the test phase.

A more promising solution would be adding another prediction model on top of the GCN, as presented in [11] and previously discussed within special embeddings. Using a GCN to create fixed-size embeddings of each node in a variable-size grid is an approach already applied in some research papers. These embeddings could be fed to another model, such as an imitation learning approach or a simple NN. The problem is that two models must be implemented, connected and trained together. For their simplified approach, the work in [11] provides proof that this method can be very successful.

A model that should be fully able to integrate all data and training on its own is a (dynamic) RGNN or *dynamic temporal graph neural network*. These networks calculate a GCN layer embedding for each node and add a recursive layer on top, including the same node's historic states into its new hidden state. Using GCN, the grid size does not affect the embedding, as only the next k neighbors are included. [12][14] Implementation-wise, if the node is new and there is no historical state, the state will be set to zero and therefore has no influence on the hidden state. The prediction occurs per node, so deleted nodes do not affect the calculation. [13]

This method was primarily used and developed for traffic prediction [13][14]. We do not have a prediction case at hand and instead want to merge information of one time step. However, the historical predictions are similar at each node and might give an idea of the usual magnitude at each node. This approach could provide a complete and ready-to-use pipeline that is openly available. The literature review paper on GNNs in power systems [10] proposes using dynamic GNNs to solve the problem of making the models adaptable to grid changes. The research in this area is cutting-edge and in high development, so we will be the first to test if it can be applied to OPF prediction under more realistic conditions.

5.2. Test with Dynamic Temporal GNN

5.2.1. Data Structure in PyTorch Geometric Temporal

In the following, we describe all the necessary steps to our model tests. The full Python scripts, data sets and some test logs can be taken from our repository [15]. The first open-source library for temporal graph learning was published in 2021, called PyTorch Geometric Temporal [13]. It implements different data structures and deep learning layers to process spatio-temporal data structures. The library implements discrete time *signals*, snapshots of graphs over a constant time difference. Thus this kind of model is sometimes referred to as *graph signal processing*. Discrete constant time differences are given with our data structure, precisely one per hour. The library can cover static and dynamic graphs and is built on PyTorch Geometric [126].[13]

PyTorch Geometric Temporal implements different data structures built on PyTorch Geometric, offering GNN implementation pipelines. The *DynamicGraphTemporalSignal* data structure needs to be used to account for temporal signals with dynamic graphs. A temporal data snapshot is a PyTorch Geometric Data object. It encodes a graph. The attributes include *edge_index* of format $[T \times 2 \times E]$, which encodes the admittance matrix or graph connectivity in *coordinate (COO) format*. The COO format contains two lists of all source and all target node indices, while both vectors are mirrored to account for bidirectional edges. *edge_attr* $[T \times E]$ captures an iterator of all edge weights at that timestamp in the order of the index. Then x of format $[T \times F]$ includes all node features and y of $[T \times O]$ the targets.

Two things are mentioned wrong in the documentation but are corrected in the GitHub code or issue reports. All attributes are encoded as *numpy* arrays, otherwise, the DynamicGraphTemporalSignal class will run an error. Additionally, it must be noted that for this data object, only one edge feature can be included since multiple is not supported in the implementation of the layers. This limitation means that we must reduce our two edge weights from the problem model. We have tried different approaches here, more on it in subsection 5.2.4. We provide the shorter pseudocode with helper functions in algorithm 7 and data set generation in algorithm 8.

```
input  : $buses$ of size $Nbus \geq 1$, $edges$ and the timestamp $time$
output: $features$, for a DynamicGraphTemporalSignal instance
// Help functions to create attributes
Function edgeIndices(time):
    source ← edges.where('datetime' == time).$from$;
    target ← edges.where('datetime' == time).$to$;
    return ([concat(source, target), concat(target, source)])
End Function;
Function nodeFeatures(time):
    features ← buses.where('datetime' == time).select
      ([$mwNorm, incPrice, interStart, minEcomin, maxEcomax$]);
    return features
End Function;
Function edgeFeatures(time):
    // Different options possible here, one value between x and
       capacity
    features ← edges.where('datetime' == time).select ([$capacity$];
    return (concat([features, features], $axis = 0$)
End Function;
Function targetFeatures(time):
    features ← buses.where('datetime' == time).select ([$\theta$,$P_G$]);
    return features;
    End Function;

```

Algorithm 7: Helper functions to create the DynamicGraphTemporalSignal data set.

```
input  : buses of size $Nbus \geq 1$ and edges
output: dataset, a DynamicGraphTemporalSignal instance
1  timeStamps ← buses.unique('datetime');
2  for time ← 1 to timeStamps do
3      edgeIndex ← numpyArray(edgeIndices(time), type = float32);
4      x ← numpyArray(nodeFeatures(time), type = float32);
5      edgeWeight ← numpyArray(edgeFeatures(time), type = float32);
6      y ← numpyArray(targetFeatures()(time), type = float32);
7      edgeIndices.append(edgeIndex);
8      edgeWeights.append(edgeWeight);
9      features.append(x);
10     targets.append(y);
11 end
12 dataset ← DynamicGraphTemporalSignal (edgeIndices, edgeWeights, features, targets);
13
```

Algorithm 8: Creation of the DynamicGraphTemporalSignal dataset.

A created graph snapshot can be visualized like in Figure 5.2. It was created with the libraries networkx and matplotlib based on a PyTorchGeometric Data object, instantiated in a similar fashion.

For temporal data, it is important that the buses and edges are given in historical and node-specific order. This is the case with the buses and edges data sets acquired through section 4.3. For our tests, we worked with different (buses, edges)-formatted dynamic data sets to test the applicability of variable grid sizes. Since the ratio of different set types has a strong influence, we concatenated the sets while taking various amounts of each, e.g., through applying the $tail(numNodes \cdot instances)$ function for earlier sets and $head(numNodes \cdot instances)$ function for later sets or by using the provided sampling method in the chapter 4.3.4.

From this data, the training, testing and validation data sets can be generated through the PyTorch Geometric included temporal split. It takes the earlier part for the training and the latter for the test data set by the given ratio. We choose 80/20 for training/test and 60/20/20 for training/validation/test data sets. This results in the function calls in algorithm 9.

```
input  : buses of size $Nbus \geq 1$ and edges
output: trainData, validationData and testData, DynamicGraphTemporalSignal
        instances
1  trainData, testData ← temporalSignalSplit (dataset, trainRatio=0.8);
2  // Function Calls with validation set
3  trainData, testData ← temporalSignalSplit (dataset, trainRatio=0.6);
4  validationData, testData ← temporalSignalSplit (dataset, trainRatio=0.5);
```

Algorithm 9: Data set split into training, validation and test data.

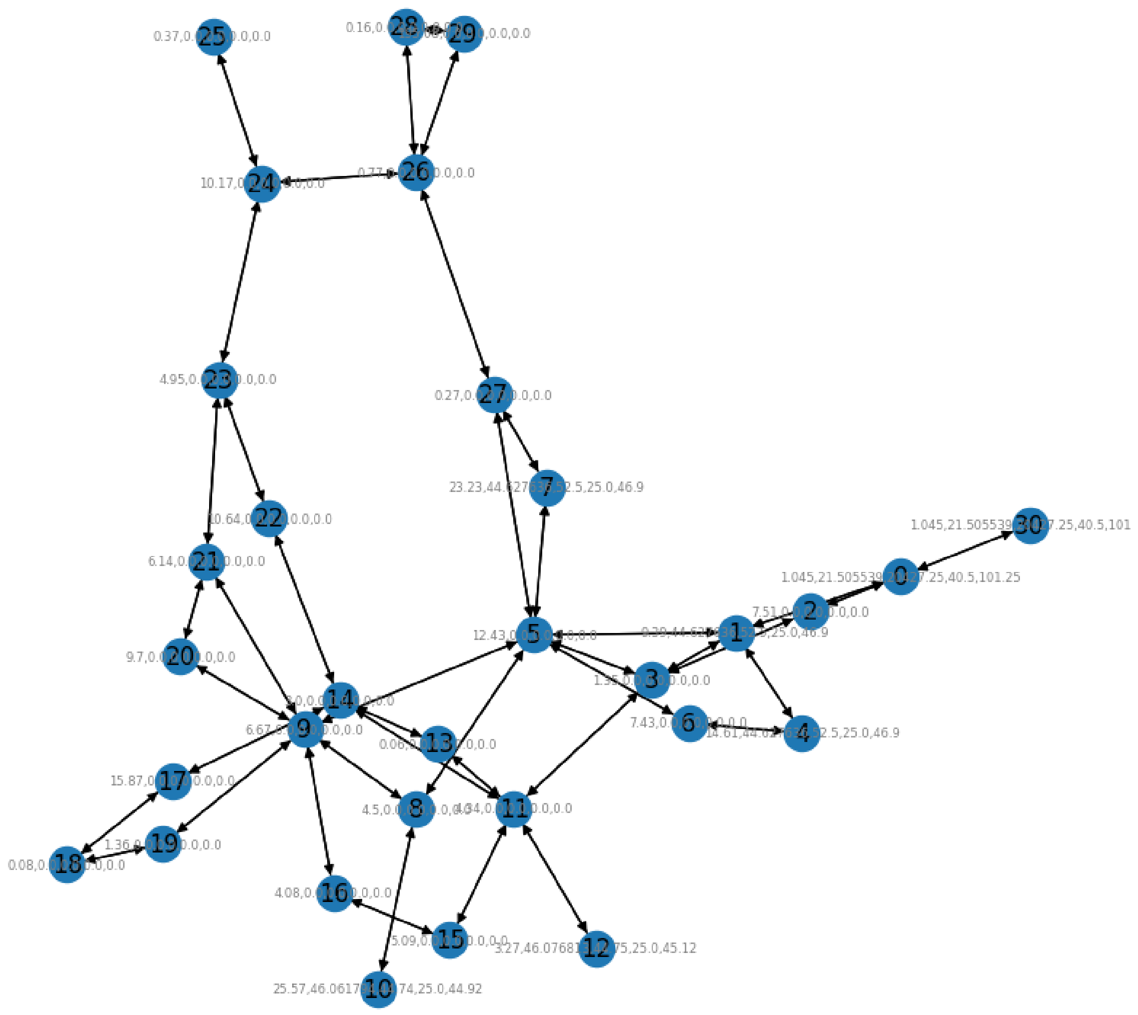

Figure 5.2.: Visualization of one PyTorch Geometric graph instance with the Python networkx package, including edges, nodes and their feature values.

5.2.2. Network Architecture in PyTorch Geometric Temporal

The PyTorch Geometric Temporal library is built on the PyTorch Geometric functionality and implements additional temporal layers from recent research papers. The training pipeline is implemented. To set up the network, one must choose the layers, set parameter sizes and hyperparameters in the constructor and forward pass functions . For our system to work with our data, we chose a recurrent graph convolutional layer. Because of the convolution aspect, these can work with dynamic data and add a recurrent layer on top for the time perspective between node instances. We specifically chose the *diffusion convolution recurrent neural networks (DCRNN)* layer from [14], as it is proposed in many examples, is more intuitive to understand and did best in a small sample test of other common layers. None of the possible layers implemented support matrix multiplications with multiple edge features.

