Power Failure Cascade Prediction using Machine Learning

by

Sathwik P. Chadaga

Dual Degree (B.Tech. and M.Tech.), Electrical Engineering,
Indian Institute of Technology Madras, 2020

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Masters of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

©2023 Sathwik P. Chadaga. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Sathwik P. Chadaga
Department of Aeronautics and Astronautics
August 8, 2023

Certified by: Eytan H. Modiano
R.C. Maclaurin Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: Jonathan P. How
R. C. Maclaurin Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Power Failure Cascade Prediction using Machine Learning

by

Sathwik P. Chadaga

Submitted to the Department of Aeronautics and Astronautics
on August 8, 2023, in partial fulfillment of the
requirements for the degree of
Masters of Science in Aeronautics and Astronautics

Abstract

We consider the problem of predicting power failure cascades due to branch failures. We propose several flow-free models using machine learning techniques like support vector machines, naive Bayes classifiers, and logistic regression. These models predict the grid states at every generation of a cascade process given the initial contingency. Further, we also propose a model based on graph neural networks (GNNs) that predicts cascades from the initial contingency and power injection values. We train the proposed models using a cascade sequence data pool generated from simulations. We then evaluate our models at various levels of granularity. We present several error metrics that gauge the models' ability to predict the failure size, the final grid state, and the failure time steps of each branch within the cascade. We benchmark the proposed models against the influence model proposed in the literature. We show that the proposed machine learning models outperform the influence models under every metric. We also show that the graph neural network model, in addition to being generic over randomly scaled power injection values, outperforms multiple influence models that are built specifically for their corresponding loading profiles. Finally, we show that the proposed models reduce the computational time by almost two orders of magnitude.

Thesis Supervisor: Eytan H. Modiano
Title: R.C. Maclaurin Professor of Aeronautics and Astronautics

Acknowledgments

I want to thank my advisor Prof. Eytan Modiano for his guidance and constant support. His insights and intuitions have been crucial in the formulation and development of this work. I also want to thank my colleagues Xinyu Wu and Dr. Dan Wu. Their works on the influence model and their implementation of the cascading failure simulator, with which I generate the data required to train my models, have been vital to my thesis. I am also grateful to my undergraduate advisors Prof. David Koilpillai and Prof. Nambi Seshadri who inspired me to be a researcher.

This work was supported by NSF grants CNS-1735463 and CNS-2106268, and by a research award from the C3.ai Digital Transformation Institute.

I would like to thank my colleagues at the Communications and Networking Research Group - Bai Liu, Vishrant Tripathi, Chirag Rao, Nick Jones, Jerrod Wigmore, Quang Nguyen, Vallabh Ramakanth, and Xinzhe Fu - for creating such a supportive and friendly atmosphere in the lab. I also want to thank my friends Akshay Subramanian and Avik Pal for the useful discussions on graph neural networks that initially encouraged me to explore its application in the field of power systems.

I am grateful to my friends - Naman Aggarwal, Keshav Gupta, Mansi Joisher, Mrigi Munjal, Siddharth Nayak, Richa Nayak, and Saachi Chandrashekhar - for making my life at MIT fun and enjoyable. I am lucky to have found you all. I thank the groups that I have been a part of during my graduate life like the MIT Cricket Club, the Tang Hall Residents Association, and the LIDS Socials Committee for the fun and useful activities they have offered me.

I am grateful to my parents, Ahalya H. S. and Udaya Prakash Chadaga, for their utmost love and support. To this day, I find myself using the skills that I learned from my talented mother when I was a child, be it related to basic algebra, music or life in general. My father's hard work and dedication has been and will always be a constant source of inspiration to me. I thank them for believing in me and encouraging me to pursue my dreams.

Contents

1 Introduction **15**

1.1 Motivation . . . 15

1.2 Related Work . . . 15

1.2.1 Flow-Based Methods . . . 15

1.2.2 Flow-Free Methods . . . 16

1.2.3 Machine Learning Techniques . . . 16

1.2.4 Graph Neural Network Methods . . . 17

1.3 Problem Formulation . . . 18

1.4 Contributions . . . 19

1.5 Outline . . . 20

2 Machine Learning Techniques for Failure Cascade Prediction **21**

2.1 Naive Bayes Model . . . 21

2.1.1 Prerequisite . . . 21

2.1.2 Naive Bayes for Cascade Prediction . . . 22

2.2 Support Vector Machine Model . . . 23

2.2.1 Prerequisite . . . 23

2.2.2 Support Vector Machines for Cascade Prediction . . . 24

2.3 Logistic Regression Model . . . 25

2.3.1 Prerequisite . . . 25

2.3.2 Regression Model for Cascade Prediction . . . 25

3 The Graph Neural Network Model 27
3.1 Model Definition . . . 27
3.1.1 Initial Stage . . . 27
3.1.2 Attention Stage . . . 29
3.1.3 Averaging Stage (Hidden Layers) . . . 30
3.1.4 Final Stage . . . 31
3.2 Model Training . . . 32
3.2.1 Forward Pass . . . 32
3.2.2 Back Propagation . . . 33
4 Results: Data Synthesis and Design Specifications 37
4.1 Data Synthesis . . . 37
4.1.1 The Cascading Failure Simulator Oracle . . . 37
4.1.2 Data for Machine Learning Models . . . 38
4.1.3 Data for the GNN Model . . . 39
4.2 Model Design Specifications . . . 40
4.2.1 GNN Model . . . 40
4.2.2 Machine Learning Models . . . 40
4.2.3 Influence Model . . . 41
5 Results: Graph Level Performance 43
5.1 Performance Metrics Definitions . . . 43
5.1.1 Failure Size Error Rate . . . 43
5.1.2 Final State Error Rate . . . 44
5.1.3 Failure Step Error Rate . . . 44
5.2 Results for the Machine Learning Models . . . 44
5.3 Results for the GNN Models . . . 47
6 Results: Branch Level Performance 51
6.1 Performance Metrics Definitions . . . 51
6.1.1 Branch Failure Frequency . . . 51

6.1.2 Branch Final State Error Rate 52
6.1.3 Branch Failure Step Error Rate 52
6.2 Results for the Machine Learning Models 53
6.3 Results for the GNN Models 55

7 Results: Runtime Analysis 61

8 Conclusion 63
8.1 Future Directions 63

List of Figures

3-1 Block diagram of the GNN model. 28

5-1 Failure size error rates l^{α}_{size} of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α. 45

5-2 Final state error rates l^{α}_{state} of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α. 46

5-3 Failure step error rates $l^{\alpha}_{failure\text{-}step}$ of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α. 46

5-4 Failure size error rates l^{α}_{size} of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α. 48

5-5 Final state error rates l^{α}_{state} of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α. 49

5-6 Failure step error rates $l^{\alpha}_{failure\text{-}step}$ of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α. 49

6-1 Branch-average error in prediction of final state $l_{state,e}$ for IEEE89, IEEE118, and IEEE1354 (left to right) for various load scaling plotted against $l_{freq,e}$. 53

6-2 Branch-average error in prediction of failure steps $l_{failure-step,e}$ for IEEE89, IEEE118, and IEEE1354 (left to right) for various load scaling plotted against $l_{freq,e}$. 55

6-3 Error in prediction of final state $l_{state,e}$ for IEEE89 (left) and IEEE118 (right) averaged over all scaling values $[1, 2]$ against branch failure frequencies $l_{freq,e}$. 56

6-4 Branch-average error in prediction of final state $l_{state,e}$ for IEEE89 (left) and IEEE118 (right) for various load scaling against branch failure frequencies $l_{freq,e}$. 57

6-5 Error in prediction of failure time steps $l_{failure-step,e}$ for IEEE89 (left) and IEEE118 (right) averaged over all scaling $[1, 2]$ against failure frequencies $l_{freq,e}$. 58

6-6 Branch-average error in prediction of failure steps $l_{failure-step,e}$ for IEEE89 (left) and IEEE118 (right) for various scaling against branch failure frequencies $l_{freq,e}$. 59

List of Tables

7.1 Prediction time in seconds per 1000 samples 61

Chapter 1

Introduction

1.1 Motivation

Modern power grids often experience unpredictable component failures that are caused due to an exogenous event like a tree branch falling, bad weather, failure of an aged device, or an operator error. These random failures, if not treated properly, can propagate rapidly through the grid, potentially resulting in large scale blackouts. Hence, it is important to study such *failure cascades* as part of the power contingency analysis. Further, power grids have seen a recent surge in outages [1] due to extreme weather conditions [2, 3] and power grid aging [4], causing significant losses to businesses, industries, and healthcare sectors [5, 6]. Additionally, the move towards electrification of fossil fuel technologies [7] makes the modeling and prediction of power failures increasingly important. However, failure cascade prediction is a difficult task due to the complex and time varying nature of interactions between various grid components.

1.2 Related Work

1.2.1 Flow-Based Methods

There have been several studies performed on historical failure cascade data [8–11]. However, the scarce historical records of cascading failures are not representative of

all the possibilities, leaving potential blackouts concealed. To tackle this challenge, numerical simulations and analysis of different initial outages have been studied based on power flow models. This involves solving the static power flow problem step-by-step and determining the sequence of quasi-static transmission link overflows [12, 13]. However, the AC power flow model is computationally expensive, while the computationally tractable DC power flow model has been shown to underestimate the failure sizes [14].

1.2.2 Flow-Free Methods

To overcome the high complexity of flow-based methods, efforts have been devoted to constructing flow-free models of failure cascades. The branching process is a popular tool that measures the distribution of the number of outages in a cascade [9, 10], and the random chemistry algorithm together with such a distribution can efficiently estimate the overall blackout risk [15, 16]. Moreover, a model to predict blackout occurrence has been proposed using chemical master equation in [17]. The expectation-maximization algorithm has been used to estimate interactions between branches during a cascade in [18]. The branch interactions have also been captured using the interaction [19] and influence [11, 20] models. The above flow-free models aspire to capture the cascade flow dynamics from data, obtained either from simulations or historic outage records. This approach of designing cascade models from data has led researchers to investigate fast and accurate machine learning models.

1.2.3 Machine Learning Techniques

Machine learning has been used in power system analysis in various settings [21–23]. For example, as power flow calculation using Newton-Raphson is computationally expensive, more efficient power flow calculation methods have been proposed using deep [24] and convolutional [25] neural networks. Moreover, in the area of cascade prediction, support vector machines have been employed in blackout prediction [26], cascade failure size estimation [27], and load loss estimation [28]. A performance

comparison of various machine learning tools like support vector machines, k-nearest neighbors, logistic regression, and decision trees for cascade size estimation is done in [27]. Additionally, methods using Bayes networks have been proposed for failure cascade prediction in [29]. Despite being computationally efficient, these techniques fail to take advantage of the power grid topology information leading us to explore techniques that use graph neural networks.

1.2.4 Graph Neural Network Methods

Graph neural networks (GNNs) are a type of neural networks that operate on graph-structured data [30, 31]. They process input graphs by repetitively updating the information at each node based on its neighbors, thereby leveraging the underlying graph topology. There have been recent applications of GNNs in the field of power networks. One such application is the design of computationally efficient power flow solvers. In [32–34], GNNs are trained in a supervised way to imitate the Newton-Raphson power flow solver. Whereas [35,36] follow an unsupervised learning method that minimizes the violation of Kirchoff's laws. GNNs have also been used to design fast optimal power flow solvers [37, 38].

Moreover, GNNs have seen recent application in the field of power failure cascades. GNN based methods have been proposed for predicting grid safety from the grid operation conditions [39, 40]. Additionally, GNNs have been used for real time grid monitoring tasks during a cascade, like predictions of optimal load shedding [41], total load lost [42], and fraction of tripped branches [43]. All these works involve a graph-level prediction task, i.e. they predict a particular property of the grid as a whole. GNNs can also be used for edge-level and node-level prediction tasks. For example, in [44], a node-level vulnerability metric called the Avalanche Centrality is predicted for all nodes of the grid using GNNs.

The existing works as discussed above are focused on characterizing one or two aspects of failure cascades, like load loss, failure size, or blackout possibility, lacking a comprehensive evaluation of the cascade at finer levels of granularity. This is done in [20], where an influence model is trained to predict the power grid states at every

generation of the cascade. However, the influence model approach cannot generalize for variable loading as it only takes the initial contingency as its input and not the power injection or the power flow values. A flow-based GNN model has been proposed in [45] that can generalize for variable power injections. However, this model is centered around predicting the power flow values in a step-by-step manner to obtain the sequence of branch overflows. Hence, even though this technique speeds up the cascade prediction process compared to traditional methods, it still involves a high computational overhead in handling the formation of islands during the cascade, such as identification of islands and rebalancing the load and power generation within islands.

1.3 Problem Formulation

We consider the power failure cascade process due to branch failures. In this setting, a failure cascade begins with an initial failure of one or more branches in the grid. The initial branch failures perturb the power flow in the grid, leading other branches to overload and trip. The new failures further cause additional branches to trip and so on, consequently triggering a cascade process. The cascade process can be grouped into generations in time [10], which we refer to as time steps.