The DCRNN layer is adapted from the paper [14], which initially developed a network to predict traffic forecasts. The learned parameters W refer to the *diffusion*. Diffusion relates to the probability distribution of how much traffic from one node is transferred to another, which refers to the amount of electricity transferred between nodes in our example. The diffusion is applied in steps, where with each step k, the diffusion reaches a node further away. This process translates to a standard graph convolutional layer. The convolution calculation is applied without normalization and divided into two parts, one for each flow direction. Their mathematical model weights the adjusted adjacency matrix $\tilde{A}$ with the edge weights X_E. The calculations for the *diffusion convolution* proposed by the paper, translated to our notation, look as follows: [14]

$$X_{V_{i,f}}^{(k+1)} = \sum_{k=1}^{K} (W_1^k(\tilde{A}^{-1}V_E)^k + W_2^k(\tilde{A}^{-1}V_E^T)^k) X_{V_{i,f}}^k \qquad \forall i = 1, ...N \quad \text{and} \quad f = 1, ..F$$

The embedding length is now fixed on the number of features F for each node N. To transfer it to a specific amount of outputs O, they are summed up and weighted by the learned parameters W. W therefore is of the size $[O \times F \times K \times 2]$. These dimensions imply that the learned parameters do not depend on the number of nodes and can be used for data points of varying grid sizes. The calculation is done for each node separately and therefore, the number of nodes is included in the output H of size $[N \times O]$. The final hidden representation output is calculated by:

$$H_{i,o} = \sum_{f=1}^{F} (X_{V_{i,f}}^K \times W_{o,f,:,:}) \qquad \forall i = 1, ...N \quad \text{and} \quad o = 1, ..O$$

This representation is then fed to a GRU, whose functions we have given in section 3.2.9. H_{t-1} refers to our hidden representation output of the diffusion convolution and X_t is the input vector of the respective node at time step t. Both are given to the GRU functions.

The following parameters can be set to use the first DCRNN layer in PyTorch Geometric Temporal. The number of input channels refers to the number of node features F, as long as no layer comes before that would influence it. The number of output channels refers to the size of the output variables *O*. *K* refers to the number of steps. Lastly, a boolean can be set to whether biases should be included in the calculations; only 'true' runs without errors in the implementation. The calculation is done for each node, so the size *N* does not need to be included in the parameters.

Non-linearity is in the Pytorch Geometric implementation set outside of the layer implementation, so we need to add an activation function. We chose the ReLU function, which also works with non-negative output values. Since we have a regression task, we add a linear layer last that transforms the output to a vector of size two for our two targets. The simplest version of our network looks like the following:

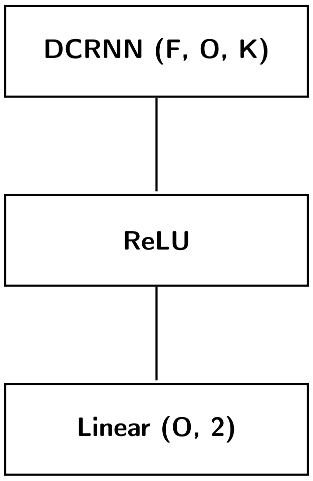

Figure 5.3.: The Temporal GCN in the first level of complexity with its three layers: DCRNN, ReLU, and Linear.

In our case *F* is set to 5, *O* to 40 (or 60 for more complexity) and *K* to 2. This configuration seemed the best solution in our tests. During subsection 5.2.4, we add more layers to add complexity and to combat overfitting.

5.2.3. Environment Setup

A specific environment has to be set up to test the PyTorch Geometric Temporal model. First, we need to check the available processing units. We tested on two different environments based on the processor, one on CPU and one on NVIDIA GPU. A Windows Machine was used, with a 64-bit operating system and Windows 10 Pro. The CPU is an Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz. On Windows devices, it can be checked for a graphical processing unit (GPU) in the device manager menu. We have an NVIDIA GeForce GTX 1050 GPU available. NVIDIA GPUs can be utilized for PyTorch processing.

It has to be decided on a development environment (IDE). PyTorch recommends using Anaconda as a package manager, which is helpful since specific Python and library versions need to be installed. We set up Anaconda 3, which installs Python 3.10 with it. For the IDE, Visual Studio (VS) Code is used. The Python and ipykernel extensions have to be installed. Anaconda needs to be selected as interpreter and a default command prompt enables interaction with Anaconda within VS. Since this paper supplies Jupyter Notebooks, this extension must also be installed. This IDE setup leads to more uncomplicated handling of the models and code.

Next, *PyTorch* needs to be installed. An Anaconda environment can be created within the command prompt with Python version 3.9, which PyTorch recommends. Python might need to be uninstalled and reinstalled within the environment in the correct version. The prerequisites, e.g., numpy installation, and correctly configured installation commands for PyTorch can be taken from here [127]. We used PyTorch 2. PyTorch can be installed for CPU or GPU if a CUDA-capable system is available. CUDA is a parallel computing platform developed by NVIDIA to speed up general computing tasks by utilizing the GPU [128]. The PyTorch installation will install a CUDA version with it; we used the newest version, 11.8. The correct installation can be checked with the following two lines of code: *import torch* & *torch.cuda.is_available()*. It might run into an error and switching the Conda PyTorch installation to the develop branch will be necessary. On the CPU version, PyTorch for CPU can be installed.

Next, the two packages *PyTorch Geometric (PyG)* [126] and *PyTorch Geometric Temporal (PyGT)* [13] must be installed. Within Anaconda, both can be installed via pip. PyG again makes the difference between CUDA and CPU installation. We used PyTorch 2.0. Other libraries might need to be installed throughout development as well but will be pointed out in the command outputs.

Finally, PyGT runs into problems working with the common CUDA *toDevice()* functions. To work with CUDA, the data, the model, and all its parameters need to be mounted onto the GPU. PyGT requires the data to be built with numpy arrays, which cannot be mounted to the device. Within the PyGT layers, mounting the parameters to the device also is not supported. They could be added with manual changes of each parameter to the *torch* data type and mounting to the CUDA device, if available, but this runs into many uncertainties. The issues section of the PyGT GitHub leaves this problem unresolved for now. However, PyGT supports the PyTorch Lightning [129] package.

PyTorch Lightning is a deep learning framework that can be used as an accelerator and can access the GPU. It requires the code to be strongly rewritten, but the PyGT library provides a short example, which we improved and added functionality. The train, validation, test and forward functions need to be rewritten. All of them include similar code. Lightning includes a validation step after each epoch to tune the hyperparameters and test the performance within training on unseen data. [129]

Lightning also integrates logging functionality within *TensorBoard* [130]. TensorBoard is a visualization toolkit built on TensorFlow for machine learning tasks. It can create helpful graphs from logged metrics. [130] Both libraries need to be installed. Afterwards, the board can be started within the Jupyter Notebook or the command line. Use the command *"tensorboard - -logdir ."* once in the conda command prompt to start the visualization board on localhost in the browser. An overview of the necessary tools to set up the PyGT environment on CPU and GPU is given in table Table 5.4. Some additional libraries or extensions will need

	CPU	GPU
Processing Unit	Intel Core	NVIDIA GPU
IDE	Anaconda (create environment & set as kernel) Python 3.9 Visual Studio Code (+ Python, ipykernel, Jupyter Notebook extensions)	
PyTorch	PyTorch 2.0 for CPU	CUDA-Capable: PyTorch 2.0 with CUDA 11.8
PyTorch Geometric	PyG 2.0 on CPU (pip install)	PyG 2.0 on CUDA 11.8 (pip develop install)
PyTorch Geometric Temporal	pip install	pip install PyTorch Lightning for GPU support (+ TensorBoard for visualizations)

Table 5.4.: Overview of tools for the PyGT environment for CPU and GPU.

to be installed but will be pointed out by the environment or outputs.

5.2.4. Model Tests

From this design and the PyTorch Geometric Temporal documentation, we implemented the first model on the CPU. We use the Adam optimizer and set our evaluation metric to the mean squared error (MSE), discussed in 3.2.2. MSE is a widely used metric to evaluate regression tasks. It returns absolute squared differences. The *root mean squared error (RMSE)* applies the root to the MSE values. [131] In this model, the MSE is calculated over all nodes within time intervals of the tested data points. As the problem at hand requires two targets, the MSE calculation is divided into two parts as follows:

$$\sum_{i=1}^{n} \frac{1}{n}((y_{1_i} - \hat{y}_{1_i})^2 + (y_{2_i} - \hat{y}_{2_i})^2)$$

Since the first output target θ has a higher variance, it will be valued higher in the cost. Additional weights could adjust the weight of the differences of each target in the cost but might lead to wrongly scaled updates.

It can be helpful to make the test values comparable as percentages. A percentage can be computed by scaling the MSE with the variance of the data. R^2 is commonly used to evaluate regression tasks and is shown to be one of the more informative metrics. It can be interpreted as the variance explainable by the model with a maximum value of one. Specifically, given data y and predicted data $\hat{y}$, the R^2 value is defined to be: [131]

$$R^2(y, \hat{y}) = 1 - \frac{\frac{1}{n}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(\bar{y}_i - y_i)^2} = 1 - \frac{MSE}{Var(y)}$$

The final Python code is added to the Addenda because the actual code is very syntax-specific. A.2 computes the PyTorch Geometric Temporal data structures. A.3 explains the network implementation as well as code for training in A.4 and testing on CPU in A.5. We first used a sample of 1000 same-size data points to improve the model formulation and adapt hyper-parameters. We start with tests on CPU, move to the GPU version, test for one whole data set and finally experiment with varying grid sizes. The experiments for best configurations, improvements of the architecture and calculations to present relative performance are discussed.

Beginning **tests on CPU**, the first problem of the model we had to circumvent is, as already noted, the fact that the model can only compute its matrix transformations for one given edge weight. We tested four possibilities:

- Only the Reactance
- Only the Capacity
- Reactance / Capacity
- Reactance · Capacity / (Sum of Capacities)

The tests showed that it only produces a difference in the MSE of less than 3%. The last option produced the smallest MSE but also the lowest variance in the result. Therefore only the capacity was chosen as edge weight, with an MSE only slightly lower but more variance in the results.

The model at the first complexity level returned test MSEs of more than $5.36{\cdot}10^5$, which would refer to an RSE of more than 730 difference for both targets. These results show severe underfitting of the model and hint that adding complexity is needed. We did this by adding more layers and increasing the number of parameters. In the second complexity level, we added another linear layer, which improved the MSE by more than 30% towards $3.53{\cdot}10^5$ (RMSE of 590). Complexity was also expanded by raising the number of parameters in the given layers, but this did not create much effect on the result. More filters in the DCRNN for example, improve the MSE by a little more than 1%, while more kernels or steps in the DCRNN worsened the results.