We represent the power grid by a directed graph $G = (V, E)$, where the nodes V represent buses and the directed edges E represent branches. For a branch $e \in E$ at time t, we choose the *branch state* $s_e[t]$ to be its binary operational state, which can either be 0 (failed) or 1 (active). We define the *network state* at time t as $s[t] := (s_e[t])_{e \in E}$.

Our goal is to predict the *cascade sequence* $s := (s[t])_{t=0}^{T-1}$ for a given *initial contingency* $s[0]$ (the network state at $t = 0$) where T is the cascade length. However, we assume that once a branch fails, it stays in the failed state for the rest of the cascade. This allows us to define the *failure step* of a branch e, the time step at which its state changes from 1 to 0, as $f_e := \sum_{t=0}^{T-1} s_e[t]$. From this failure step f_e, we can fully recover the branch states $s_e[t]$, and hence s, by setting $s_e[t] = 1$ for $0 \leq t < f_e$ and

$s_e[t] = 0$ for $f_e \leq t < T$ for all $e \in E$. Hence, instead of predicting s directly, we design a model that predicts the branch failure steps $f := (f_e)_{e \in E}$.

1.4 Contributions

In this thesis, we build several flow-free models for cascade sequence prediction without requiring power flow calculation at every generation of the cascade. We summarize our contributions below.

1. We propose several flow-free models using machine learning techniques like the support vector machines, naive Bayes classifiers, and logistic regression. Given the initial contingency, these models predict the grid state at every generation of a cascade, providing a way to comprehensively evaluate cascades at various levels of granularity.

2. We propose a flow-free model based on GNN that predicts grid states at every generation of a cascade. This model takes as input the node power injection values, the initial contingency, and the grid topology. Hence, the GNN model can be generalized over variable load injection profiles.

3. We use the cascading failure simulator oracle from [15] to generate a cascade sequence dataset to train our models. We then evaluate the performance of our models at various levels of granularity including prediction of the failure size, the final grid state, and the failure steps within a cascade.

4. We benchmark our models against the influence model and show that the machine learning models have much lower error rates. Further, we show that the GNN model, in addition to being generic over randomly scaled loading values, outperforms different load-specific influence models under every metric.

5. We perform a runtime analysis and show that the proposed models reduce the prediction time by almost two orders of magnitude compared to the DC power flow calculation based simulators.

1.5 Outline

The rest of the thesis is organized as follows. We propose the machine learning models in Chapter 2 and propose the GNN model in Chapter 3. We discuss the cascading failure simulator oracle with which we generate the data required to train and evaluate our models in Chapter 4. We also discuss several implementation details in Chapter 4. We present graph-level prediction performance of the models in Chapter 5, branch-level prediction performance of the models in Chapter 6, and runtime performance of the models in Chapter 7. We end with some concluding remarks in Chapter 8.

Chapter 2

Machine Learning Techniques for Failure Cascade Prediction

In this chapter, we propose several models based on machine learning techniques to predict cascade sequences from the given initial contingency. We explore three classifiers from the machine learning domain: the naive Bayes classifier, the support vector machine classifier, and the logistic regression classifier. The proposed models take as input the initial contingency $s[0]$ and predict the cascade failure steps f_e of all branches $e \in E$. We formally define the proposed models in the following sections.

2.1 Naive Bayes Model

2.1.1 Prerequisite

We first explain the naive Bayes classifier [46] before we explain how we adapt it to solve our problem. A naive Bayes classifier `naiveBayes` takes a vector $\vec{x} := [x_1, x_2, ..., x_n]$ as input and predicts a label $\widehat{y} := \texttt{naiveBayes}(\vec{x}) \in \{0, ..., K-1\}$. For our problem, we explore a suitable variant called the Bernoulli naive Bayes classifier, in which each of the inputs is binary $\forall j, x_j \in \{0, 1\}$. The mechanics of such a classifier is explained below.

On a high level, the naive Bayes classifier tries to find a label y that maximizes

the conditional probability $\mathbb{P}(y \mid \vec{x})$. We modify this conditional probability using the Bayes theorem as follows.

$$\mathbb{P}(y \mid \vec{x}) = \frac{\mathbb{P}(y)\mathbb{P}(\vec{x} \mid y)}{\mathbb{P}(\vec{x})} = \frac{\mathbb{P}(y)\mathbb{P}(x_1, ..., x_n \mid y)}{\mathbb{P}(\vec{x})}. \tag{2.1}$$

Further, the classifier assumes conditional independence among the input features (hence, the name naive Bayes) allowing the numerator to be decomposed as follows.

$$\mathbb{P}(y \mid \vec{x}) = \frac{\mathbb{P}(y) \prod_j \mathbb{P}(x_j \mid y)}{\mathbb{P}(\vec{x})}. \tag{2.2}$$

Hence, for a given input, the naive Bayes classifier predicts the output label by picking the y that maximizes the above probability.

$$\widehat{y} = \texttt{naiveBayes}(\vec{x}) := \arg \max_{y \in \{0,...,K-1\}} \mathbb{P}(y) \prod_j \mathbb{P}(x_j \mid y) \tag{2.3}$$

The probabilities $\mathbb{P}(y)$ can be estimated as the fraction of samples in training dataset whose labels are y. And since $x_j \in \{0, 1\}$, the probabilities $\mathbb{P}(x_j \mid y)$ can be estimated as follows.

$$\mathbb{P}(x_j \mid y) = \mathbb{P}(x_j = 1 \mid y)x_j + \mathbb{P}(x_j = 0 \mid y)(1 - x_j) \tag{2.4}$$

where, the probabilities $\mathbb{P}(x_j = 1 \mid y)$ and $\mathbb{P}(x_j = 0 \mid y)$ can be estimated from training dataset again by counting the frequencies of data samples.

2.1.2 Naive Bayes for Cascade Prediction

In our cascade prediction problem, our goal is to predict the failure time step f_e for all branches $e \in E$ given the initial contingency $s[0]$. For this purpose, we use a set of naive Bayes classifiers $\{\texttt{naiveBayes}_e,\ e \in E\}$, where each naive Bayes classifier $\texttt{naiveBayes}_e$ is associated with the prediction task of a particular branch $e \in E$. Analogous to the previous subsection, for each classifier $\texttt{naiveBayes}_e$, we treat the initial contingency $s[0]$ as the classifier's input vector $\vec{x}$ and treat the classifier's output

$\widehat{y}$ as the failure step prediction $\widehat{f_e}$ with possible labels $\{0, ..., T-1\}$. Hence, our proposed naive Bayes model is composed of $|E|$ naive Bayes classifiers that predict the failure steps of all branches as follows.

$$\forall e \in E, \quad \widehat{f_e} = \texttt{naiveBayes}_e(s[0]) \tag{2.5}$$

We train each of these $|E|$ classifiers separately using training data as explained in the previous subsection by treating $s[0]$ as the input vectors $\vec{x}$ and treating the true failure steps f_e as the target labels y.

2.2 Support Vector Machine Model

2.2.1 Prerequisite

We first explain the support vector machine (SVM) classifier [47] before we explain how we adapt it to solve our problem. Similar to naive Bayes, an SVM classifier `svm` takes a vector $\vec{x} \in \mathbb{R}^n$ as input and predicts a label $\widehat{y} := \texttt{svm}(\vec{x}) \in \{0, ..., K-1\}$.

The SVM classifier contains trainable parameters $w_k \in \mathbb{R}^n$ and $b_k \in \mathbb{R}$ corresponding to each label $k \in \{0, ..., K-1\}$. It predicts the label $\widehat{y}$ as

$$\widehat{y} := \texttt{svm}(\vec{x}) = \arg \max_{k \in \{0,...,K-1\}} w_k^T \vec{x} + b_k. \tag{2.6}$$

Note that we are using the linear variant of SVMs. Now, given the training data with input-output pairs $(\vec{x}_i, y_i)$, $i = 1, ..., m$, the training phase involves learning the parameters w_k and b_k by solving the following optimization problem for each class $k \in \{0, ..., K-1\}$ separately. This type of classification in which we are predicting whether the output is a particular label or not for each label separately (note the

term $\mathbb{I}(y_i = k)$) is called one-versus-rest classification.

$$\min_{w_k, b_k, \gamma} \frac{1}{2} w_k^T w_k + C \sum_{i=1}^{n} \gamma_{k,i} \tag{2.7}$$

$$\text{subject to } (2\mathbb{I}(y_i = k) - 1)(w_k^T \vec{x}_i + b) \geq 1 - \gamma_{k,i} \text{ and } \gamma_{k,i} \geq 0 \text{ for all } i = 1, ..., n.$$

where, $\mathbb{I}(\cdot)$ represents the indicator function, $\gamma_{k,i}$ represents the distance of each sample $i = 1, ..., m$ away from the decision hyperplane that separates the label k from other labels, and C is a penalty term that controls the extent of regularization. We refer the reader to [47] for details on this optimization problem and its solution.

2.2.2 Support Vector Machines for Cascade Prediction

In our cascade prediction problem, our goal is to predict the failure time step f_e for all branches $e \in E$ given the initial contingency $s[0]$. For this purpose, we use a set of SVM classifiers $\{\texttt{svm}_e,\ e \in E\}$, where each classifier $\texttt{svm}_e$ is associated with the prediction task of a particular branch $e \in E$. Analogous to the previous subsection, for each classifier $\texttt{svm}_e$, we treat the initial contingency $s[0]$ as the classifier's input vector $\vec{x}$ and treat the classifier's output $\widehat{y}$ as the failure step prediction $\widehat{f_e}$ with possible labels $\{0, ..., T-1\}$. Hence, our proposed SVM model is composed of $|E|$ SVM classifiers that predict the failure steps of all branches as follows.

$$\forall e \in E, \quad \widehat{f_e} = \texttt{svm}_e(s[0]) \tag{2.8}$$

We train each of these $|E|$ classifiers separately using training data as explained in the previous subsection by treating $s[0]$ as the input vectors $\vec{x}$ and treating the true failure steps f_e as the target labels y.

2.3 Logistic Regression Model

2.3.1 Prerequisite

We first explain the logistic regression classifier [48] before we explain how we adapt it to solve our problem. Similar to naive Bayes and SVM, a logistic regression classifier `regression` takes a vector $\vec{x} \in \mathbb{R}^n$ as input and predicts a label $\widehat{y} := \texttt{regression}(\vec{x}) \in \{0, ..., K-1\}$.

The logistic regression classifier contains trainable parameters $w_k \in \mathbb{R}^n$ and $b_k \in \mathbb{R}$, where each parameter corresponds to a label $k \in \{0, ..., K-1\}$. The classifier predicts the probability $\widehat{p}_k(\vec{x}) := \mathbb{P}(y = k|\vec{x})$ for each label $k \in \{0, ..., K-1\}$ as

$$\widehat{p}_k(\vec{x}) := \mathbb{P}(y = k|\vec{x}) = \frac{\exp(w_k^T \vec{x} + b_k)}{\sum_{l=0}^{K-1} \exp(w_l^T \vec{x} + b_l)}. \tag{2.9}$$

Then, it predicts the output label as

$$\widehat{y} := \texttt{regression}(\vec{x}) = \arg \max_{k \in \{0,...,K-1\}} \widehat{p}_k(\vec{x}) \tag{2.10}$$

Now, given the training data with input-output pairs $(\vec{x}_i, y_i)$, $i = 1, ..., m$, the training phase involves learning the parameters w_k and b_k by solving the following optimization problem.

$$\min_{w} \frac{1}{2} \sum_{k=0}^{K-1} w_k^T w_k - C \sum_{i=1}^{m} \sum_{k=0}^{K-1} \mathbb{I}(y_i = k) \log \widehat{p}_k(\vec{x}_i) \tag{2.11}$$

where, C is a penalty term that controls the extent of regularization, and $\mathbb{I}(\cdot)$ is the indicator function. We refer the reader to [48] for details on this optimization problem and its solution.

2.3.2 Regression Model for Cascade Prediction

In our cascade prediction problem, our goal is to predict the failure time step f_e for all branches $e \in E$ given the initial contingency $s[0]$. For this purpose, we use a set

of regression classifiers $\{\texttt{regression}_e,\ e \in E\}$, where each classifier $\texttt{regression}_e$ is associated with the prediction task of a particular branch $e \in E$. Analogous to the previous subsection, for each classifier $\texttt{regression}_e$, we treat the initial contingency $s[0]$ as the classifier's input vector $\vec{x}$ and treat the classifier's output $\widehat{y}$ as the failure step prediction $\widehat{f_e}$ with possible labels $\{0, ..., T-1\}$. Hence, our proposed regression model is composed of $|E|$ logistic regression classifiers that predict the failure steps of all branches as follows.

$$\forall e \in E, \quad \widehat{f_e} = \texttt{regression}_e(s[0]) \tag{2.12}$$

We train each of these $|E|$ classifiers separately using training data as explained in the previous subsection by treating $s[0]$ as the input vectors $\vec{x}$ and treating the true failure steps f_e as the target labels y.