To evaluate the model, we also looked at the difference between the training and the test error. Throughout the training, the RMSE training loss was reduced by almost 80%, see the figure Figure 5.4

The training error almost halves the test error at this point, so we added methods to combat overfitting. We included dropout and normalization layers into the architecture of the network. *Dropout* is a regularization method that thins out the network by randomly dropping neurons and their connections as part of the training process. It has proven potential to significantly reduce overfitting. [132] The dropout again strongly improved the test MSE by more than 60% towards an MSE at $1.64{\cdot}10^5$ and an RMSE of 400. We experimented with different *normalization* layers on the input (or activation values), not discovering substantial changes. The BatchNorm layer from [133] resulted in the lowest test error. It creates the same value distribution at the layers' inputs for all training batches, ultimately limiting the vanishing gradient problem. It can also be applied to all activation layer outputs. [133] We

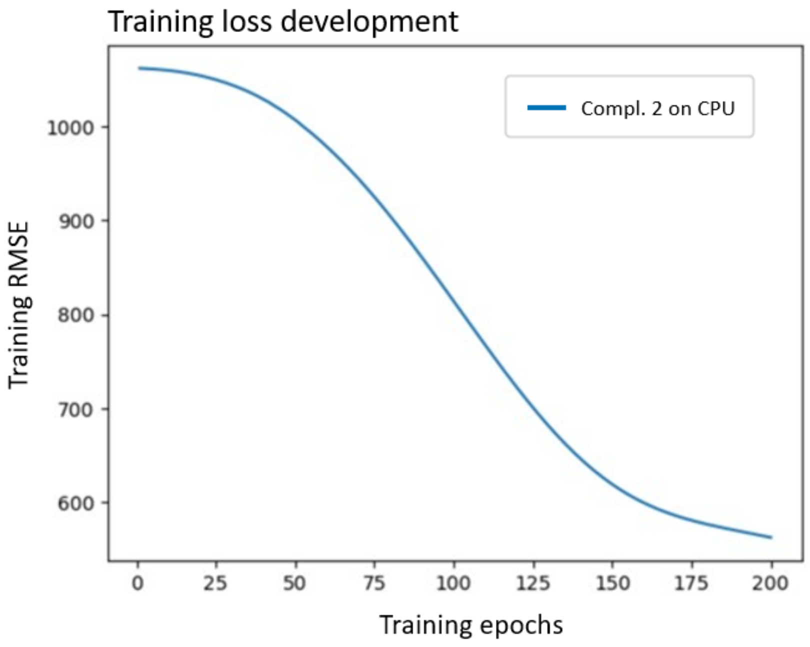

Figure 5.4.: RMSE training error reduction per epoch within the sample of one data set at complexity level 1-2.

refer to the architecture design at this stage as complexity level two. Its design is depicted in graphic Figure 5.5 and in the code example A.3.

To reduce the training time and train the model for larger data sets, we implemented the model on **GPU with PyTorch Lightning**. A.6 and A.7 show the code for GPU. For the CPU version, we have a training time of 20-40 minutes. On the GPU, the time more than halved between 10-15 minutes. The biggest time wastes occur with more kernels in the DCRNN layer or other layer types, but both changes were rejected due to lower test performance. With the Lightning implementation, there is a validation step at the end of each training epoch. This helps to fine-tune the model's hyperparameters on unseen data, resulting in improved test results. Additionally, the model works with higher learning rates (and a lower number of epochs. *Epochs* defines how often the training data will be traversed. [134]. It leads to good results after the first epoch and only slight improvements over the other epochs when applying the high learning rate. With the same model and the defaults on epochs and learning rate, we achieve a lower test MSE of $1.34 \cdot 10^5$ compared to the last CPU test.

Lightning offers an early stop flag once the training loss stabilizes. It also sets the default learning rate to $1 \cdot 10^{-2}$. We offer a comparison in Figure 5.6 of the default configuration epochs versus the same as on the CPU. It shows in the pink graph the training MSE with the default values of Lightning, a learning rate of $1 \cdot 10^{-2}$ and 10 epochs through the early stop callback. The second was trained with 100 epochs. A high improvement in the first epoch of the default configuration can be seen and only slight improvements after. The more detailed configuration allows for slightly lower MSE through longer training, but the training

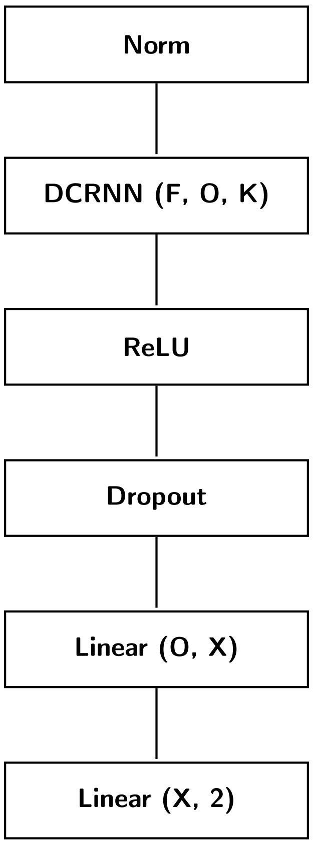

Figure 5.5.: The temporal GCN in the second level of complexity with its six layers: DCRNN, ReLU, dropout and linear.

time doubles (14 to 35 minutes). Even higher epochs or lower learning rates were not further explored, because they led to worse results in the test loss, strong oscillations in the validation loss, and a high scaling of the training time.

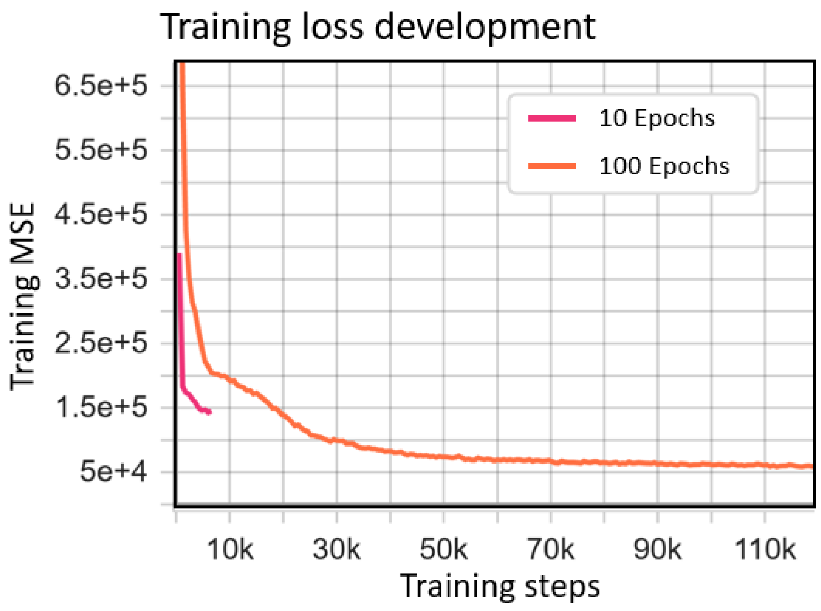

Figure 5.6.: MSE training loss development over training steps of different number of epochs on GPU with Lightning.

We compared the model's predictions against the targets per node. For a simple model within complexity level two, the model strongly underfits for θ, while it produces more promising values for generation. For θ, we have the problem of similar values for some (potentially connected) nodes and repetitive outliers for quite different nodes, for example, these who produce a lot. The model tends to average out the similar nodes and therefore does not provide sufficient complexity. For generation, the problem lies in the combination of a logistic and regression problem, which is predicted only with regression. The high amount of zero values leads to the effect that the model averages all values towards the middle. This effect is well visible in a simple model of complexity level two in Figure 5.7. Post-processing rules on target two can improve the outcome of target two but will have a negligible impact on the loss overall.

These limitations in the prediction of each target lead us to add another layer of complexity with a second temporal layer and another normalization layer between layers to emphasize the differences between features. This model produces to our best result for the sample set of one grid size of $1.13 \cdot 10^5$, where the number of filters has increased to sixty and the learning rate lowered to $1 \cdot 10^{-2}$. While target one (θ) is getting closer, we now see problems in predicting target two (generation). Both targets exhibit very different ranges of solution values and different levels of complexity. Predicting only θ does not improve the results additionally. The results of target two seem to follow target one, which renders them unusable. The predicted

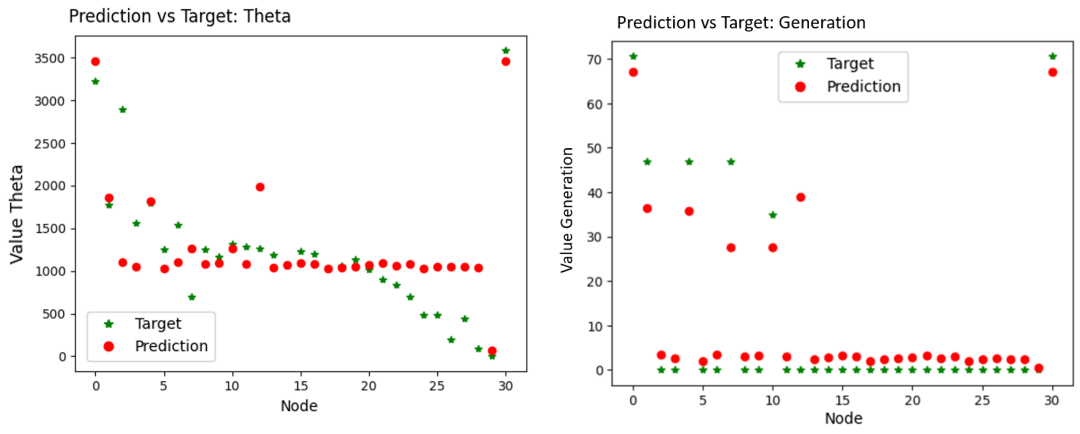

Figure 5.7.: RMSE training error reduction per epoch within the sample of one data set at complexity level 2.

versus targeted values per node for the prediction on the sample with complexity level three is depicted in Figure 5.8

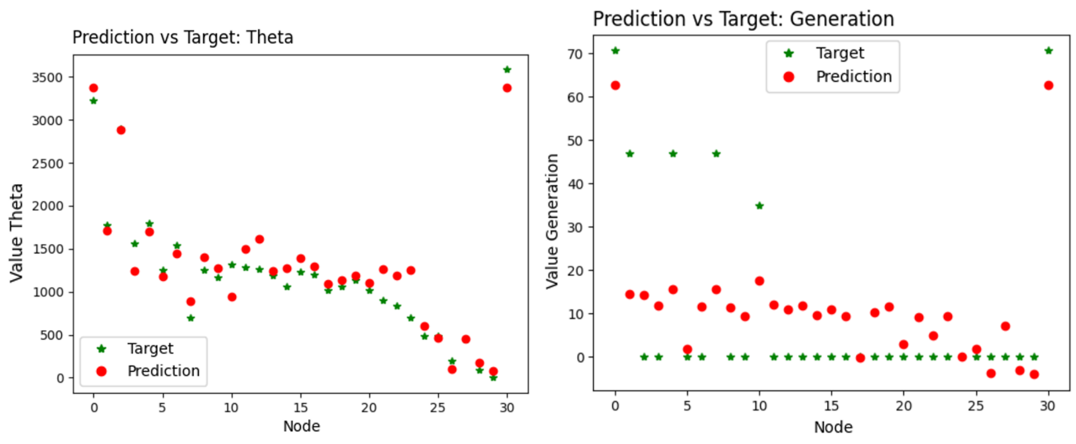

Figure 5.8.: RMSE training error reduction per epoch within the sample of one data set at complexity level 3.

The training error almost halves the test at $7.27{\cdot}10^4$. However, further efforts to combat overfitting, such as regularization with weight decay, dropout, learning rate or additional normalization, did not result in better test performance. Also, experimenting with other layers or more parameters showed no relevant effect. Not learning additive biases by setting bias to false resulted in a compilation error within the implementation. We concluded that the performance will stagnate at this level and stopped the tests.

The training error for the three complexity levels is presented in Figure 5.9. The epoch number is chosen by the early stop flag. We can see that the higher complexity level yields better training results, especially for θ. This holds true for validation and testing. The second two complexity levels have great improvements in the first training steps but then saturate. This is also not combated by a lower learning rate or more epochs.