Chapter 3

The Graph Neural Network Model

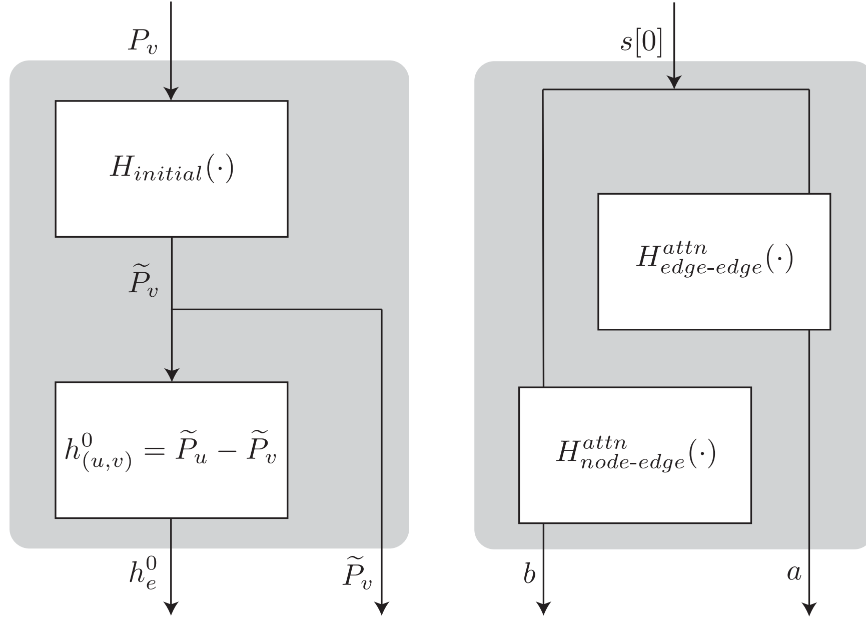
The machine learning models proposed in Chapter 2 do not take the node power injection values as input. Hence, it is not possible to generalize a single machine learning model to predict cascades under variable loading profiles. Further, the machine learning models do not take advantage of the information available regarding the graph topology. Thus, in order to build a model that can predict cascade sequences for any given loading profile, we propose a model based on GNNs that take the node power injection as one of its inputs.

3.1 Model Definition

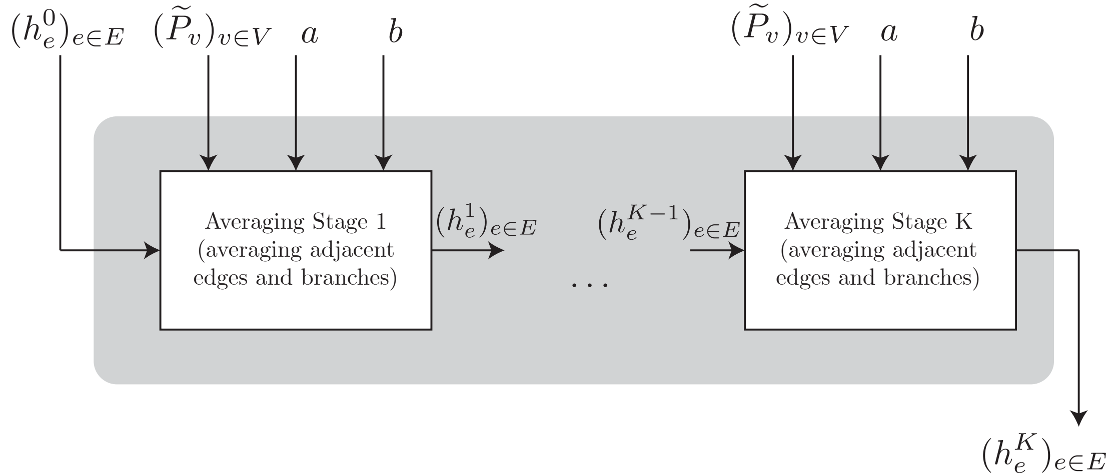
The proposed model takes as input the topology of the grid $G = (V, E)$, the initial contingency $s[0] \in \{0, 1\}^{|E|}$, and the power injection values $P_v \in \mathbb{R}$ at each node $v \in V$. The model predicts the cascade failure steps f_e of all branches $e \in E$. In this model, we process the input data in multiple stages as explained in the following subsections. Fig. 3-1 shows a block diagram that summarizes the model.

3.1.1 Initial Stage

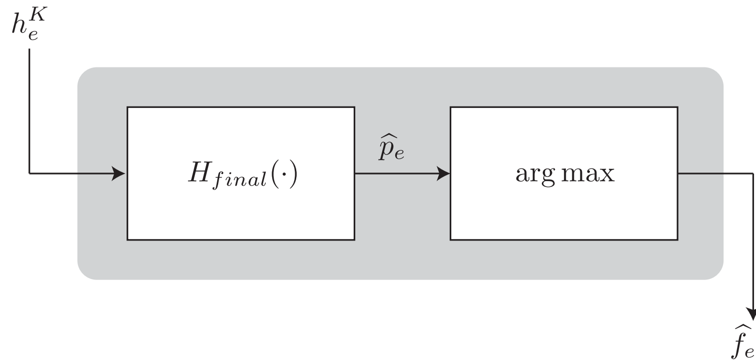
We start by removing the edges corresponding to failed branches in the initial contingency and get the new set of edges $E' = \{e \in E : s_e[0] = 1\}$. Then, we pass the



(a) Initial stage (left) and attention stage (right).



(b) Hidden stage.



(c) Final stage.

Figure 3-1: Block diagram of the GNN model.

node power injection values P_v through a neural network to obtain the hidden latent features $\widetilde{P}_v \in \mathbb{R}^L$ as follows.

$$\forall v \in V, \quad \widetilde{P}_v = H_{initial}(P_v) \tag{3.1}$$

where, the mapping $H_{initial} : \mathbb{R} \rightarrow \mathbb{R}^L$ represents a dense neural network

$$H_{initial}(P_v) = \sigma(\boldsymbol{W}^M_{initial} ... \sigma(\boldsymbol{W}^2_{initial} \sigma(\boldsymbol{W}^1_{initial} P_v + \boldsymbol{b}^1_{initial}) + \boldsymbol{b}^2_{initial}) ... + \boldsymbol{b}^M_{initial}) \tag{3.2}$$

where, the weights and biases $\boldsymbol{W}_{initial}$ and $\boldsymbol{b}_{initial}$ are trainable parameters of appropriate dimensions. These parameters will be learned during the training phase. Further, σ is a non-linear function like sigmoid, tanh, or rectified linear unit (ReLU). Note that the same neural network is being used on all the nodes $v \in V$ i.e. the weights and biases are not a function of v.

Further, we use these values $\widetilde{P}_v$ to initiate the edge hidden features h^0_e as follows.

$$\forall e = (u, v) \in E', \quad h^0_e = h^0_{(u,v)} = \widetilde{P}_u - \widetilde{P}_v \tag{3.3}$$

where, a directed edge $e \in E'$ is represented as $e = (u, v)$ with $u, v \in V$ being its source and destination nodes respectively.

3.1.2 Attention Stage

In this stage, we generate several attention coefficients that will be necessary in the later stages. But first, for an edge $e = (u, v) \in E'$, we define the set of adjacent edges as $\mathcal{N}_e = \mathcal{N}_{(u,v)} = \{(u, w) : w \in V, \ (u, w) \in E'\} \cup \{(w, v) : w \in V, \ (w, v) \in E'\}$.

Now, we generate two types of attention coefficients: the edge-to-edge coefficients a_{ed} for every two neighboring edges $e \in E', d \in \mathcal{N}_e$; and the node-to-edge coefficients (b_{eu}, b_{ev}) for all edges $e = (u, v) \in E'$ and their nodes u, v. We generate these attention coefficients by passing the given initial contingency $s[0]$ through dense neural networks as follows.

$$a = H^{attn}_{edge\text{-}edge}(s[0]), \quad b = H^{attn}_{node\text{-}edge}(s[0]) \tag{3.4}$$

where, a is the collection of edge-to-edge coefficients $a = (a_{ed})_{\{e,d \in E' : d \in \mathcal{N}_e, d \neq e\}}$, and b is the collection of node-to-edge coefficients $b = (b_{eu}, b_{ev})_{e=(u,v)\in E'}$. The mappings $H^{attn}_{\textit{edge-edge}} : \mathbb{R}^{|E|} \to \mathbb{R}^{\sum_{e\in E'}(|\mathcal{N}_e|-1)}$ and $H^{attn}_{\textit{node-edge}} : \mathbb{R}^{|E|} \to \mathbb{R}^{2|E|}$ represent two dense neural networks

$$H^{attn}_{\textit{edge-edge}}(s[0]) = \sigma(\boldsymbol{W}^{attn,M}_{\textit{edge-edge}} ... \sigma(\boldsymbol{W}^{attn,1}_{\textit{edge-edge}} s[0] + \boldsymbol{b}^{attn,1}_{\textit{edge-edge}}) ... + \boldsymbol{b}^{attn,M}_{\textit{edge-edge}}) \tag{3.5}$$

$$H^{attn}_{\textit{node-edge}}(s[0]) = \sigma(\boldsymbol{W}^{attn,M}_{\textit{node-edge}} ... \sigma(\boldsymbol{W}^{attn,1}_{\textit{node-edge}} s[0] + \boldsymbol{b}^{attn,1}_{\textit{node-edge}}) ... + \boldsymbol{b}^{attn,M}_{\textit{node-edge}}) \tag{3.6}$$

where, the weights and biases will be learned through back propagation in training, and σ is a non-linear function like sigmoid, tanh, or rectified linear unit (ReLU).

These attention coefficients will be used in the next hidden stage, where we repetitively update edge hidden features by weighted-averages of neighboring edge and node hidden features. The attention coefficients will act as weights for this purpose. They represent how much weight, or attention, needs to be given on adjacent edges and nodes while updating an edge's hidden features.

3.1.3 Averaging Stage (Hidden Layers)

In this stage, we pass the outputs of the initial stage h^0_e through a sequence of K averaging steps. In each step $k = 1, ..., K$, we calculate the weighted average of neighboring branch and node features and pass them through a neural network to obtain the new branch features. Here, the coefficients obtained as outputs from the attention stage are used as weights in the calculation of weighted averages. This is formally described in the following equation.

$$\begin{aligned} &\forall k = 1, ..., K, \ \forall e = (u,v) \in E', \\ &\qquad h^k_e = \frac{h^{k-1}_e}{|\mathcal{N}_e|} + H^k_{\textit{edge-edge}}\left(\sum_{d\in\mathcal{N}_e, d\neq e} \frac{a_{ed}}{\sqrt{|\mathcal{N}_e|}\sqrt{|\mathcal{N}_d|}} h^{k-1}_d\right) \\ &\qquad\qquad + H^k_{\textit{node-edge}}\left(\frac{b_{eu}\widetilde{P}_u + b_{ev}\widetilde{P}_v}{2}\right) \end{aligned} \tag{3.7}$$

where, K is the total number of averaging steps, h_e^k is the edge hidden feature of edge e at k-th averaging step, and $\mathcal{N}_e$ is the set of edges that are adjacent to and including e. The coefficients a_{ed} and b_{ue} are the edge-to-edge and node-to-edge attention coefficients respectively outputted by the attention stage. Finally, the functions $H^k_{\text{edge-edge}}, H^k_{\text{node-edge}} : \mathbb{R}^L \to \mathbb{R}^L$ for $k = 1, ..., K$ represent dense neural networks

$$H^k_{\text{edge-edge}}(\cdot) = \sigma(\boldsymbol{W}^{k,M}_{\text{edge-edge}} ... \sigma(\boldsymbol{W}^{k,1}_{\text{edge-edge}}(\cdot) + \boldsymbol{b}^{k,1}_{\text{edge-edge}}) ... + \boldsymbol{b}^{k,M}_{\text{edge-edge}}) \tag{3.8}$$

$$H^k_{\text{node-edge}}(\cdot) = \sigma(\boldsymbol{W}^{k,M}_{\text{node-edge}} ... \sigma(\boldsymbol{W}^{k,1}_{\text{node-edge}}(\cdot) + \boldsymbol{b}^{k,1}_{\text{node-edge}}) ... + \boldsymbol{b}^{k,M}_{\text{node-edge}}) \tag{3.9}$$

where, the weights and biases will be learned through back propagation during the training phase, and σ is a non-linear function like sigmoid, tanh, or rectified linear unit (ReLU). Note that these neural networks are the same for all edges.