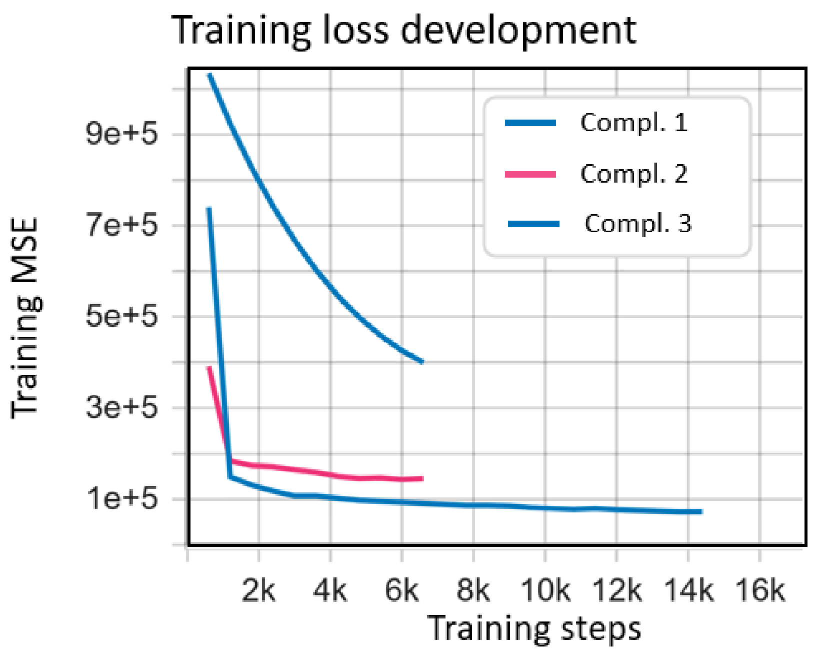

Figure 5.9.: Comparison of MSE training error of different complexity levels on GPU.

We provide some calculations to make these MSE and RSE values relative. Within the sample, for θ (rounded to full numbers) the minimal value is zero and the maximum value 5245. The average is 903 and the variance of all values is 312307. For generation (rounded to full numbers), the minimum value is zero and the maximum is 110. The average is eight and the variance of all values is 320. For the MSE at this point, the value lays at $1.13 \cdot 10^5$ for testing. This results in an R^2 of 64%. In this calculation, most of the weight lies on the prediction of θ. Simply put, this would categorize the model below "ok" [135].

Often models from the literature review, such as the highlighted work [11] using GNNs, only predict the generation. θ is still needed for the MIP optimization to have a complete solution from which the constraints can be calculated or new solutions can be deduced. We tested to only predict the generation, our target two, with the best model and parameters for complexity level two on the sample data. It showed the most promising results in the direct comparison of prediction and target per value. It took 34 minutes to train this model for 100 epochs. We receive a MSE of 106, which is an RMSE of 10.3. It translates to an R^2 of 66.9%, which almost categorizes as a good model. The closest paper of Owerko [11] was able to produce a RMSE below one. However, this metric does not include comparability or normalization, e.g., through division by the data variance. Another paper from the literature review, which predicts the generation dispatch and states their metric as R^2, was able to increase the explainable variance to 90% [60].

From a time perspective, the training takes around fifteen minutes. The testing takes five seconds, including printing, with around 36 iterations per second and two-hundred test instances. For the prediction of only one target, it takes four sections with almost 50 iterations per second. The pure Gurobi optimization takes between 0.03 and 0.1 seconds per model, with a more than ten times increase through the configured printing steps. The ML model halves the Gurobi time, which depicts an average of 0.05 seconds per MIP optimization and 200 data points, resulting in 10 seconds (compared to five).

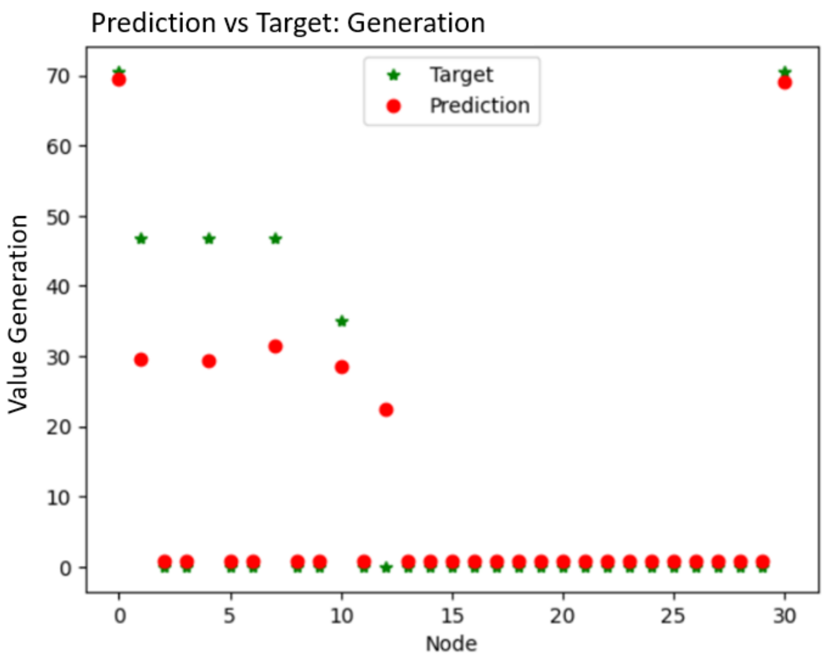

Figure 5.10.: Comparison of Prediction and Target values for only predicting the generation on sample data set.

Next, we tested the model for more data points, specifically **one entire data set** with about ten thousand data points of the same grid size. It refers to one year in our data model. The MSE can be lowered again to 8.68·10^4 at best with complexity level two. Test error reduces to an RMSE of about 295 for both targets together. Some efforts to combat overfitting were applied, like weight decay for regularization, but only improvements with one more dropout layer were found. For this amount of data, the variance reduces to 267082. The R^2 is at 0.675, close to the threshold for "ok" (at 0.7). It is to be noticed that in the provided data a few data points for specific outliers exceed the capacity in the optimization, the reason is unclear. This might reduce the MSE and increase the variance, but it influences less than 3% of the data.

The best configurations throughout the different stages of the test are given in Table 5.5.

In order to make the model performance with all data comparable to the other prediction models, we train it again to predict only the generation as a baseline. The training time reaches values above 40. We use complexity level two again to predict target two, with another dropout layer, 60 filters, learning rate or $1 \cdot 10^{-3}$ and weight decay of $1 \cdot 10^{-5}$. We achieve an MSE of 43, relating to an RMSE of 6.5. The variance of the target is 475.2, so we achieve a

Data	Model	Parameters	Time (min)	Training Error (first and last epoch)	Test Error	Accuracy (R^2)
Sample (1.000)	CPU Compl. 2	Limit as edge weight K: 2 Epochs: 200	36		$1.64 \cdot 10^5$ RMSE: 400	Below 0.5
	GPU Compl. 2	Filters: 40 lr: $1 \cdot 10^{-2}$ Epochs: 10	14	$3.91 \cdot 10^5$ to $1.45 \cdot 10^5$	$1.34 \cdot 10^5$	Below 0.5
		Epochs: 100	25	$2.14 \cdot 10^5$ to $1.01 \cdot 10^5$	$1.23 \cdot 10^5$	Below 0.5
	Compl. 3	Filters: 60 lr: $1 \cdot 10^{-3}$ Epochs: 24	14	$7.42 \cdot 10^5$ to $7.27 \cdot 10^4$	$1.13 \cdot 10^5$	0.64
Full (10.000)	Compl. 3 + Dropout	Epochs: 10	30	$1.45 \cdot 10^5$ to $8.96 \cdot 10^4$	$8.68 \cdot 10^4$ RMSE: 295	0.675

Table 5.5.: Overview of the best configurations throughout the different tests and their results for both targets.

Data	Model	Parameters	Time	Test Error	Accuracy
Sample (1.000)	Compl. 2	lr: $1 \cdot 10^{-2}$ Filters:40 Epochs: 85	34	106 RMSE: 10.3	0.67
Full (10.000)	Compl. 3 + Dropout	lr: $1 \cdot 10^{-3}$ Filters: 60 Epoch: 23	70	43 RMSE: 6.5	0.91

Table 5.6.: Overview of the best configurations throughout the tests and their results for one target generation.

very good R^2 of 0.91. This is slightly higher than the R^2 found in the prediction of generation dispatch within [60]. The predicted values printed against the actual values can be taken from Figure 5.11. The logistic part is predicted well, but high outliers are underestimated. In some cases, the model outputs predictions above the target, which often refers to the maximum. We additionally provide a table of the best tests predicting only the generation in Table 5.6. The computation time per instance remains the same for the full data set (35 iterations per second), so the Gurobi and ML test time relation remains the same at a 50% decrease for the ML model.

With a single grid size, the models were able to predict the generation dispatch as accomplished in previous research projects. However, the performances already decreased when including the second target of phase angles towards an 'ok' rating. **Feeding the model varying grid sizes** as input data is still an essential goal of this model. The model was already implemented in a way that it could take variable grid sizes for all the tests and the code was not adapted. We tested the model first with two sizes, later with three. Even though the calculations were possible, the results highly depended on the ratio of similar grid data to different grid data. When we have a high amount of other grid-sized data in the training

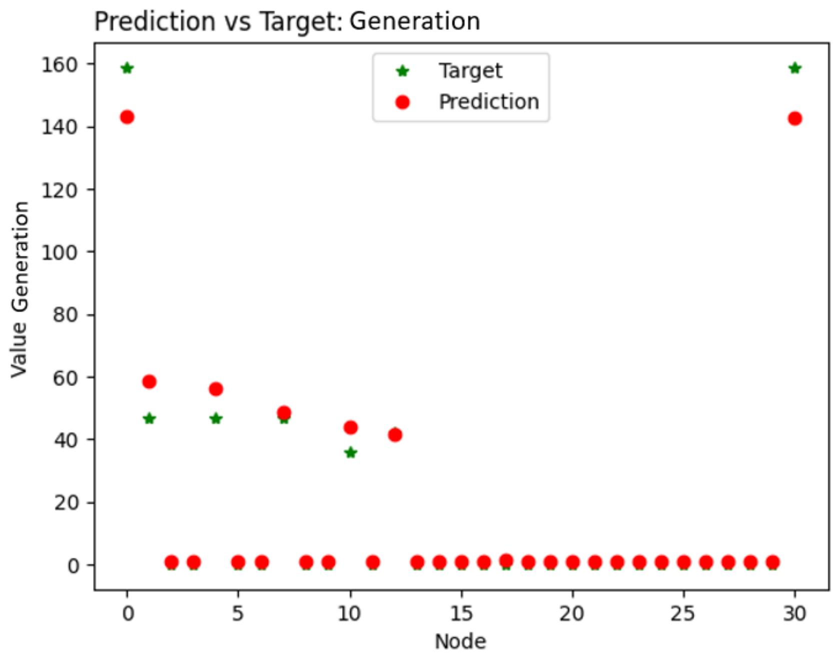

Figure 5.11.: Comparison of prediction and target values for only predicting the generation on full data set.

set (ten thousand of the old data set, ten thousand of the predicted data set, resulting in a training ratio of 10 to two), the test MSE returns $7{\cdot}10^5$. We realized that the more data we have that is far from the test and the further it is away, the worse the results will be. For a small amount of other grid size data (one thousand to eight thousand), the test results were similar with MSE at $1.06{\cdot}10^5$. When we test with three different grid sizes, where we have two new in training, the result reaches $8.12{\cdot}10^5$. The R^2 even turns negative in these cases. The data of the different data sets have a high difference in the predicted values, which the model cannot overcome. The test results suggest that the model is inapplicable for this task since high accuracy cannot be given and performance is dependent on changes to the grid.

5.2.5. Challenges and Conclusions of PyTorch Geometric Temporal for OPF

We now discuss the challenges and conclusions of the PYG Temporal model for OPF optimization with varying grid sizes. Since the model cannot guarantee exact solutions or feasibility, the model has the potential to be used as a vague starting point heuristic that is quite fast. We encountered some problems relating to the data, the model implementation and the evaluation metric. Since the data was taken from natural sources and adjusted through node splitting or node concatenation, the data exhibited a high variance related to the temporal state. This variance of target values was often quite different between training, validation and test data and made the prediction difficult. The model is not fully able to overcome this variance in the data, thus having low predictions especially when combining different data sets. Other problems with the data are the two targets that are very different in range. The target generation additionally encompasses a logistic and linear part of the result.

The implementation of the model poses some limitations. First of all, one is limited to one edge weight. Only a few configuration options are given within the temporal layers, even setting the biases to false results in a compilation error. The model also seemed to be highly dependent on the variance of the data, which is unfortunate given realistic data and changing grid sizes. If the results depend strongly on the temporal changes within the data, one solution could be to retrain the model continuously. Since we already work with one thousand time intervals and the model runs for less than an hour, this should be realizable.

For the evaluation metric, only one combined cost function can be given, even with two values. Since one target range is significantly lower than the other, the combination of both into the loss function would need to be adjusted. The evaluation metric to calculate accuracy also does not fully encompass the data's variance. Only the variance through all data points is regarded, while we have variance in at least two dimensions via nodes and time. It is to be discussed whether the variance calculation should be adjusted for temporal graph-related cases.

Overall, we conclude that the model is able to handle this kind of data structure. If the variance of the data is controlled, the model can probably predict good results. The data had some challenging changes within the sets but shows significant learning capacities and some improvements by tuning the model and the parameters. Post-processing rules can be applied to stay within the capacity limits. These kind of models are in very early development stages. Better results will be possible in the future with some improvements to the frameworks.

6. Discussion

This thesis explored improving OPF optimization with ML, including varying grid sizes. Participants of the energy market are connected to a bus in the energy grid, which is dynamic through their addition or deletion. The OPF problem requires predicting the phase angles, defining energy transfer between buses, and the energy generation loads per bus. The standard prediction using MIP faces challenges with the changes in the energy markets, such as renewable sources and real-time demand, which mandate faster optimization at higher complexity energy systems. Machine learning can predict outcomes in real time after being trained, which has the potential to accelerate the energy market clearing process of the real-time market and therefore enable shorter optimization intervals.

For general MIP acceleration with ML, methods on the branching strategy, search strategy and pruning, (primal) heuristics, and cutting are inspected in research. The methods are closely related to improving the search on the B&B tree or decreasing the solution space. ML models on OPF directly predict the generation dispatch, predict the unit commitment problem, predict warm starting points, select active constraints or find stability constraints. So, they can be divided into directly predicting (a part of) the solution or decreasing the optimization time by choosing the needed constraints. If the direct prediction proves inexact, the predictions can be used as warm starting points for OPF or adjusted by post-processing rules. For indirect methods, false negatives in selecting constraints can also lead to deviations from a feasible solution. The only ML improvements with a feasibility guarantee are choosing strategies or heuristics for B&B.

In this paper, we worked with an adapted version of the DC-OPF model, as in [6]. It requires grid data, referring to information on node identities and grid connections that are mostly seen as static in papers so far, and market/time series data, provided as bids of every participant for every optimization interval. The amount of publicly available energy data is very limited, often due to confidentiality agreements involved. As a result, few papers on ML and OPF utilized data from operational sources. For grid data, many academic projects train with individual grids of the IEEE PES PGLib OPF benchmark library and few with self-generated grids. Market data is sampled or, in a few cases, collected from American ISOs, like PJM and MISO.

To find operational data that accounts for more realistic variance, we searched for public data sources encompassing all the needed data points. We figured that only combinations of data sources can provide all data variables, requiring manual adjustments. The data sources can be divided into three types. Energy operator statistics provide a voluntary publication of real generation and demand market data directly by the energy operators, which only has the downside of poor management and many missing points. The energy data libraries are commonly known and offer a collection of aggregated and abstracted versions of available data, often lacking the required level of detail. The generation tools are research projects to produce energy grids from available market or infrastructure data. They only provide little time series data. Three appropriate data selection strategies can be based on these data sources. For Germany, ELMOD offers the data with almost all the needed information except the cost. For Europe, the ENTSO-E market data can be paired with SciGRID or the work of [104]. The IEEE grids can be connected to the PJM data for the American market, the only data collection strategy already leveraged in research, e.g., by [61]. For this reason, we chose IEEE PES PGLib and PJM as data sources. The other two data collection strategies can be explored in further research projects.

Since no ready-to-use database is available, we needed to develop a process to generate usable dynamic grid data sets from these public sources. The PJM data required much data cleaning, especially filling empty data points. Specific filling methods to keep the patterns of variance in the data must be selected, such as generation from historical or similar data points to account for these data holes, but some realism will be lost. The two most cumbersome data preparation steps were calculating the marginal energy generation cost, for which we developed a function based on [117], and normalization efforts to create feasible data. We aimed to create a model to predict feasible solutions, as otherwise, the MIP optimizers would apply strong simplifications and knowledge would be lost from the perspective of the ML model. Since we could only use one static energy grid as a database, we needed to generate dynamic grid data from it. We chose the common methods in graph theory node concatenation and node splitting, to generate feasible problems of different sizes from the grid and market data. Again, this required some abstraction from real data. The steps have proven to result in feasible data sets with the Gurobi MIP optimization. The resulting calculations, code and data can be applied in future research projects.

Many newer successful approaches apply NN models. However, they do not mention efforts to include variable grid sizes in their input data or networks. A few papers in MIP and OPF optimization in ML have the potential to leverage variable grid sizes. Most of them use GNNs. The paper of [10] was identified as the closest solution to our goal, which applies a GCN to create embeddings and then adds an imitation learning model for prediction. There is yet to be a standard pipeline for this approach as two different learning frameworks need to be incorporated. This approach resulted in very good test results while testing for one grid size. However, they only predict the generation load and use only a few data variables within the AC-OPF model. Also, they use sampled data, meaning fixed variance, and only promote the absolute RMSE. We recommend further exploring this method, introducing a standard pipeline and testing its robustness to different grid sizes.

As NNs only work with one-dimensional input, an embedding needs to be created. Finding a meaningful combination independent of grid size using linear methods is difficult. Convolution or pooling layers require strong cutting of the input matrices and the extraction of meaningful underlying features remains questionable. Some additional effort can be applied to find the optimal combination of layers and their configurations in a GCN for this purpose. RNNs cannot encompass all the needed information, spatio-temporal relations and edge features, in their embeddings. This leaves methods including some GNN. Usual (convolutional) GNNs calculate within one graph and multi-graph GNNs only predict on the graph level. The only ready-to-use NN architecture or training pipeline able to handle this data structure is a dynamic temporal/recursive GNN.

For the implementation of the RGNN model, the PyTorch Geometric Temporal library was used, because it is open source and covers the needed data structure with discrete time snapshots/signals of dynamic graphs. It adds fully implemented temporal layers and a fitting training pipeline to PyTorch Geometric. The DCRNN layer by [14] showed the best performance in our set of tests while being simple to apply and being able to include most of the data variables. Only the edge weights needed to be summarized into one value, which did not seem to impact the accuracy in the tests. Another downside of the DCRNN layer is that one is very limited in the configuration possibilities; even setting the bias as excluded resulted in errors.

We chose to set a linear function as the output function and only predicted the two linear regression parameters, phase angle and generation volume. This approach does not lose information for the predictions because the unit commitment can be deduced from the generation volume. Otherwise, volume and unit commitment might contradict. Both output variables provide different ranges of values and require different levels of complexity, which posed challenges to the model. For feasibility, the predicted solutions must fit the solution space defined by the constraints. With increasing accuracy, predictions approach feasibility. In practice, feasibility checks need to be applied.

The problem of underfitting in the first design of the network was combatted with methods to increase model complexity. Adding more layers had the best effect. Then efforts were needed to reduce overfitting, which was mainly overcome by training with more data points and including the validation step with training on GPU. Additional experiments on normalization, dropout, regularization and adjusting different hyperparameters resulted in small positive effects.

Eventually, the accuracy performance stagnated for one full data set (almost ten-thousand instances) up to 0.675 in R^2 with a test MSE of $8.68 \cdot 10^4$ and RMSE of 295 for both targets, phase angle and generation volume. This declares the model as 'ok' by common categorization, which is often criticized for lacking adaptability. When we only predicted the generation volume as a baseline, we achieved an RMSE of 6.5 and a R^2 up to 0.91 (give or take regarding data leakage), which categorizes as 'very good' and competes with the performance of comparable solutions from the literature review [60].

The test or prediction time was, on average, 0.05 seconds for the MIP optimization with Gurobi and 0.025 for the ML model, but the Gurobi optimization times displayed stronger variance in optimization time for more complicated models (0.03 until 0.1 seconds). This leads to a 50% time saving with the ML model on this test scale (without adjustments for feasibility) and can likely be increased for more complicated MIP models.

Lastly, the model's generalization capacity regarding variable grid sizes was tested by including two and then three different data sets in the training. The performance depends strongly on the ratio of the same and differently sized grids in the training sample to the test set. Most tests resulted in negative or untrackable R^2. The results hint that the model is highly volatile to this kind of temporal variance. It is to be considered though, that the evaluation metric depends on the mean squared error and the general variance of the solution. The fact that we have two different targets cannot be properly included in the loss function and only a (weighted) addition of both deviations is possible. The variance of this data structure has two dimensions, one between nodes and one between instances of the same node along the time steps, while only the averaged variance is considered. We argue that for spatio-temporal data, a specific evaluation metric is necessary.

The overall approach and its implementation were able to encompass the data structure with its variables and dynamic nature. This model slightly improves the baseline accuracy from previous research to predict the generation volume. The accuracy reaches medium values for also predicting phase angles. The fact that performance is highly dependent on changes to the grid suggests that the approach may be unfit for this task at this development stage.

We introduced four simplification steps to the real-world setting, starting from the simplified market clearing model in [6], which offers a more realistic formulation of the DC-OPF model. First, we only considered a one-sided market, one pair of generation and demand loads per bus. It means we have one type of participant per bus and one bid per participant. Additionally, we excluded inter-time relations and present inflexible demand. These assumptions were necessary for the functioning of the ML model and can easily be applied to real market data, as mainly aggregation is needed.

In order to reintroduce that information from a mathematical side, more matrices can be added with this information to assign them to the bus. In the GNN model, the previous temporal state could be included in the node's embedding as another variable. Introducing sub-nodes could track the different bids per participant and participants per node in the graph as their child nodes. Sub-nodes in the graph data are already supported by PyTorch Geometric, yet not the temporal extension library. Otherwise, pre- and post-processing rules can wrap and unwrap the data.

We inspected another step towards more operational input data for academic ML on OPF experiments. Bigger networks, more constraints and higher variance can additionally disturb performances. In practice, a feasibility check needs to be applied. Lower accuracy predictions can be used as warm starting points or be improved with post-processing rules. The research on dynamic temporal GNNs is rather new and in development. With time, better results should be possible. We advise further tests of these architectures on real-world problems.

A. General Addenda

A.1. Repository

The created data sets, the Python code scripts and discussed test logs can be found at our repository [15].

A.2. PyGT Graph Dataset Preparation

We added syntax-heavy code to the addenda.