3.1.4 Final Stage

In the final stage, we predict the failure step probability values $\widehat{p}_e := [\widehat{p}_{e,0}, ..., \widehat{p}_{e,T-1}]$ for all branches $e \in E'$, where each entry $\widehat{p}_{e,t}$ is the predicted probability that branch e fails at time t. We do this by passing the output of last averaging step h_e^K through a dense neural network as follows.

$$\forall e \in E', \quad \widehat{p}_e = H_{final}(h_e^K) \in [0,1]^T \tag{3.10}$$

where, the function $H_{final} : \mathbb{R}^L \to \mathbb{R}^T$ represents a dense neural network

$$H_{final}(h_e^K) = \sigma(\boldsymbol{W}^M_{final} ... \sigma(\boldsymbol{W}^2_{final} \sigma(\boldsymbol{W}^1_{final} h_e^K + \boldsymbol{b}^1_{final}) + \boldsymbol{b}^2_{final}) ... + \boldsymbol{b}^M_{final}) \tag{3.11}$$

where, the weights $\boldsymbol{W}_{final}$ and biases $\boldsymbol{b}_{final}$ will be learnt during the training phase, and σ is a non-linear function like sigmoid, tanh, or rectified linear unit (ReLU). However, in this neural network, the last non-linear function needs to be softmax, so that its outputs represent valid probability values. Finally, the failure steps of edges

is predicted by picking the index with the highest predicted probability value,

$$\forall e \in E', \quad \widehat{f}_e = \arg \max_{t=0,...,T-1} \widehat{p}_{e,t}. \tag{3.12}$$

3.2 Model Training

In this section, we discuss the training of the proposed GNN model using the cross entropy loss function and the Adam optimizer. The goal here is to learn all the trainable weights and biases present in all the stages of our model. This includes the weights and biases of the following neural networks: the neural network $H_{initial}$ in the initial stage, the neural networks $H^{attn}_{edge\text{-}edge}$ and $H^{attn}_{node\text{-}edge}$ in the attention stage, the neural networks $H^{k}_{edge\text{-}edge}$ $(k = 1, ..., K)$ and $H^{k}_{node\text{-}edge}$ $(k = 1, ..., K)$ in the averaging stage, and the neural network H_{final} in the final stage.

We start by initializing these weights and biases randomly. We then update these weights and biases in multiple iterations. In each iteration, we first sample a random batch of cascade samples $\mathcal{D}_{batch} \subset \mathcal{D}_{train}$ from the training dataset $\mathcal{D}_{train}$. For these samples, we run two routines called the forward pass and back propagation. In forward pass, we obtain the model's prediction and calculate the cross entropy loss between the prediction and the true labels. In back propagation, we calculate the gradients of the loss with respect to each parameter (weights and biases) and update the parameters using this loss gradient. We repeat this process over multiple iterations of random batches until the loss value has been reduced to a required minimum threshold value. We discuss the details of these routines in the following subsections.

3.2.1 Forward Pass

During forward pass, we simply run the GNN model on all the samples in $\mathcal{D}_{batch}$, obtain the model's predicted output values, and save these values. Specifically, we run a forward pass through the model and save the model's predicted failure step probability values $\widehat{p}^d_e := [\widehat{p}^d_{e,0}, ..., \widehat{p}^d_{e,0}]$ for all branches $e \in E$, where each entry $\widehat{p}^d_{e,t}$ is the predicted probability that the branch e fails at time step t. We also save all the

intermediate variables calculated during the forward pass like edge hidden features h_e^k for all $e \in E$, $k = 1, ...K$, transformed node injection values $\widetilde{P}_v$ for all $v \in V$, and attention coefficients a_{ed}, b_{ev} for all $e, d \in E$, $v \in V$.

Now that we have the model's prediction, we can calculate the cross entropy loss between these predicted values and the true values. Let f_e^d for any branch $e \in E$ be the true time step at which the branch failed. Then the cross entropy loss for this train sample d and branch e is defined as $C_e^d := -\log \widehat{p}^d_{e,f_e^d}$. Note that the constraint $\sum_t \widehat{p}^d_{e,t} = 1$ is already satisfied since the final layer of the model's final stage is a softmax layer. So it is enough to penalize only the predicted probability value of the index corresponding to $t = f_e^d$. Finally, we can accumulate the loss of all the branches and samples of the training batch by calculating the following average value.

$$C_{batch} := \frac{1}{|\mathcal{D}_{batch}|} \sum_{d \in \mathcal{D}_{batch}} \frac{1}{|E|} \sum_{e \in E} C_e^d = \frac{1}{|\mathcal{D}_{batch}|} \sum_{d \in \mathcal{D}_{batch}} \frac{1}{|E|} \sum_{e \in E} -\log \widehat{p}^d_{e,f_e^d}. \tag{3.13}$$

3.2.2 Back Propagation

Now to update the model parameters, we calculate the gradients of the cross entropy loss with respect to each parameter of the model using the chain rule. We do this in a backward manner starting from the final stage and going back to the averaging stage, and finally to the attention and initial stages.

Final Stage

Consider the final stage of the model. Recall that the final stage includes the following operations.

$$\forall e \in E', \quad \widehat{p}_e = H_{final}(h_e^K). \tag{3.14}$$

Hence, the final stage contains the dense neural network $H_{final}(\cdot)$, whose weights and biases need to be learned. Let the parameters of this dense neural network be $\boldsymbol{W}_{final}$ and $\boldsymbol{b}_{final}$ (weights and biases respectively). Expanding the internal calcula-

tions within the neural network $H_{final}(\cdot)$, the final stage now looks as follows.

$$\forall e \in E', \quad \widehat{p}_e = H_{final}(h_e^K) = \sigma(\boldsymbol{W}_{final} h_e^K + \boldsymbol{b}_{final}) \tag{3.15}$$

where, σ is a non-linear activation function (in this case, softmax). Note that the dense neural networks in practice and in our experiments have more than one layers $H_{final}(h_e^K) = \sigma(\boldsymbol{W}^M_{final} ... \sigma(\boldsymbol{W}^2_{final} \sigma(\boldsymbol{W}^1_{final} h_e^K + \boldsymbol{b}^1_{final}) + \boldsymbol{b}^2_{final}) ... + \boldsymbol{b}^M_{final})$. However, here we describe for brevity the neural network with only one layer. We update these parameters $\boldsymbol{W}_{final}$ and $\boldsymbol{b}_{final}$ as follows.

1. We can first calculate the gradient of the cross entropy loss $\nabla_{\widehat{p}_e} C_{batch}$ with respect to the final stage output $\widehat{p}_e$. From (3.13), the gradient is given as

$$\begin{aligned}
\nabla_{\widehat{p}_e} C_{batch} &= \frac{1}{|\mathcal{D}_{batch}|} \sum_{d \in \mathcal{D}_{batch}} \frac{1}{|E|} \nabla_{\widehat{p}_e} \left(\sum_{e' \in E} C^d_{e'} \right) \\
&= \frac{1}{|\mathcal{D}_{batch}|} \sum_{d \in \mathcal{D}_{batch}} \frac{1}{|E|} \nabla_{\widehat{p}_e} C^d_e \\
&= \frac{1}{|\mathcal{D}_{batch}||E|} \sum_{d \in \mathcal{D}_{batch}} \nabla_{\widehat{p}_e} \left(-\log \widehat{p}^d_{e, f^d_e} \right) \\
&= \frac{1}{|\mathcal{D}_{batch}||E|} \sum_{d \in \mathcal{D}_{batch}} -\frac{1}{\widehat{p}^d_{e, f^d_e}} \mathbf{I}_{f^d_e}.
\end{aligned}$$

where, $\mathbf{I}_t$ represents a one-hot vector with an unit entry at t-th element and zeros everywhere else.

2. Using this gradient, we can calculate the loss gradients $\nabla_{\boldsymbol{W}_{final}} C_{batch}$ and $\nabla_{\boldsymbol{b}_{final}} C_{batch}$ with respect to the parameters $\boldsymbol{W}_{final}$ and $\boldsymbol{b}_{final}$ from (3.15) using chain rule as

$$\nabla_{\boldsymbol{W}_{final}} C_{batch} = \left(\nabla_{\widehat{p}_e} C_{batch} \odot \sigma'(\boldsymbol{W}_{final} h_e^K + \boldsymbol{b}_{final}) \right) \cdot \left(h_e^K \right)^T, \tag{3.16}$$

$$\nabla_{\boldsymbol{b}_{final}} C_{batch} = \nabla_{\widehat{p}_e} C_{batch} \odot \sigma'(\boldsymbol{W}_{final} h_e^K + \boldsymbol{b}_{final}) \tag{3.17}$$

where, $\odot$ represents element-wise multiplication and σ' is the derivative of the

non-linear activation function. Recall that we have saved the values of all the intermediate variables during the forward pass. So we do have access to values like h_e^K making the gradient calculation possible.

3. Now, the values of the parameters can be updated as

$$\boldsymbol{W}_{final} \leftarrow \boldsymbol{W}_{final} - \eta \nabla_{\boldsymbol{W}_{final}} C_{batch}, \tag{3.18}$$

$$\boldsymbol{b}_{final} \leftarrow \boldsymbol{b}_{final} - \eta \nabla_{\boldsymbol{b}_{final}} C_{batch} \tag{3.19}$$

where, η is the step size. Note that the update rule shown above is followed by the gradient descent optimizer. However, we use the Adam optimizer to train our model, whose update rule is slightly more involved. We refer the reader to [49] for details about the parameter update rule of the Adam optimizer.

4. Additionally, we can also calculate the gradient $\nabla_{h_e^K} C_{batch}$ with respect to the input h_e^K to this final stage. Applying chain rule on (3.15),

$$\nabla_{h_e^K} C_{batch} = (\boldsymbol{W}_{final})^T \cdot \left(\nabla_{\widehat{p}_e} C_{batch} \odot \sigma'(\boldsymbol{W}_{final} h_e^K + \boldsymbol{b}_{final})\right) \tag{3.20}$$

This gradient calculation will be useful to extend the chain rule to other stages as explained in the later paragraphs.

After we update the parameters of the final stage, we can use the chain rule again to propagate further back to the averaging stage, the attention stage and the initial stage. We explain this briefly below without going into the mathematical details.

Averaging Stage

Recall that the averaging stage contains dense neural networks $H^k_{edge\text{-}edge}(\cdot)$ and $H^k_{node\text{-}edge}(\cdot)$ for all $k = 1, ..., K$, whose parameters need to be learned in training. Let the weights of these neural networks be $\boldsymbol{W}^k_{edge\text{-}edge}$ and $\boldsymbol{W}^k_{node\text{-}edge}$. Let the biases of these networks be $\boldsymbol{b}^k_{edge\text{-}edge}$ and $\boldsymbol{b}^k_{node\text{-}edge}$. To update the parameters of these neural networks, we follow a similar procedure as the final stage. We start with the gradient $\nabla_{h_e^K} C_{batch}$ which

has already been calculated in (3.20) and do the following for each $k = K, K-1, ..., 1$.

1. Using the chain rule and using the calculated gradient value $\nabla_{h_e^k} C_{batch}$, we can calculate the loss gradients $\nabla_{\boldsymbol{W}^k_{edge\text{-}edge}} C_{batch}$, $\nabla_{\boldsymbol{W}^k_{node\text{-}edge}} C_{batch}$, $\nabla_{\boldsymbol{b}^k_{edge\text{-}edge}} C_{batch}$, and $\nabla_{\boldsymbol{b}^k_{node\text{-}edge}} C_{batch}$ with respect to the weights and biases.

2. Once we have the gradients, we can update the weights and parameters as

$$\boldsymbol{W}^k_{edge\text{-}edge} \leftarrow \boldsymbol{W}^k_{edge\text{-}edge} - \eta \nabla_{\boldsymbol{W}^k_{edge\text{-}edge}} C_{batch} \tag{3.21}$$

$$\boldsymbol{W}^k_{node\text{-}edge} \leftarrow \boldsymbol{W}^k_{node\text{-}edge} - \eta \nabla_{\boldsymbol{W}^k_{node\text{-}edge}} C_{batch} \tag{3.22}$$

$$\boldsymbol{b}^k_{edge\text{-}edge} \leftarrow \boldsymbol{b}^k_{edge\text{-}edge} - \eta \nabla_{\boldsymbol{b}^k_{edge\text{-}edge}} C_{batch} \tag{3.23}$$

$$\boldsymbol{b}^k_{node\text{-}edge} \leftarrow \boldsymbol{b}^k_{node\text{-}edge} - \eta \nabla_{\boldsymbol{b}^k_{node\text{-}edge}} C_{batch} \tag{3.24}$$

3. Finally, using the value of $\nabla_{h_e^k} C_{batch}$ and using chain rule again, we can calculate the gradient $\nabla_{h_e^{k-1}} C_{batch}$ required to update the parameters of step $k-1$. We repeat this procedure until we have updated the parameters of all K steps of the averaging stage.