The following code creates graphs that can be processed by the PyTorch Geometric framework from the busses and edges tables. We read the two tables, introduce helper functions and build the correct structures for the attributes for each time interval. At last, the *DynamicGraphTemporalSignal* dataset is created. The code is written in Python. This code works both for the CPU and the GPU version with PyTorch Lightning. The code was created using the PyTorch Geometric [126] and PyTorch Geometric Temporal [13] introduction chapters of the documentation as well as the GitHub Repositories for code review.

```
import torch
import pandas as pd
import numpy as np

# Read busses and edges
busses =pd.read_excel('Busses_Timeseries_Table_Set2022_Solution.xlsx')
edges =pd.read_excel('Edges_Timeseries_Table_Set2022.xlsx')

#Functions to create graph from (busses, edges)-sample
def sourceNodes(timestamp):
    source =edges.loc[edges['datetime_beginning_utc'] ==timestamp, 'From Number']
    target =edges.loc[edges['datetime_beginning_utc'] ==timestamp, 'To Number']
    sourceNp =pd.concat([source, target])
    targetNp =pd.concat([target, source])
    return (sourceNp, targetNp)

def nodeAttributes(timestamp):
    attributes =busses.loc[busses['datetime_beginning_utc'] ==timestamp, ['mw_norm', '
                                                    incremental_price', 'inter_start_cost', '
                                                    min_ecomin', 'max_ecomax']]
    return attributes.to_numpy()

def edgeAttributes(timestamp):
    attributes =edges.loc[edges['datetime_beginning_utc'] ==timestamp, ['X', 'Lim MVA A']]
    weights =attributes['Lim MVA A'] # The models can not handle multiple edge features ->
                                                   choose or connect
    attributes2 =pd.concat([weights, weights], axis=0).to_numpy()
    return attributes2

# Approach to integrate reactance and capactiy
def edgeAttributesConnected(timestamp):
    attributes =edges.loc[edges['datetime_beginning_utc'] ==timestamp, ['X', 'Lim MVA A']]
    sumLimit =sum(attributes['Lim MVA A'])
    weights =attributes['X'] *attributes['Lim MVA A'] /sumLimit
    attributes2 =pd.concat([weights, weights], axis=0).to_numpy()
    return attributes2

def solAttributes(timestamp):
    attributes =busses.loc[busses['datetime_beginning_utc'] ==timestamp, ['solTheta', '
                                                   solGenerate']]
    return attributes.to_numpy()

# To only test for generation prediction
def solAttributesT1(timestamp):
    attributes =busses.loc[busses['datetime_beginning_utc'] ==timestamp, ['solTheta']]
    return attributes.to_numpy()

# Temporal data snapshots are PyTorch Geometric data object for each snapshot
# Iterated by DynamicGraphTemporalSignal

import torch_geometric_temporal
from torch_geometric_temporal.signal.dynamic_graph_temporal_signal import
                                                DynamicGraphTemporalSignal
timestamps =busses['datetime_beginning_utc'].unique()
numGraph =len(timestamps)
edge_indices =list()
features =list()
targets =list()
edge_weights =list()
# Append graphs per time
timeInterval =0
for timeInterval in range(numGraph):
    timestamp =pd.Timestamp(timestamps[timeInterval])
    source, target =sourceNodes(timestamp)
    edge_index =np.array([source, target], dtype=np.float32)
    x =np.array(nodeAttributes(timestamp), dtype=np.float32)
    y =np.array(solAttributes(timestamp), dtype=np.float32)
    edge_attr =np.array(edgeAttributes(timestamp), dtype=np.float32)
    edge_indices.append(edge_index)
    features.append(x)
    targets.append(y)
    edge_weights.append(edge_attr)

dataset =DynamicGraphTemporalSignal(edge_indices, edge_weights, features, targets)
```

A.3. PyGT CPU Model (Complexity level 2)

We first built the native CPU model, which is explained in the documentation as default. We created different models with bigger changes in complexity, which we refer to as Complexity Levels. This example refers to Complexity Level 2. It uses a DCRNN layer, two linear layers as well as Normalization and Dropout layers to combat overfitting. Filters refer to the number of output channels in the DCRNN layer. The number of output channels can be adapted. The code was created using the PyTorch Geometric Temporal [13] introduction chapters of the documentation as well as the GitHub Repository for detailed code inspection.

```
# (Dynamic Convolutional) Recurrent Graph Neural Network
# Complexity Level 2 with Normalization and Dropout
# DCRNN layer & feedforward network
import torch.nn.functional as F
from torch_geometric.nn.norm import BatchNorm
from torch_geometric_temporal.nn.recurrent import DCRNN

class RecurrentGCN(torch.nn.Module):
    def __init__(self, node_features, filters):
        super(RecurrentGCN, self).__init__()
        self.bn =BatchNorm(node_features)
        self.recurrent =DCRNN(node_features, filters, 2)
        #Arguments: In, Out (hyper-param, many for Complexity), K (#neighbors/steps, take 2)
        self.linear1 =torch.nn.Linear(filters, 20)
        self.linear2 =torch.nn.Linear(20, 2) # 2 Outputs

    def forward(self, x, edge_index, edge_weight):
        x =self.bn(x) # Normalization layer
        h =self.recurrent(x, edge_index, edge_weight) #DCRNN: GCNN + GRU
        h =F.dropout(h, training=self.training) # Dropout against Overfitting
        h =F.relu(h) # Non-Linearity
        h =self.linear1(h)
        h =self.linear2(h) # Compute Regression Output
        return h
```

A.4. PyGT CPU Model Training

The training procedure is defined in batches. We used the Adam optimizer. The learning rate as well as weight decay for regularization can be set as hyperparameters. The number of epochs can be set. Mean Squared Error (MSE) is used as an error metric. The cost for both target 1 and target 2 is stored during training, for later evaluation. The code was adapted from the PyTorch Geometric Temporal [13] introduction chapters of the documentation.

```
from tqdm import tqdm

model =RecurrentGCN(node_features=5, filters=40)
optimizer =torch.optim.Adam(model.parameters(), lr=1e-3, weight-decay=1e-5) # Learning Rate
                                                and Regularization adaptable
criterion =torch.nn.MSELoss() # MSE as Error Metric
```

```
costs =list()
model.train()

for epoch in tqdm(range(100)): # Number of Epochs adaptable
    cost =0
    costT1 =0
    costT2 =0
    for time, snapshot in enumerate(train_dataset):
        y_hat =model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
        cost =cost +criterion(y_hat, snapshot.y)
    cost =cost /(time+1)

    #Track Costs
    costs.append(cost.detach().numpy())

    cost.backward()
    optimizer.step()
    optimizer.zero_grad()
```

A.5. PyGT CPU Model Test

For the testing of the model, we compute the MSE over all test data points and store the predictions. The code was adapted from the PyTorch Geometric Temporal [13] introduction chapters of the documentation.

```
model.eval()
cost =0
predictions =list()
for time, snapshot in enumerate(test_dataset):
    y_hat =model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
    predictions.append(y_hat)
    cost =cost +criterion(y_hat, snapshot.y)
cost =cost /(time+1)
cost =cost.item()
print("MSE: {:.4f}".format(cost))
```

A.6. PyGT GPU Model (Complexity Level 2))

To run faster tests on GPU, the code had to be adapted to PyTorchLightning. It detects and uses the CUDA GPU. To define a model, the layers have to be set and then reused in the definition on each model function. These are forward to compute predictions, training_step, validation_step and test_step. The calculations are performed on the respective data batches, where the parameters are extracted. MSE is used as the cost function and is logged throughout the process. The Logs create a new folder in the IDE and can be displayed by TensorFlow. The code was adapted from the PyGT GitHub Lightning example [13] and the Lightning documentation [129].

```
# Complexity Level 2 in PyTorch Lightning
import pytorch_lightning as pl

from torch.nn import functional as F
from torch_geometric.nn.norm import BatchNorm
from torch_geometric_temporal.nn.recurrent import DCRNN


class DCRNNModel(pl.LightningModule):

    def __init__(self, node_features, filters):
        super().__init__()
        self.bn1 =BatchNorm(node_features)
        self.recurrent =DCRNN(node_features, filters, 2)
        self.linear1 =torch.nn.Linear(filters, 20)
        self.linear2 =torch.nn.Linear(20, 2) #Two Targets

    def configure_optimizers(self):
        optimizer =torch.optim.Adam(self.parameters(), lr=1e-2, weight_decay=1e-5) # Learning
                                                        Rate and Regularization adjustable
        return optimizer

    def forward(self, test_batch): #For predictions
        x =test_batch.x
        edge_index =test_batch.edge_index
        edge_weight =test_batch.edge_attr
        x =self.bn1(x) # Normalization
        h =self.recurrent(x, edge_index, edge_weight) # Temporal Layer
        h =F.relu(h) # Activation Function / Non-liearity
        h =F.dropout(h, training=self.training) #O verfitting
        h =self.linear1(h)
        h =self.linear2(h) #Linear Regression
        return h

    def training_step(self, train_batch, batch_idx):
        x =train_batch.x # Training in Batches on Training Data
        y =train_batch.y
        edge_index =train_batch.edge_index
        edge_weight =train_batch.edge_attr
        x =self.bn1(x) # Same model, repeated to fit pipeline
        h =self.recurrent(x, edge_index, edge_weight)
        h =F.relu(h)
        h =F.dropout(h, training=self.training)
        h =self.linear1(h)
        h =self.linear2(h)
        loss =F.mse_loss(h, y) # Cost Function
        self.log("train_loss", loss, on_epoch=True) # Log Results
        return loss

    def validation_step(self, val_batch, batch_idx): # Validation stpe after each Epoch
        x =val_batch.x
        y =val_batch.y
        edge_index =val_batch.edge_index
        edge_weight =val_batch.edge_attr
        x =self.bn1(x)
        h =self.recurrent(x, edge_index, edge_weight)
        h =F.relu(h)
        h =F.dropout(h, training=self.training)
        h =self.linear1(h)
        h =self.linear2(h)
        loss =F.mse_loss(h, y)
        metrics ={'val_loss': loss}
        self.log_dict(metrics)
        return metrics

    def test_step(self, test_batch, batch_idx): # Compute Test MSE
        x =test_batch.x
        y =test_batch.y
        edge_index =test_batch.edge_index
        edge_weight =test_batch.edge_attr
        x =self.bn1(x)
        h =self.recurrent(x, edge_index, edge_weight)
        h =F.relu(h)
        h =F.dropout(h, training=self.training)
        h =self.linear1(h)
        y_hat =self.linear2(h)
        test_loss =F.mse_loss(y_hat, y)
        self.log("test_loss", test_loss)
```

A.7. PyGT GPU Model Training

```
from torch_geometric_temporal.signal import temporal_signal_split
from pytorch_lightning.callbacks.early_stopping import EarlyStopping

# Data Split into Training, Validation and Test Data
train_loader, testing =temporal_signal_split(dataset,
                                            train_ratio=0.6)

val_loader, test_loader =temporal_signal_split(testing,
                                            train_ratio=0.5)

model =DCRNNModel(node_features=5,
                      filters=40) # Choose Filters

early_stop_callback =EarlyStopping(monitor='val_loss',
                                min_delta=0.00,
                                patience=10,
                                verbose=False,
                                mode='max') # Sets Numbers of Epoch if no changes within Delta

trainer =pl.Trainer(callbacks=[early_stop_callback], min_epochs=100) # Set number of Epochs,
                                               Minimum Epochs will circumvent thrown Early
```

```
                                                       Stop Flag

trainer.fit(model, train_loader, val_loader)
```

List of Figures

List of Tables

Bibliography

[1] B. für Wirtschafts und Klimaschutz. *Unsere Energiewende: sicher, sauber, bezahlbar*. `https://www.bmwk.de/Redaktion/DE/Dossier/energiewende.html`. (Accessed on 04/20/2023).

[2] P. Cramton. "Electricity market design". In: *Oxford Review of Economic Policy* 33 (2017), pp. 589–612. URL: `https://academic.oup.com/oxrep/article/33/4/589/4587939?login=true`.

[3] F. Hasan, A. Kargarian, and A. Mohammadi. "A survey on applications of machine learning for optimal power flow". In: *2020 IEEE Texas Power and Energy Conference (TPEC)*. IEEE, pp. 1–6. ISBN: 1728144361.

[4] Y. Yang and L. Wu. "Machine learning approaches to the unit commitment problem: Current trends, emerging challenges, and new strategies". In: *The Electricity Journal* 34.1 (2021), p. 106889. ISSN: 1040-6190.

[5] S. Chatzivasileiadis. "Optimal Power Flow (DC-OPF and AC-OPF)". In: *DTU Summer School* (2018).

[6] M. Ş. Ahunbay, M. Bichler, and J. Knörr. "Pricing Optimal Outcomes in Coupled and Non-Convex Markets: Theory and Applications to Electricity Markets". In: *arXiv preprint arXiv:2209.07386* (2022).

[7] D. Deka and S. Misra. "Learning for DC-OPF: Classifying active sets using neural nets". In: *2019 IEEE Milan PowerTech*. IEEE, pp. 1–6. ISBN: 1538647222.

[8] G. Rebala, A. Ravi, and S. Churiwala. *An introduction to machine learning*. Springer, 2019.

[9] W. Chen, S. Park, M. Tanneau, and P. Van Hentenryck. "Learning optimization proxies for large-scale security-constrained economic dispatch". In: *Electric Power Systems Research* 213 (2022), p. 108566. ISSN: 0378-7796.

[10] W. Liao, B. Bak-Jensen, J. R. Pillai, Y. Wang, and Y. Wang. "A review of graph neural networks and their applications in power systems". In: *Journal of Modern Power Systems and Clean Energy* (2021). ISSN: 2196-5625.

[11] D. Owerko, F. Gama, and A. Ribeiro. "Optimal power flow using graph neural networks". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 5930–5934.

[12] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein. "Temporal graph networks for deep learning on dynamic graphs". In: *arXiv preprint arXiv:2006.10637* (2020).

[13] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar. "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models". In: *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 2021, pp. 4564–4573.

[14] Y. Li, R. Yu, C. Shahabi, and Y. Liu. "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting". In: *arXiv preprint arXiv:1707.01926* (2017).

[15] N. Angermeier. *Repository: Clearing Electricity Market of Variable Size with Graph Neural Networks*. May 2023. URL: `https://github.com/NadineAngermeier/ElectricityMCVariableGNN`.

[16] E. Rossi and M. Bronstein. *Deep learning on dynamic graphs*. `https://blog.twitter.com/engineering/en_us/topics/insights/2021/temporal-graph-networks`. (Accessed on 04/19/2023). Jan. 2021.

[17] M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer programming*. Vol. 271. Springer, 2014. ISBN: 331911008X.

[18] R. Lazimy. "Mixed-integer quadratic programming". In: *Mathematical Programming* 22.1 (1982), pp. 332–349. ISSN: 1436-4646.

[19] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, and P. Wang. "Solving mixed integer programs using neural networks". In: *arXiv preprint arXiv:2012.13349* (2020).

[20] D. Divényi, B. Polgári, Á. Sleisz, P. Sőrés, and D. Raisz. "Algorithm design for European electricity market clearing with joint allocation of energy and control reserves". In: *International Journal of Electrical Power Energy Systems* 111 (2019), pp. 269–285. ISSN: 0142-0615.

[21] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discrete Optimization* 19 (2016), pp. 79–102. ISSN: 1572-5286.

[22] L. Huang, X. Chen, W. Huo, J. Wang, F. Zhang, B. Bai, and L. Shi. "Branch and bound in mixed integer linear programming problems: A survey of techniques and trends". In: *arXiv preprint arXiv:2111.06257* (2021).

[23] R. Gomory. *An algorithm for the mixed integer problem*. Report. RAND CORP SANTA MONICA CA, 1960.

[24] E. Klotz and A. M. Newman. "Practical guidelines for solving difficult mixed integer linear programs". In: *Surveys in Operations Research and Management Science* 18.1-2 (2013), pp. 18–32. ISSN: 1876-7354.

[25] J. T. Linderoth and T. K. Ralphs. "Noncommercial software for mixed-integer linear programming". In: *Integer Programming*. CRC Press, 2005, pp. 269–320. ISBN: 0429125364.

[26] J. Jablonský. "Benchmarks for current linear and mixed integer optimization solvers". In: *Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis* 63.6 (2015), pp. 1923–1928. ISSN: 1211-8516.

[27] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, and K. Halbig. "The SCIP optimization suite 7.0". In: (2020). ISSN: 1438-0064.

[28] T. Berthold. "Rens". In: *Mathematical Programming Computation* 6.1 (2014), pp. 33–54. ISSN: 1867-2957.

[29] H. He, H. Daume III, and J. M. Eisner. "Learning to search in branch and bound algorithms". In: *Advances in neural information processing systems* 27 (2014).

[30] L. A. Wolsey. *Integer programming*. John Wiley Sons, 2020. ISBN: 1119606527. URL: `https://books.google.de/books?hl=de&lr=&id=knH8DwAAQBAJ&oi=fnd&pg=PP1&dq=integer+programming+wolsey&ots=wkCtcwOEi7&sig=Wh6f1PeGs6qPz0lcPSSPddGVt74#v=onepage&q=integer%20programming%20wolsey&f=false`.

[31] T. Berthold. "A computational study of primal heuristics inside an MI (NL) P solver". In: *Journal of Global Optimization* 70.1 (2018), pp. 189–206. ISSN: 1573-2916.

[32] T. Berthold. "Primal heuristics for mixed integer programs". Thesis. 2006.

[33] A. G. u. J. Krüger. *MIPLIB*. Web Page. 2022. URL: `https://miplib.zib.de/index.html`.

[34] H. Mittelmann. *Benchmarks for Optimization Software*. Web Page. 2022. URL: `http://plato.asu.edu/bench.html`.

[35] T. Koch, T. Berthold, J. Pedersen, and C. Vanaret. "Progress in mathematical programming solvers from 2001 to 2020". In: *EURO Journal on Computational Optimization* (2022), p. 100031. ISSN: 2192-4406.

[36] A. Marcos Alvarez, L. Wehenkel, and Q. Louveaux. "Online learning for strong branching approximation in branch-and-bound". In: (2016).

[37] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel. "A supervised machine learning approach to variable branching in branch-and-bound". In: (2014).

[38] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. "Learning to branch in mixed integer programming". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. ISBN: 2374-3468.

[39] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. "Exact combinatorial optimization with graph convolutional neural networks". In: *Advances in Neural Information Processing Systems* 32 (2019).

[40] H. Sun, W. Chen, H. Li, and L. Song. "Improving learning to branch via reinforcement learning". In: (2020).

[41] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky. "Dash: Dynamic approach for switching heuristics". In: *European Journal of Operational Research* 248.3 (2016), pp. 943–953. ISSN: 0377-2217.

[42] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. "Learning to branch". In: *International conference on machine learning*. PMLR, pp. 344–353. ISBN: 2640-3498.

[43] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio. “Parameterizing branch-and-bound search trees to learn branching policies”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35, pp. 3931–3939. ISBN: 2374-3468.

[44] K. Yilmaz and N. Yorke-Smith. “A study of learning search approximation in mixed integer branch and bound: Node selection in scip”. In: *Ai* 2.2 (2021), pp. 150–178. ISSN: 2673-2688.

[45] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. “Learning to Run Heuristics in Tree Search”. In: *Ijcai*, pp. 659–666.

[46] G. Hendel. “Adaptive large neighborhood search for mixed integer programming”. In: *Mathematical Programming Computation* 14.2 (2022), pp. 185–221. ISSN: 1867-2957.

[47] J. Song, Y. Yue, and B. Dilkina. “A general large neighborhood search framework for solving integer linear programs”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 20012–20023.

[48] J.-Y. Ding, C. Zhang, L. Shen, S. Li, B. Wang, Y. Xu, and L. Song. “Accelerating primal solution findings for mixed integer programs based on solution prediction”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34, pp. 1452–1459. ISBN: 2374-3468.

[49] Y. Tang, S. Agrawal, and Y. Faenza. “Reinforcement learning for integer programming: Learning to cut”. In: *International conference on machine learning*. PMLR, pp. 9367–9376. ISBN: 2640-3498.

[50] Z. Huang, K. Wang, F. Liu, H.-L. Zhen, W. Zhang, M. Yuan, J. Hao, Y. Yu, and J. Wang. “Learning to select cuts for efficient mixed-integer programming”. In: *Pattern Recognition* 123 (2022), p. 108353. ISSN: 0031-3203.

[51] Y. Ng, S. Misra, L. A. Roald, and S. Backhaus. “Statistical learning for DC optimal power flow”. In: *2018 Power Systems Computation Conference (PSCC)*. IEEE, pp. 1–7. ISBN: 1910963100.

[52] M. A. Laughton and M. G. Say. *Electrical engineer’s reference book*. Elsevier, 2013.

[53] NEMO Committee. “Euphemia Public Description”. In: (2020).

[54] I. PES Task Force on Benchmarks for Validation of Emerging Power System Algorithms. *PGLib-OPF: Benchmarks for the Optimal Power Flow Problem (GitHub - power-grid-lib)*. `https://github.com/power-grid-lib/pglib-opf`. (Accessed on 01/30/2023).

[55] U. Illinois Center for a Smarter Electric. *Power Cases*. Web Page. 2021. URL: `https://icseg.iti.illinois.edu/power-cases/`.

[56] U. W. Electrical Engineering. *Power Systems Test Case Archive*. `https://labs.ece.uw.edu/pstca/rts/pg_tcarts.htm`. (Accessed on 01/30/2023).

[57] R. Canyasse, G. Dalal, and S. Mannor. “Supervised learning for optimal power flow as a real-time proxy”. In: *2017 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*. IEEE, pp. 1–5. ISBN: 1538628902.

[58] A. S. Zamzam and K. Baker. "Learning optimal solutions for extremely fast AC optimal power flow". In: *2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE, pp. 1–6. ISBN: 1728161274.

[59] X. Pan. "DeepOPF: deep neural networks for optimal power flow". In: *Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pp. 250–251.

[60] Y. Sun, X. Fan, Q. Huang, X. Li, R. Huang, T. Yin, and G. Lin. "Local feature sufficiency exploration for predicting security-constrained generation dispatch in multi-area power systems". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, pp. 1283–1289. ISBN: 153866805X.

[61] T. Navidi, S. Bhooshan, and A. Garg. "Predicting solutions to the optimal power flow problem". In: *Project Report, Stanford University* (2016).

[62] K. Baker. "Learning warm-start points for AC optimal power flow". In: *2019 IEEE 29th International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, pp. 1–6. ISBN: 1728108241.

[63] S. Misra, L. Roald, and Y. Ng. "Learning for constrained optimization: Identifying optimal active constraint sets". In: *INFORMS Journal on Computing* 34.1 (2022), pp. 463–480. ISSN: 1091-9856.

[64] A. J. Ardakani and F. Bouffard. "Prediction of umbrella constraints". In: *2018 Power Systems Computation Conference (PSCC)*. IEEE, pp. 1–7. ISBN: 1910963100.

[65] K. Baker and A. Bernstein. "Joint chance constraints reduction through learning in active distribution networks". In: *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, pp. 922–926. ISBN: 1728112958.

[66] K. Baker and A. Bernstein. "Joint chance constraints in AC optimal power flow: Improving bounds through learning". In: *IEEE Transactions on Smart Grid* 10.6 (2019), pp. 6376–6385. ISSN: 1949-3053.