Attention Stage and Initial Stage

Using the gradient values calculated in the previous step, we can further apply chain rule to calculate the loss gradients $\nabla_a C_{batch}$, $\nabla_b C_{batch}$ with respect to the attention coefficients a and b. This can be used to update the parameters of neural networks $H^{attn}_{edge\text{-}edge}(\cdot)$ and $H^{attn}_{node\text{-}edge}(\cdot)$ in the attention stage.

Similarly, we can calculate the loss gradients $\nabla_{\widetilde{P}_v} C_{batch}$ with respect to the transformed node injection values $\widetilde{P}_v$ and with this, we can update the parameters of the neural network $H_{initial}(\cdot)$ in the initial stage of the model.

This way, we can use back propagation to update all the parameters within the GNN model at each iteration of the training process. We keep repeating the feed forward and back propagation pair with random training batches until the model's prediction loss has reached a required minimum threshold.

Chapter 4

Results: Data Synthesis and Design Specifications

4.1 Data Synthesis

In this section, we explain the data synthesis approach we use to generate the data useful for training the proposed models. We follow the cascading failure simulator (CFS) oracle proposed in [15] to generate the training data. However, historic data obtained from utility records or synthetic data generated from other oracles can also be used to train our model without any changes to its architecture.

4.1.1 The Cascading Failure Simulator Oracle

As summarized in Algorithm 1, the CFS oracle simulates the cascade process for a given initial contingency $s[0]$ and power injection values $(P_v)_{v\in V}$, and outputs the failure steps $f := (f_e)_{e\in E}$. The CFS oracle treats branch failures deterministically, a branch e is treated to be failed whenever the power flow g_e through it crosses its given capacity value c_e. The oracle also assumes that optimal redispatch is not applicable within a fast cascade. However, it does perform urgent load shedding and generation curtailment in case of power mismatch. We implement the CFS oracle in MATLAB, where we use the MATPOWER toolbox [50] to get the graph topology and branch

capacity values. We also use the toolbox's DC power flow solver to calculate the branch power flow values.

Algorithm 1 Simulating failure cascade using the CFS oracle.

Input: The grid topology $G = (V, E)$, the initial contingency $s[0]$, the capacity values of branches $(c_e)_{e \in E}$, and the power injection values $(P_v)_{v \in V}$.
Output: The branch failure steps in the cascade $(f_e)_{e \in E}$.
initialize $t \leftarrow 0$; overloaded $\leftarrow$ true.
while overloaded **do**
  1) $E \leftarrow \{e \in E : s_e[t] = 1\}$.
  2) Detect all the islands (disconnected sub-graphs) that appear. If the whole network is connected, then it can be viewed as the only island.
  for each island **do**
    a) Specify a bus as the slack bus of the island.
    b) Rebalance the power injection within the island by either generation curtailment or load shedding depending on whether the supply exceeds demand in the island.
    c) Recompute the power flow values g_e within the island for each branch of the island.
  end for
  3) $s[t+1] \leftarrow s[t]; \quad \forall e \in E : g_e > c_e, \; s_e[t+1] \leftarrow 0$.
  4) **if** $s[t+1] = s[t]$ **then** overloaded $\leftarrow$ false **end if**
  5) $t \leftarrow t + 1$.
end while
return $f = \sum_t s[t]$.

4.1.2 Data for Machine Learning Models

Using the CFS oracle, we generate a data pool $\mathcal{D}_\alpha$ of size M as summarized in Algorithm 2. Each sample in the data pool $\mathcal{D}_\alpha$ is a tuple $(s[0], (f_e)_{e \in E})$ containing a random initial contingency and the corresponding cascade failure steps respectively. All the samples in this dataset $\mathcal{D}_\alpha$ are generated by scaling the node power injections by a constant value α. Specifically, in each sample, we first generate random $|E| - 2$ initial contingencies $s[0]$ by selecting two branches randomly, say e_1, e_2, and setting their states to failed $s_{e_1}[0] = s_{e_2}[0] = 0$. Then, we scale the default power injection values obtained from the MATPOWER toolbox uniformly across all nodes by a constant given scaling value α to get $(P_v)_{v \in V}$. Finally, we use the CFS oracle described in Algorithm 1 to get the cascade failure steps $(f_e)_{e \in E}$ for each sample.

Algorithm 2 Generating the failure cascade data pool for machine learning models.

Input: The grid topology $G = (V, E)$, power injection values $(P_v)_{v \in V}$ that are already scaled by a constant value α, and required data pool size M.
Output: The failure cascade data pool $\mathcal{D}_\alpha$.
initialize $\mathcal{D}_\alpha \leftarrow \{\}$.
while $|\mathcal{D}_\alpha| < M$ **do**
1) $s[0] \leftarrow$ Random $|E| - 2$ initial contingency.
2) $(f_e)_{e \in E} \leftarrow$ CFS oracle's output for $(s[0], (P_v)_{v \in V}, G)$.
3) $\mathcal{D}_\alpha \leftarrow \mathcal{D}_\alpha$ appended with $(s[0], (P_v)_{v \in V}, (f_e)_{e \in E})$.
end while
return $\mathcal{D}_\alpha$.

4.1.3 Data for the GNN Model

We generate a data pool $\mathcal{D}$ of size M as summarized in Algorithm 3. Each sample in the data pool $\mathcal{D}$ is a tuple $(s[0], (P_v)_{v \in V}, (f_e)_{e \in E})$ containing a random initial contingency, randomly scaled power injection values, and the corresponding cascade failure steps respectively. Specifically, in each sample, we first generate random $|E| - 2$ initial contingencies $s[0]$ by selecting two branches randomly, say e_1, e_2, and setting their states to failed $s_{e_1}[0] = s_{e_2}[0] = 0$. Then, we scale the default power injection values obtained from the MATPOWER toolbox uniformly across all nodes by a random scaling value $\alpha \sim \text{Unif}[1, 2]$ to get $(P_v)_{v \in V}$. This way, our data pool contains cascades at various loading and initial contingencies. Finally, we use the CFS oracle described in Algorithm 1 to get the cascade failure steps $(f_e)_{e \in E}$ for each sample.

Algorithm 3 Generating the failure cascade data pool for the GNN model.

Input: The grid topology $G = (V, E)$, default power injection values $(P_v^0)_{v \in V}$, and required data pool size M.
Output: The failure cascade data pool $\mathcal{D}$.
initialize $\mathcal{D} \leftarrow \{\}$.
while $|\mathcal{D}| < M$ **do**
1) $s[0] \leftarrow$ Random $|E| - 2$ initial contingency.
2) $\alpha \leftarrow \text{Unif}[1, 2]; \quad \forall v \in V,\ P_v \leftarrow \alpha P_v^0$.
3) $(f_e)_{e \in E} \leftarrow$ CFS oracle's output for $(s[0], (P_v)_{v \in V}, G)$.
4) $\mathcal{D} \leftarrow \mathcal{D}$ appended with $(s[0], (P_v)_{v \in V}, (f_e)_{e \in E})$.
end while
return $\mathcal{D}$.

4.2 Model Design Specifications

In this section, we discuss the design details of the proposed models that we use to evaluate the performance of the models in this work.

4.2.1 GNN Model

To train the GNN model, we generate data pools $\mathcal{D}$ for two power grids, IEEE89 and IEEE118, which are available in the MATPOWER toolbox using the data synthesis methods explained in Section 4.1.3. The two generated datasets each contain 200,000 cascade sequence samples, each sample simulated on a random initial contingency and random uniform scaling as discussed in the previous section. We get the required graph topology, default power injection values, and branch capacities (we set the unavailable capacities to twice the default power flows through the branches) from the MATPOWER toolbox. In both cases, we split 90% of the dataset $\mathcal{D}$ into train $\mathcal{D}_{train}$ and 10% to test $\mathcal{D}_{test}$ sets.

We train two instances of the proposed GNN model, one corresponding to the IEEE89 case and another to the IEEE118 case. We build the instances in Python using the neural network modules available in the PyTorch library [51]. The implementation can be found in `https://github.com/sathwikchadaga/failure-cascade`.

4.2.2 Machine Learning Models

To train the machine learning models, we generate data pools $\mathcal{D}_\alpha$ for three power grids, IEEE89, IEEE118, and IEEE1354, which are available in the MATPOWER toolbox using the data synthesis methods explained in Section 4.1.2. For each of these IEEE systems, we generate eleven datasets $\mathcal{D}_\alpha$ for scaling values $\alpha = 1.00, 1.10, ..., 2.00$. The generated datasets each contain 10,000 cascade sequence samples, each sample simulated on a random initial contingency as discussed in the previous section. We get the required graph topology, default power injection values, and branch capacities (we set the unavailable capacities to twice the default power flows through the branches) from the MATPOWER toolbox. In all cases, we split 90% of the dataset $\mathcal{D}_\alpha$ into

train $\mathcal{D}^{\alpha}_{train}$ and 10% to test $\mathcal{D}^{\alpha}_{test}$ sets.

We train instances of the machine learning models (naive Bayes, SVM, and regression) corresponding to the IEEE89, IEEE118, and IEEE1354 cases. Note, however, that the machine learning models do not take node power injection values into consideration, making them specific to a single loading profile. Hence, it is impossible to generalize a single machine learning model over all random load scaling values. Thus, for each IEEE system and for each of SVM, naive Bayes, and regression models, we build multiple instances of the model for different load scaling values α and train them using the generated data pool $\mathcal{D}^{\alpha}_{train}$. We build these models in Python using the Scikit-Learn library [52]. The implementation can be found in `https://github.com/sathwikchadaga/failure-cascade`.

4.2.3 Influence Model

Finally, we use the performance of the influence model [20] as a benchmark since this model can evaluate metrics in almost the same granularity level as the GNN and the machine learning models. Similar to the machine learning models, we cannot generalize a single influence model over all random load scaling values. Thus, we train multiple instances of the influence models specialized for different load scaling values and IEEE systems. The implementation can be found in `https://github.com/sathwikchadaga/failure-cascade`.

Chapter 5

Results: Graph Level Performance

In this chapter, we evaluate the trained models on several metrics that capture the graph-level prediction performance. We first define several evaluation metrics in Section 5.1 like the graph-level failure size error rate, final state error rate, and failure step error rate. We then show the performance of the proposed machine learning models in Section 5.2 using the defined metrics. We show that the regression model performs the best among them. Finally, we show the performance of the GNN model in Section 5.3. Here, we show that the GNN model, in addition to being generic over randomly scaled loading values, outperforms the load-specific influence models and competes with the load-specific regression models. An implementation of these models can be found in `https://github.com/sathwikchadaga/failure-cascade`.

5.1 Performance Metrics Definitions

5.1.1 Failure Size Error Rate

The cascade failure size is defined as the number of branches in the failed state at the end of a cascade. The failure size error rate l^{α}_{size} at a load scaling value α encapsulates the average error in model's prediction of the cascade failure size. Formally, let $\mathcal{D}^{\alpha}_{test} \subset \mathcal{D}_{test}$ be the set of test samples whose input load scaling is α. For a sample $d \in \mathcal{D}^{\alpha}_{test}$ (with a random $|E| - 2$ initial contingency), let $E^{d}_{failed} \subset E$ be the set of

edges that truly fail and let $\widehat{E}^d_{failed} \subset E$ be the set of edges that are predicted to have failed, then failure size error rate at this load scaling value is given by

$$l^{\alpha}_{size} = \frac{1}{|\mathcal{D}^{\alpha}_{test}|} \sum_{d \in \mathcal{D}^{\alpha}_{test}} \frac{||E^d_{failed}| - |\widehat{E}^d_{failed}||}{|E^d_{failed}|}. \tag{5.1}$$

5.1.2 Final State Error Rate

The final state error rate l^{α}_{state} at a load scaling value α represents the error in model's prediction of the network state at the end of the cascade. Formally, if the true final state of a cascade sample $d \in \mathcal{D}^{\alpha}_{test}$ (with a random $|E| - 2$ initial contingency) is $s^d[T] = (s^d_e[T])_{e \in E}$, and its predicted final state is $\hat{s}^d[T] = (\hat{s}^d_e[T])_{e \in E}$, then the final state error rate at this load scaling value is given by

$$l^{\alpha}_{state} = \frac{1}{|\mathcal{D}^{\alpha}_{test}|} \sum_{d \in \mathcal{D}^{\alpha}_{test}} \frac{1}{|E|} \sum_{e \in E} \left| s^d_e[T] - \hat{s}^d_e[T] \right|. \tag{5.2}$$

5.1.3 Failure Step Error Rate

Failure step error $l^{\alpha}_{failure-step}$ is the error between model's predicted failure steps and the true failure steps at a load scaling value α. If the true failure steps in a sample $d \in \mathcal{D}^{\alpha}_{test}$ (with a random $|E| - 2$ initial contingency) is $f^d = (f^d_e)_{e \in E}$ and the predicted failure steps are $\widehat{f}^d = (\widehat{f}^d_e)_{e \in E}$, then the failure step error at this load scaling value is given by

$$l^{\alpha}_{failure-step} = \frac{1}{|\mathcal{D}^{\alpha}_{test}|} \sum_{d \in \mathcal{D}^{\alpha}_{test}} \frac{1}{|E|} \sum_{e \in E} |f^d_e - \widehat{f}^d_e|. \tag{5.3}$$

5.2 Results for the Machine Learning Models

In this section, we present the graph-level performance metrics defined in Section 5.1 for the proposed machine learning models: the regression model, the SVM model, and the naive Bayes model. Note, however, that none of these models take node power injection values into consideration, making them specific to a single loading profile. Hence, it is impossible to generalize a single model over all random load scaling values

α. Thus, we build multiple instances of these models, each trained on a unique load scaling value, and plot their error rates as a function of their corresponding load scaling values. Specifically, for a given IEEE bus, we train eleven instances of the regression model, eleven instances of the SVM model, and eleven instances of the naive Bayes model that are specific to load scaling values of $1.00, 1.10, ..., 2.00$.