[67] R. T. A. King, X. Tu, L.-A. Dessaint, and I. Kamwa. "Multi-contingency transient stability-constrained optimal power flow using multilayer feedforward neural networks". In: *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, pp. 1–6. ISBN: 1467387215.

[68] F. Thams, L. Halilbasic, P. Pinson, S. Chatzivasileiadis, and R. Eriksson. "Data-driven security-constrained opf". In: *Proc. 10th Bulk Power Syst. Dyn. Control Symp*, pp. 1–10.

[69] L. Halilbašić, F. Thams, A. Venzke, S. Chatzivasileiadis, and P. Pinson. "Data-driven security-constrained AC-OPF for operations and markets". In: *2018 power systems computation conference (PSCC)*. IEEE, pp. 1–7. ISBN: 1910963100.

[70] L. Conejo Antonio J.and Baringo. "Unit Commitment and Economic Dispatch". In: *Power System Operations*. Springer International Publishing, 2018, pp. 197–232. DOI: 10.1007/978-3-319-69407-8_7. URL: https://doi.org/10.1007/978-3-319-69407-8_7.

[71] G. Dalal, E. Gilboa, S. Mannor, and L. Wehenkel. "Unit commitment using nearest neighbor as a short-term proxy". In: *2018 Power Systems Computation Conference (PSCC)*. IEEE, pp. 1–7. ISBN: 1910963100.

[72] Á. S. Xavier, F. Qiu, and S. Ahmed. "Learning to solve large-scale security-constrained unit commitment problems". In: *INFORMS Journal on Computing* 33.2 (2021), pp. 739–756. ISSN: 1091-9856.

[73] Y. Yang, X. Lu, and L. Wu. "Integrated data-driven framework for fast SCUC calculation". In: *IET Generation, Transmission Distribution* 14.24 (2020), pp. 5728–5738. ISSN: 1751-8687.

[74] O. Sondermeijer, R. Dobbe, D. Arnold, C. Tomlin, and T. Keviczky. "Regression-based inverter control for decentralized optimal power flow and voltage regulation". In: *arXiv preprint arXiv:1902.08594* (2019).

[75] A. Garg, M. Jalali, V. Kekatos, and N. Gatsis. "Kernel-based learning for smart inverter control". In: *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, pp. 875–879. ISBN: 1728112958.

[76] F. Bellizio, S. Karagiannopoulos, P. Aristidou, and G. Hug. "Optimized local control for active distribution grids using machine learning techniques". In: *2018 IEEE Power Energy Society General Meeting (PESGM)*. IEEE, pp. 1–5. ISBN: 1538677032.

[77] T. Summers, J. Warrington, M. Morari, and J. Lygeros. "Stochastic optimal power flow based on convex approximations of chance constraints". In: *2014 Power Systems Computation Conference*. IEEE, pp. 1–7. ISBN: 8393580137.

[78] Y. Guo, K. Baker, E. Dall'Anese, Z. Hu, and T. H. Summers. "Data-based distributionally robust stochastic optimal power flow—Part I: Methodologies". In: *IEEE Transactions on Power Systems* 34.2 (2018), pp. 1483–1492. ISSN: 0885-8950.

[79] Y. Guo, K. Baker, E. Dall'Anese, Z. Hu, and T. H. Summers. "Data-based distributionally robust stochastic optimal power flow—Part II: Case studies". In: *IEEE Transactions on Power Systems* 34.2 (2018), pp. 1493–1503. ISSN: 0885-8950.

[80] ENTSO-E. *Energy Identification Codes*. https://www.entsoe.eu/data/energy-identification-codes-eic/. (Accessed on 01/28/2023).

[81] I. Neutelings. *Neural networks – TikZ.net*. https://tikz.net/neural_networks/. (Accessed on 02/21/2023).

[82] P. Bichler. *Business Analytics: Neural Networks*. TUM Course at Chair of Decision Sciences Systems - Department of Informatics. Oct. 2021.

[83] P. Michiardi. *Deep Learning: Convolutional Neural Networks*. TUM Course at Chair of Decision Sciences Systems - Department of Informatics. Feb. 2022.

[84] *Drawing a convolution with Tikz - TeX - LaTeX Stack Exchange*. `https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz`. (Accessed on 02/24/2023). Jan. 2021.

[85] I. Shafkat. *Intuitively Understanding Convolutions for Deep Learning*. Towards Data Science. June 2018. URL: `https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1`.

[86] S. Günnemann. *Machine Learning for Graphs and Sequential Data: Graphs - Semi-Supervised Learning*. TUM Course at Chair of Decision Sciences Systems - Department of Informatics. June 2022.

[87] T. N. Kipf and M. Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[88] P. Michiardi. *Deep Learning: Convolutional Neural Networks*. TUM Course at Chair of Decision Sciences Systems - Department of Informatics. Feb. 2022.

[89] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[90] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[91] *Open energy system models*. Jan. 2023. URL: `https://en.wikipedia.org/wiki/Open_energy_system_models`.

[92] *Open energy system databases*. Jan. 2023. URL: `https://en.wikipedia.org/wiki/Open_energy_system_databases`.

[93] *Transmission network datasets - wiki.openmod-initiative.org*. `https://wiki.openmod-initiative.org/wiki/Transmission_network_datasets`. (Accessed on 01/28/2023).

[94] *Datasets - ENERGYDATA.INFO*. `https://energydata.info/dataset`. (Accessed on 01/28/2023).

[95] ENTSO-E. *Transparency Platform*. `https://transparency.entsoe.eu/`. (Accessed on 01/28/2023).

[96] ENTSO-E. *Power Statistics*. `https://www.entsoe.eu/data/power-stats/`. (Accessed on 01/28/2023).

[97] PJM. *Data Miner 2*. `https://dataminer2.pjm.com/list`. (Accessed on 01/28/2023). 2017.

[98] Midcontinent Independent System Operator. *MISO Market Data*. `https://www.misoenergy.org/markets-and-operations/real-time--market-data/market-reports/#t=10&p=0&s=MarketReportPublished&sd=desc`. (Accessed on 01/28/2023). 2023.

[99] *Global Energy Statistics Yearbook - Datasets - ENERGYDATA.INFO*. `https://energydata.info/dataset/key-world-energy-statistics-enerdata`. (Accessed on 01/28/2023).

[100] *Open Power System Data – A platform for open data of the European power system.* https://open-power-system-data.org/. (Accessed on 01/28/2023).

[101] *OEP.* https://openenergy-platform.org/. (Accessed on 01/28/2023).

[102] *Grid Optimization Competition.* https://gocompetition.energy.gov/. (Accessed on 01/30/2023).

[103] C. Arderne, C. Zorn, C. Nicolas, and E. E. Koks. "Predictive mapping of the global power system using open data". In: *Scientific Data* 7.1 (2020), p. 19. ISSN: 2052-4463. DOI: 10.1038/s41597-019-0347-4. URL: https://doi.org/10.1038/s41597-019-0347-4.

[104] Q. Zhou and J. W. Bialek. "Approximate model of European interconnected system as a benchmark system to study effects of cross-border trades". In: *IEEE Transactions on power systems* 20.2 (2005), pp. 782–788.

[105] *Updated and Validated Power Flow Model of the Main Continental European Transmission Network » Knowledge Base » PowerWorld.* https://www.powerworld.com/knowledge-base/updated-and-validated-power-flow-model-of-the-main-continental-european-transmission-network. (Accessed on 01/28/2023).

[106] *SciGRID General information.* https://www.power.scigrid.de/. (Accessed on 01/28/2023).

[107] *DIW Berlin: Modelle.* https://www.diw.de/de/diw_01.c.599753.de/modelle.htm#ab_textabschnitt_599760. (Accessed on 01/28/2023).

[108] *GitHub - bdw/GridKit: GridKit is an power grid extraction toolkit.* https://github.com/bdw/GridKit. (Accessed on 01/29/2023).

[109] J. Hörsch, F. Hofmann, D. Schlachtberger, and T. Brown. "PyPSA-Eur: An open optimisation model of the European transmission system". In: *Energy strategy reviews* 22 (2018), pp. 207–215.

[110] *GitHub - PyPSA/PyPSA: PyPSA: Python for Power System Analysis.* https://github.com/PyPSA/PyPSA. (Accessed on 01/28/2023).

[111] *GitHub - openego/ding0: DIstribution Network Generat0r.* https://github.com/openego/ding0. (Accessed on 01/29/2023).

[112] *MATPOWER – Free, open-source tools for electric power system simulation and optimization.* https://matpower.org/. (Accessed on 01/29/2023).

[113] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas. "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education". In: *IEEE Transactions on power systems* 26.1 (2010), pp. 12–19. ISSN: 0885-8950.

[114] *Data Miner 2 - Hourly Load: Metered.* https://dataminer2.pjm.com/feed/hrl_load_metered. (Accessed on 01/30/2023).

[115] *Data Miner 2 - Energy Market Generation Offers.* https://dataminer2.pjm.com/feed/energy_market_offers. (Accessed on 01/30/2023).

[116] *IEEE 30-Bus System.* https://electricgrids.engr.tamu.edu/electric-grid-test-cases/ieee-30-bus-system/. (Accessed on 01/30/2023).

[117] O. Vogel. “Pricing in Non-Convex Electricity Markets: An Empirical Benchmarking of Different Pricing Algorithms”. Thesis. 2021.

[118] J. L. Gross and T. W. Tucker. *Topological graph theory*. Courier Corporation, 2001.

[119] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: `https://www.gurobi.com`.

[120] *gurobipy, the Gurobi Python Interface - Gurobi Optimization*. `https://www.gurobi.com/documentation/10.0/quickstart_windows/cs_grbpy_the_gurobi_python.html`. (Accessed on 02/07/2023).

[121] Z. Wu, S. Pan, G. Long, J. Jiang, X. Chang, and C. Zhang. “Connecting the dots: Multivariate time series forecasting with graph neural networks”. In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery data mining*, pp. 753–763.

[122] Y. Verma. *A beginner’s guide to Spatio-Temporal graph neural networks*. Analytics India Magazine (AIM). Jan. 2022.

[123] W. Liao, Y. Yu, Y. Wang, and J. Chen. “Reactive power optimization of distribution network based on graph convolutional network”. In: *Dianwang Jishu/Power System Technology* 45.6 (2021), pp. 2150–2160.

[124] X. Li, Z. Guo, X. Dai, Y. Lin, J. Jin, F. Zhu, and F.-Y. Wang. “Deep imitation learning for traffic signal control and operations based on graph convolutional neural networks”. In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2020, pp. 1–6.

[125] F. Gama, Q. Li, E. Tolstaya, A. Prorok, and A. Ribeiro. “Synthesizing decentralized controllers with graph neural networks and imitation learning”. In: *IEEE Transactions on Signal Processing* 70 (2022), pp. 1932–1946.

[126] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, and G. Chanan. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[127] M. Fey and J. E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: 2019. URL: `https://github.com/pyg-team/pytorch_geometric`.

[128] NVIDIA, P. Vingelmann, and F. H. Fitzek. *CUDA*. 2020. URL: `https://developer.nvidia.com/cuda-toolkit`.

[129] W. Falcon et al. “PyTorch Lightning”. In: *GitHub.https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[130] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[131] D. Chicco, M. J. Warrens, and G. Jurman. "The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation". In: *PeerJ Computer Science* 7 (2021), e623.

[132] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[133] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[134] J. Brownlee. "What is the Difference Between a Batch and an Epoch in a Neural Network". In: *Machine Learning Mastery* 20 (2018).

[135] Padma M. *End-to-End Introduction to Evaluating Regression Models*. `https://www.analyticsvidhya.com/blog/2021/10/evaluation-metric-for-regression-models/`. (Accessed on 15/24/2023). Sept. 2022.