5.2.1 Failure Size Error Rate

Fig. 5-1 shows the failure size error rates of the regression, SVM, and naive Bayes models trained and tested on IEEE89, IEEE118, and IEEE1354 (left to right) datasets. Note again that, since these models cannot be generalized over variable loading values, each of the points in the plot corresponds to a different instance of the model that is specifically trained for its corresponding load scaling value. As can be seen, the regression models have the best performance followed by the SVM models at all load scaling values. The naive Bayes models perform worse than others as expected because of their independence assumption.

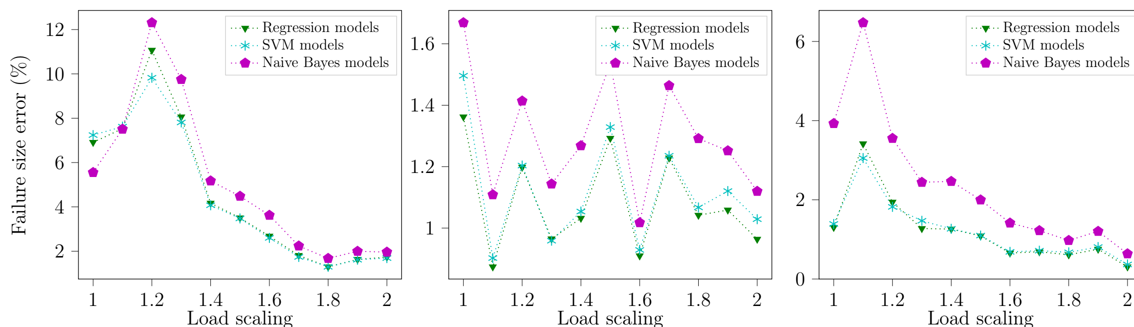


Figure 5-1: Failure size error rates l^{α}_{size} of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α.

5.2.2 Final State Error Rate

Similar to the previous metric, Fig. 5-2 shows the final state error rates of various load-specific instances of the regression, SVM, and naive Bayes models trained and tested on IEEE89, IEEE118, and IEEE1354 (left to right) datasets. As can be seen, the regression models have the best performance at all load scaling values. The worst

case final state error rate is 4.1%, which is experienced by the naive Bayes model at a load scaling value of 1.3 for the IEEE89 case.

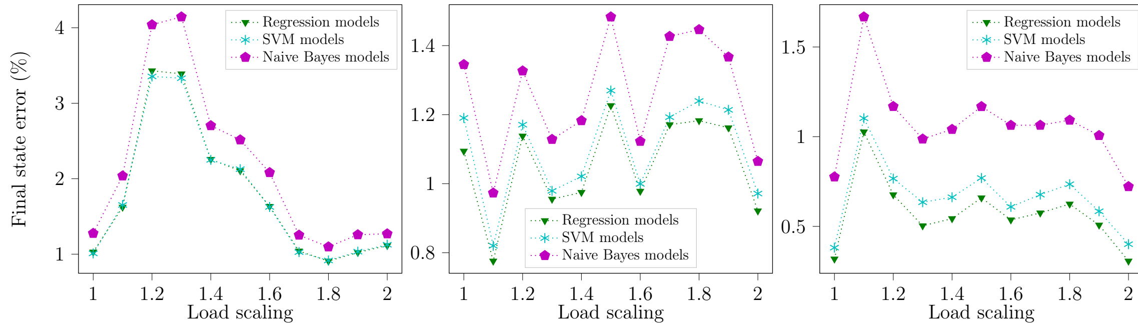


Figure 5-2: Final state error rates l^{α}_{state} of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α.

5.2.3 Failure Step Error Rate

Similar to the previous metric, Fig. 5-3 shows the failure step error rates of various load-specific instances of the regression, SVM, and naive Bayes models trained and tested on IEEE89, IEEE118, and IEEE1354 (left to right) datasets. Similar to the previous metrics, the regression models outperform the naive Bayes and SVM models at all scaling values.

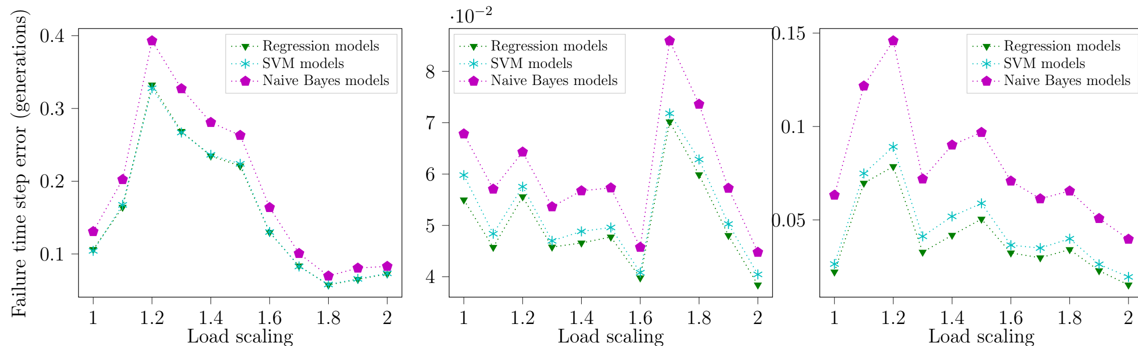


Figure 5-3: Failure step error rates $l^{\alpha}_{failure\text{-}step}$ of various machine learning models for IEEE89, IEEE118, and IEEE1354 (left to right) against load scaling values α.

5.3 Results for the GNN Models

In this section, we present the graph-level performance metrics defined in Section 5.1 for the GNN model. We have seen in the previous section that the regression models perform the best among proposed machine learning models. Hence, we compare the GNN model against the regression models. We also present the performance metrics of the influence model [20] as a benchmark since this model can evaluate metrics in almost the same granularity level as the GNN and the regression models.

Similar to the regression model, the influence model does not take node power injection values into consideration, making it specific to a single loading profile. Hence, it is impossible to generalize a single influence model over all random load scaling values α. Thus, we build multiple instances of the influence model, each trained on a unique load scaling value, and compare our single generalized GNN model against them. Specifically, for a given IEEE bus, we train eleven instances of the influence model and eleven instances of the regression model that are specific to load scaling values of $1.00, 1.10, ..., 2.00$. Whereas, we train only one instance of the GNN model for a given bus, which is generic over all load scaling values.

5.3.1 Failure Size Error Rate

Fig. 5-4 shows the failure size error rates of the two GNN models trained and tested on IEEE89 (left) and IEEE118 (right) datasets. Here, the GNN model's error rates at nearby load scaling values are grouped together and their bin averages are plotted for clarity. The plot also shows the error rates of multiple load-specific instances of the regression and influence models trained and tested on IEEE89 (left) and IEEE118 (right) datasets.

As can be seen, the GNN model experiences a worst case relative error of 15% for IEEE89 and 2% for IEEE118. Further, for a given IEEE bus, even though we are using a single instance of the GNN model generic to all scaling values, it has lower error rates than all the load-specific influence models. Additionally, its performance is comparable to all the load-specific regression models. Finally, we see that the error

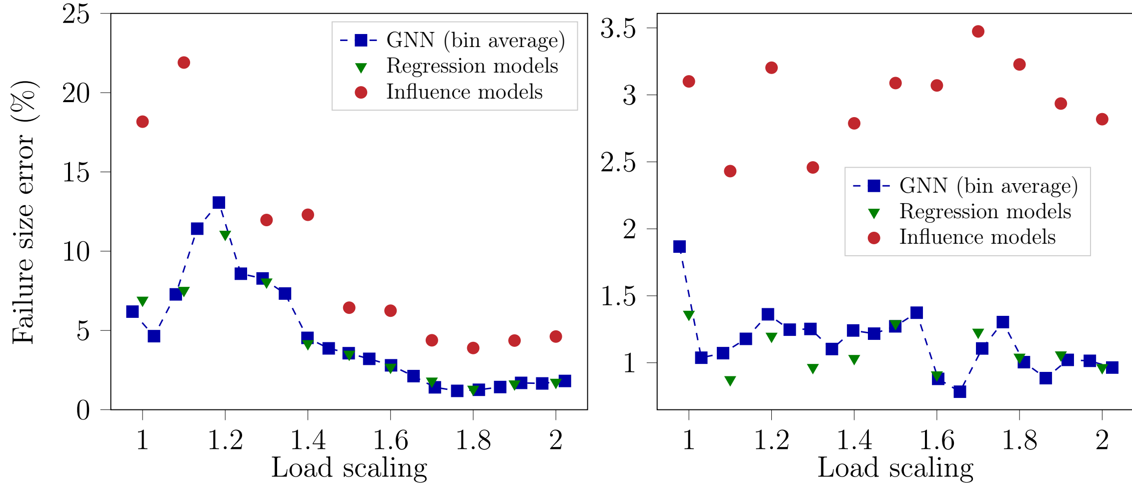


Figure 5-4: Failure size error rates l^{α}_{size} of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α.

rates are higher for all the models when the load scaling values are low for IEEE89. This is because when the system loading is low, the true failure sizes are small. This increases the error rate metric, as it is calculated relative to the true sizes.

5.3.2 Final State Error Rate

Similar to the previous metric, Fig. 5-5 shows the final state error rates of two instances of the GNN model trained and tested on IEEE89 (left) and IEEE118 (right) datasets. The figure also shows the error rates of multiple load-specific instances of the regression and influence models trained and tested on IEEE89 (left) and IEEE118 (right) datasets.

As can be seen, both instances of the GNN model are better by around 2% (nearly a factor of 2) than the load-specific influence models at all load scaling values. Further, the GNN models' performance is similar to that of the load-specific regression models at almost all of the load scaling values. The worst final state error rate experienced by the GNN model is 4.8%, which is at a loading value of 1.23 for the IEEE89 case.

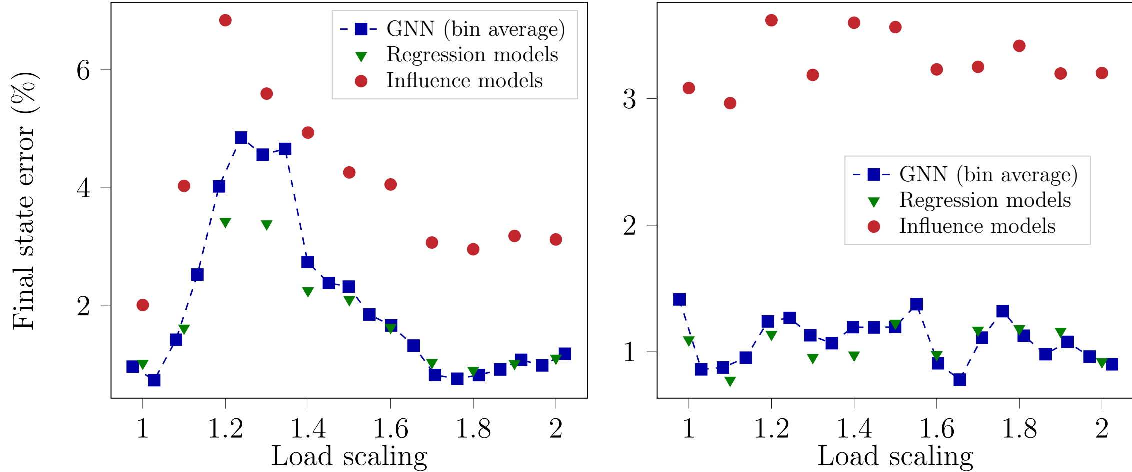


Figure 5-5: Final state error rates l^{α}_{state} of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α.

5.3.3 Failure Step Error Rate

Similar to the previous metric, Fig. 5-6 shows the failure step error rates of two instances of the GNN model trained and tested on IEEE89 (left) and IEEE118 (right) datasets. The figure also shows the error rates of multiple load-specific instances of the regression and influence models trained and tested on IEEE89 (left) and IEEE118 (right) datasets.

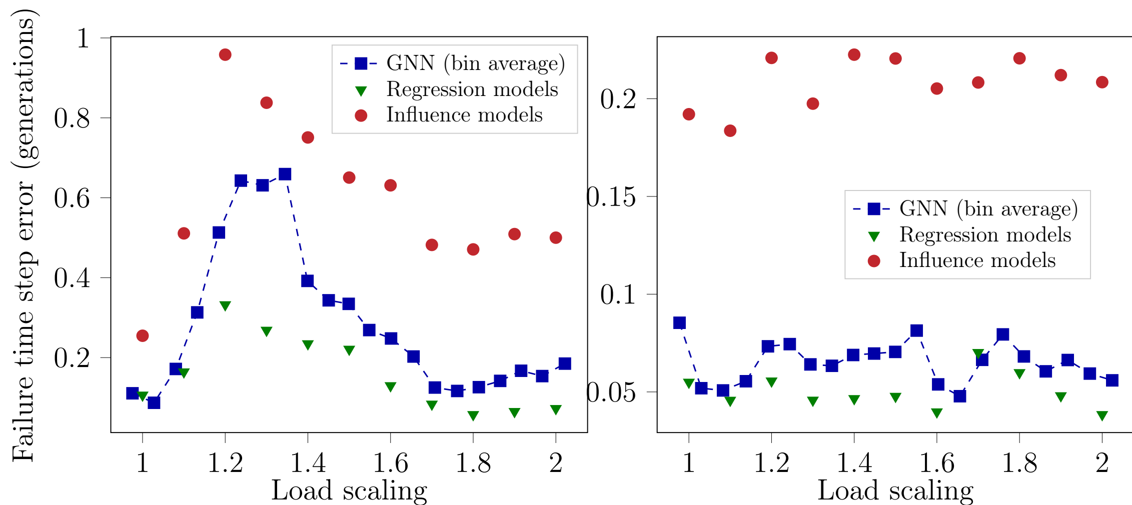


Figure 5-6: Failure step error rates $l^{\alpha}_{failure\text{-}step}$ of various models for IEEE89 (left) and IEEE118 (right) against load scaling values α.

As can be seen, the GNN model's worst error rates are 0.7 time steps (or genera-

tions) and 0.1 time steps for IEEE89 and IEEE118 respectively. For both buses, the generic GNN model outperforms all the influence model instances built specifically for their corresponding load scaling values. However, it is worse than the load-specific regression models.

Chapter 6

Results: Branch Level Performance

In this chapter, we evaluate the trained models on several metrics that represent the performance in prediction of several branch features of the cascade. We first define several evaluation metrics in Section 6.1 like the branch-level final state error rate and failure step error rate. We then show the performance of the proposed machine learning models in Section 6.2 using the defined metrics. We show that the regression model performs the best among them. Finally, we show the performance of the GNN model in Section 6.3. Here, we show that the GNN model, in addition to being generic over randomly scaled loading values, outperforms the load-specific influence models and compares well with the load-specific regression models. An implementation of these models can be found in `https://github.com/sathwikchadaga/failure-cascade`.

6.1 Performance Metrics Definitions

In this section, we define several performance metrics that are useful to evaluate the model's branch-level prediction performance.

6.1.1 Branch Failure Frequency

Before we define the branch-level performance metrics, we define the branch failure frequency $l_{freq,e}$ of branch $e \in E$. Let $\mathcal{D}^{e}_{failed} \subset \mathcal{D}_{train}$ be the set of train samples

where the branch has eventually failed in the cascade but not as part of the initial contingency, then the branch failure frequency is given by the following fraction.

$$l_{freq,e} = \frac{|\mathcal{D}^{e}_{failed}|}{|\mathcal{D}_{train}|}. \tag{6.1}$$

This is an important value as it captures the prediction differences between branches with different failure frequencies. For example, predicting the features of a branch that rarely fails is easier than a branch that fails half the time. Hence, in the later sections, we will be plotting the branch-level performance metrics as a function of the branch failure frequencies.

6.1.2 Branch Final State Error Rate

Branch final state error rate $l_{state,e}$ for a branch $e \in E$ represents the ratio of test samples with incorrect final state predictions. Say, $\mathcal{D}^{e}_{wrong} \subset \mathcal{D}_{test}$ represents the set of test samples in which the model wrongly predicted the final state of edge e. Further, say $\mathcal{D}^{e}_{initial} \subset \mathcal{D}^{e}_{wrong}$ be the samples in which branch e failed as part of the initial contingency. We do not count such samples as predicting their states is trivial, hence we define the final state error as the following ratio.

$$l_{state,e} = \frac{|\mathcal{D}^{e}_{wrong}| - |\mathcal{D}^{e}_{initial}|}{|\mathcal{D}_{test}|}. \tag{6.2}$$

6.1.3 Branch Failure Step Error Rate

Branch failure step error rate $l_{failure-step,e}$ is the error in prediction of the failure step of a particular branch $e \in E$ across all test samples. Let $\mathcal{D}^{e}_{failed} \subset \mathcal{D}_{test}$ be the set of test samples where the branch e has eventually failed in the cascade but not as part of the initial contingency. Say, the true failure step of a branch $e \in E$ in sample $d \in \mathcal{D}^{e}_{failed}$ is f^{d}_{e} and the predicted state is $\widehat{f}^{d}_{e}$, then

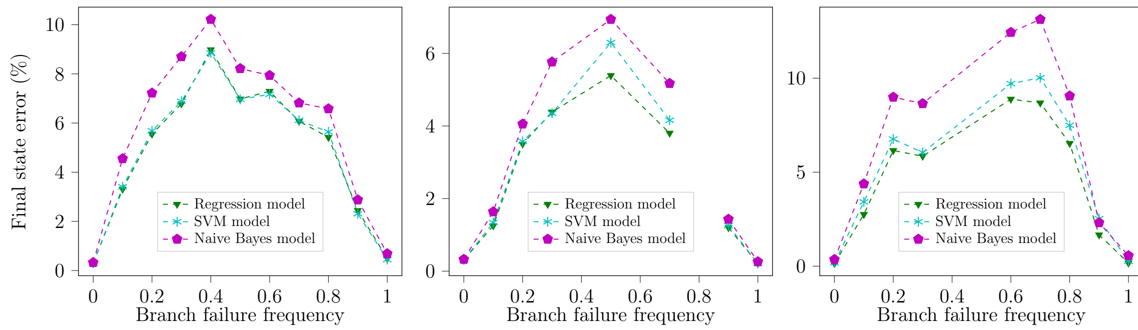
$$l_{failure-step,e} = \frac{1}{|\mathcal{D}^{e}_{failed}|} \sum_{d \in \mathcal{D}^{e}_{failed}} |\widehat{f}^{d}_{e} - f^{d}_{e}|. \tag{6.3}$$

6.2 Results for the Machine Learning Models

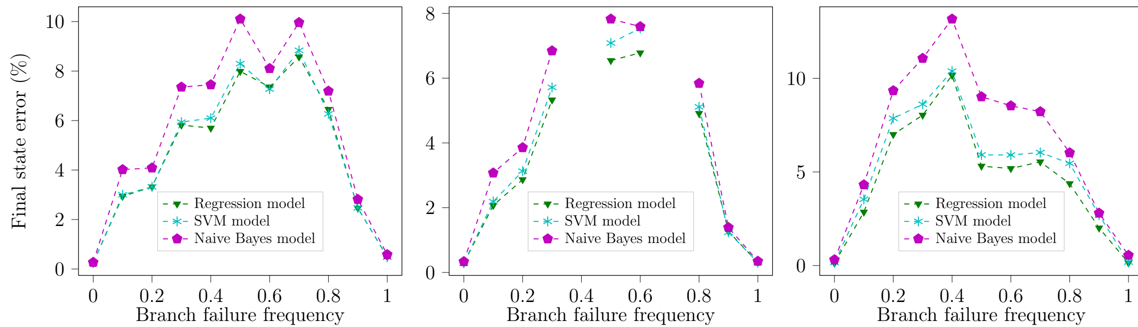
In this section, we present the performance of proposed machine learning models like regression, SVM, and naive Bayes models using the metrics defined in Section 6.1.

6.2.1 Branch Final State Error Rate

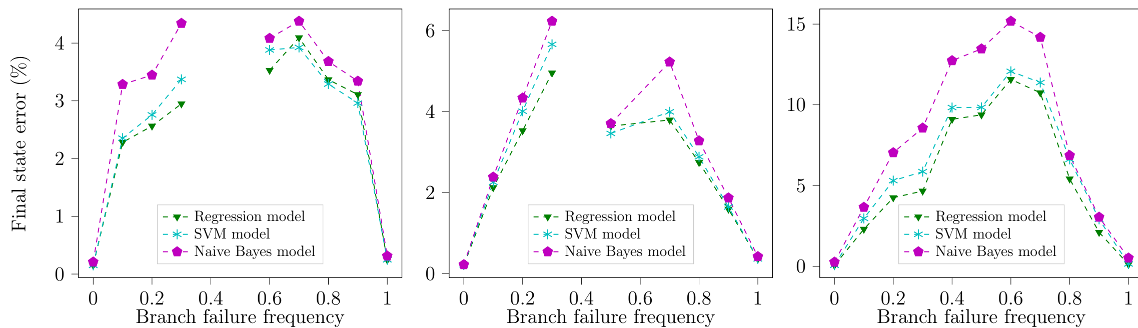
Fig. 6-1 shows the branch final state error rates $l_{state,e}$ for various instances of the



(a) Load scaling = 1.40 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.



(b) Load scaling = 1.50 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.



(c) Load scaling = 1.90 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.

Figure 6-1: Branch-average error in prediction of final state $l_{state,e}$ for IEEE89, IEEE118, and IEEE1354 (left to right) for various load scaling plotted against $l_{freq,e}$.

regression model, the SVM model, and the naive Bayes model. The figure contains three sub-figures (a), (b), and (c) showing the performance of different model instances trained and tested on load scaling values of 1.40, 1.50, and 1.90 respectively. Further, each of these sub-figures present three model instances trained and tested on IEEE89, IEEE118, and IEEE1354 (left to right) datasets. The error rates of branches are plotted as a function of the branch failure frequencies $l_{freq,e}$ and the branches having similar branch failure frequencies are grouped together and their bin averages are plotted for clarity.

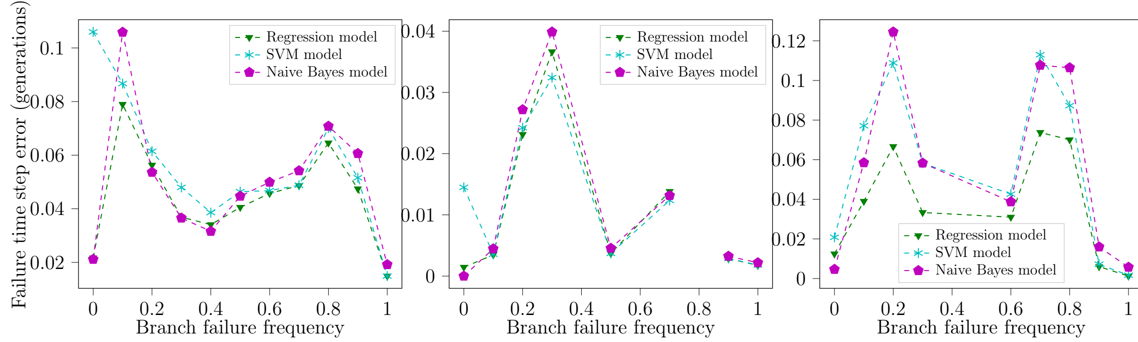
We can see that the error rate is highest for those branches with failure frequencies close to 0.5. This is expected since these branches are harder to predict because of their outcome's high variance. Further, the final state prediction error is below 10%, 8%, and 15% in all the instances for IEEE89, IEEE118, and IEEE154 respectively. In these plots, the highest final state prediction error is seen 15%, which is seen by the naive Bayes model for the IEEE1354 system at a load scaling value of 1.90 and a branch failure frequency of 0.6. Finally, the regression model outperforms the SVM and the naive Bayes models in all the cases.

6.2.2 Branch Failure Step Error Rate

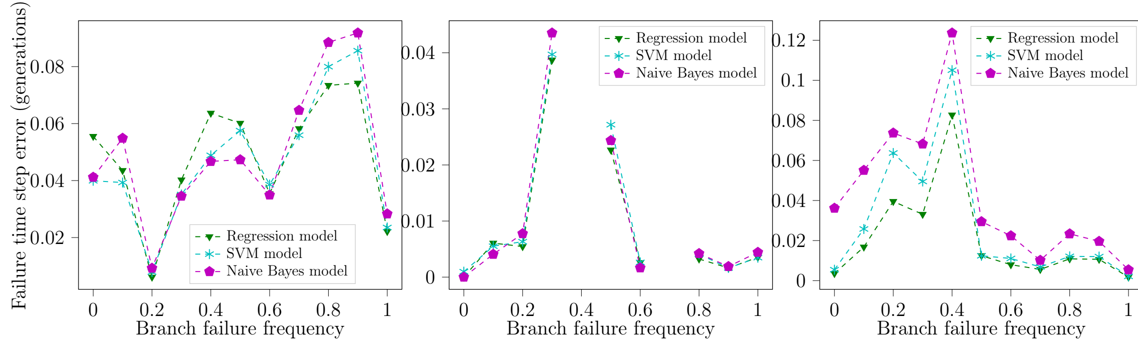
Fig. 6-2 shows the branch failure step error rates $l_{failure-step,e}$ of various instances of the regression model, the SVM model, and the naive Bayes model trained and tested on load scaling values of 1.40, 1.50, and 1.90. Each of these sub-figures show three plots corresponding to three model instances trained and tested on IEEE89, IEEE118, and IEEE1354 (left to right) datasets. The error rates of branches are plotted as a function of the branch failure frequencies $l_{freq,e}$ and the branches having similar branch failure frequencies are grouped together and their bin averages are plotted for clarity.

We can see that the error in failure time step prediction is in the order of 0.1 time steps (or generations) on an average whenever the branch is failed. The highest error in failure step prediction is 0.12, which occurs in the case of IEEE1354 system at a load scaling of 1.40 for branches with failure frequency 0.2. Finally, the regression

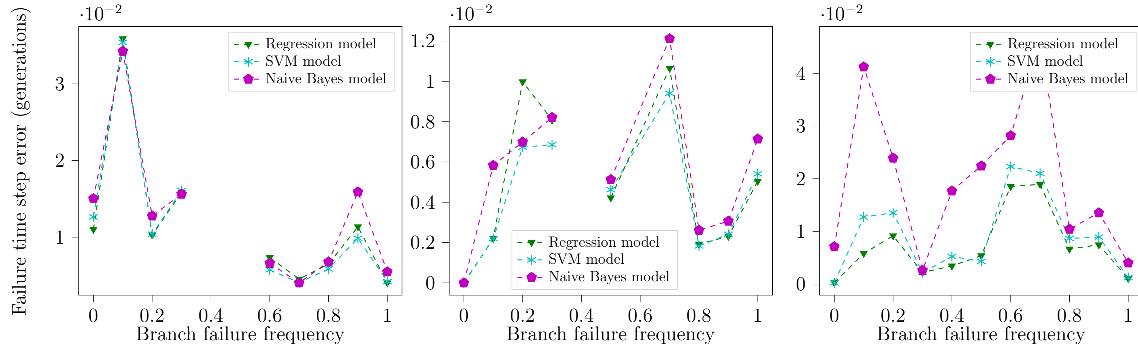
model has the best performance in most except some cases, where it is outperformed by the SVM model.



(a) Load scaling = 1.40 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.



(b) Load scaling = 1.50 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.



(c) Load scaling = 1.90 for IEEE89, IEEE118, and IEEE1354 (left to right) datasets.

Figure 6-2: Branch-average error in prediction of failure steps $l_{failure-step,e}$ for IEEE89, IEEE118, and IEEE1354 (left to right) for various load scaling plotted against $l_{freq,e}$.

6.3 Results for the GNN Models

In this section, we present the performance of the proposed GNN model using the metrics defined in Section 6.1.

6.3.1 Branch Final State Error Rate

Fig. 6-3 shows the final state error rate $l_{state,e}$ for two instances of the GNN model trained and tested on IEEE89 (left) and IEEE118 (right) datasets. The error rates are averaged over all test samples, containing random initial contingencies and random load scaling values, and plotted against failure frequency $l_{freq,e}$. Also, the failure frequencies close to each other are grouped together and their bin averages are plotted.

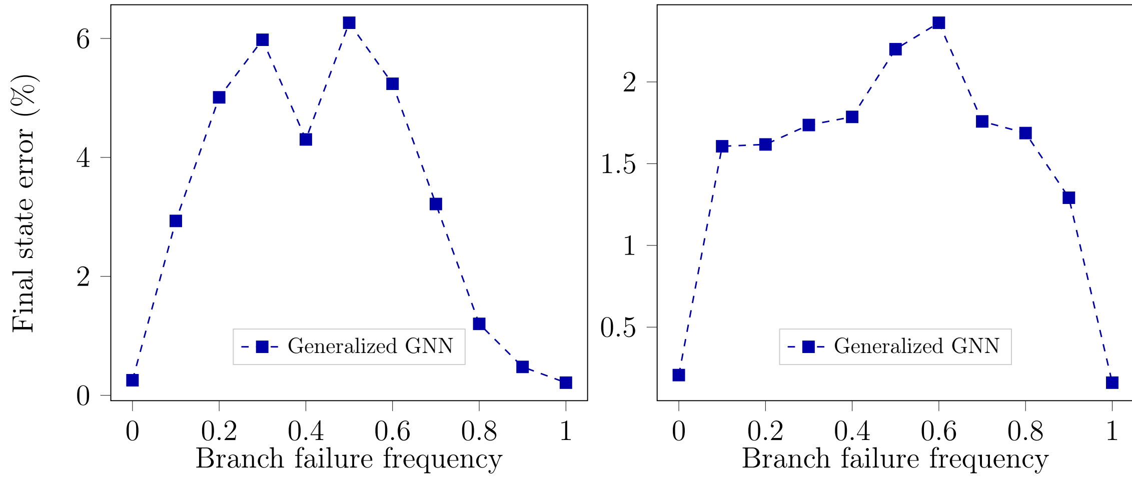
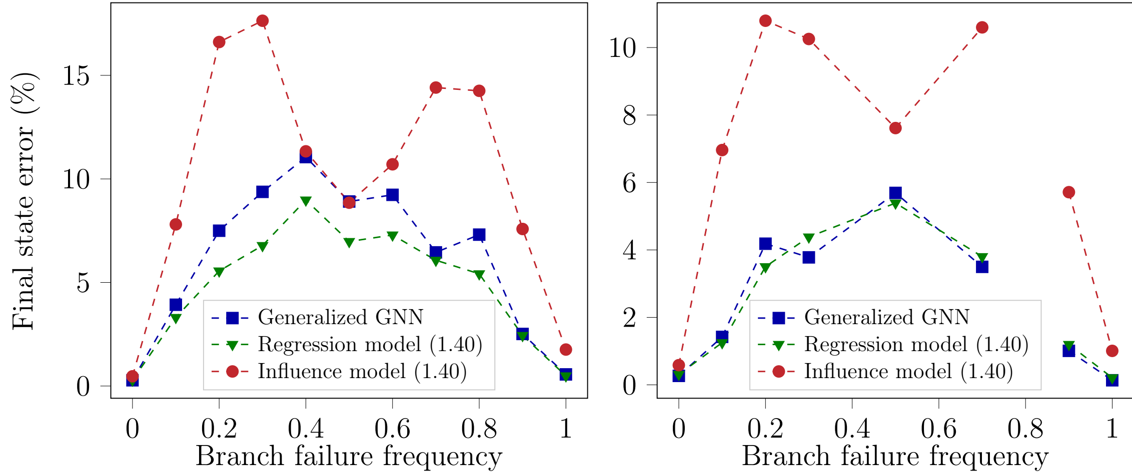


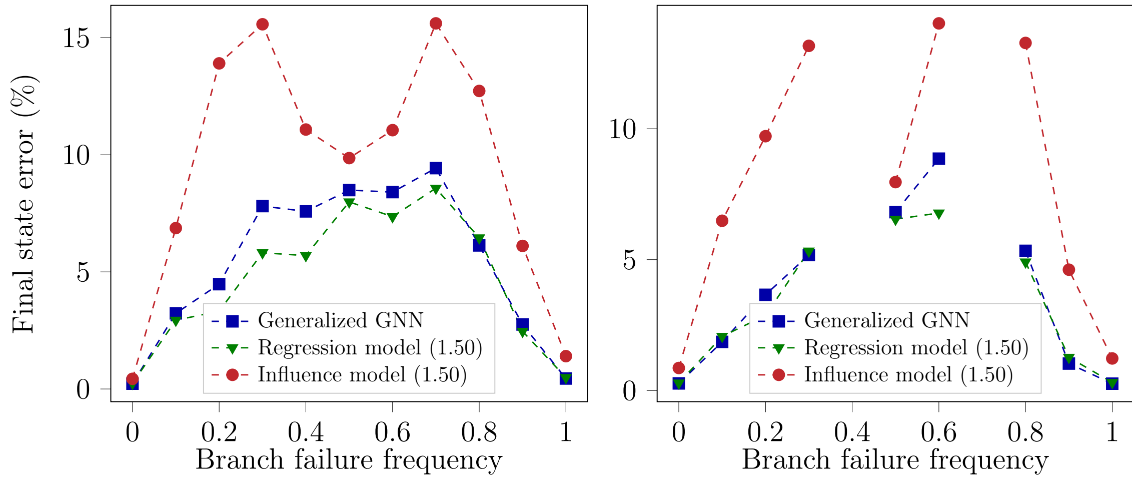
Figure 6-3: Error in prediction of final state $l_{state,e}$ for IEEE89 (left) and IEEE118 (right) averaged over all scaling values $[1, 2]$ against branch failure frequencies $l_{freq,e}$.

As can be seen, the average error rate in final state prediction by the GNN model is below 6% and 2.5% at all branches for IEEE89 and IEEE118 systems respectively. It can be observed that the error rate is higher for the branches whose failure frequencies are close to 0.5. This is expected since these branches are harder to predict because of their outcome's high variance.

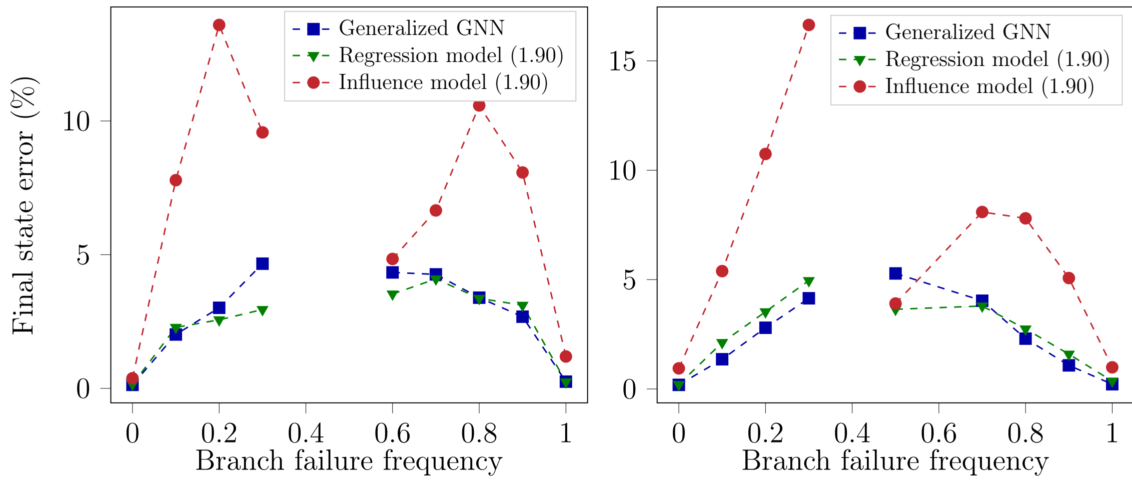
Further, the error plot in Fig. 6-3 is generated by averaging over random scaling values in $[1, 2]$, which demonstrates that the GNN model can be generalized over variable load profiles. Now, in order to benchmark the performance, Fig. 6-4 plots the final state error rates of the same GNN model, tested on load scaling values of 1.40, 1.50, and 1.90, against different instances of the regression and influence models built specifically for load scaling values of 1.40, 1.50, and 1.90. Note again that the GNN model works on any load scaling values in $[1, 2]$ as shown in Fig. 6-5, where the metric has been averaged over random scaling values.



(a) Load scaling = 1.40 for IEEE89 (left) and IEEE118 (right).



(b) Load scaling = 1.50 for IEEE89 (left) and IEEE118 (right).



(c) Load scaling = 1.90 for IEEE89 (left) and IEEE118 (right)s.

Figure 6-4: Branch-average error in prediction of final state $l_{state,e}$ for IEEE89 (left) and IEEE118 (right) for various load scaling against branch failure frequencies $l_{freq,e}$.

As can be seen, the GNN model, despite being generic over load scaling values, outperforms the load-specific instances of the influence model at almost all branches, beating the influence model by almost 10% in some cases. Further, the regression model performs better by around 3% than the GNN model at most of the branches. However, at some load scaling values in the IEEE118 case, the GNN model outperforms the regression model.

6.3.2 Branch Failure Step Error Rate

Fig. 6-5 shows the branch failure step error rate $l_{failure-step,e}$ for two instances of the GNN model trained and tested on IEEE89 (left) and IEEE118 (right) systems. The error rates are averaged over all test samples, containing random initial contingencies and random load scaling values, and plotted as a function of branch failure frequencies $l_{freq,e}$. The failure frequencies close to each other are grouped together and their bin averages are plotted.

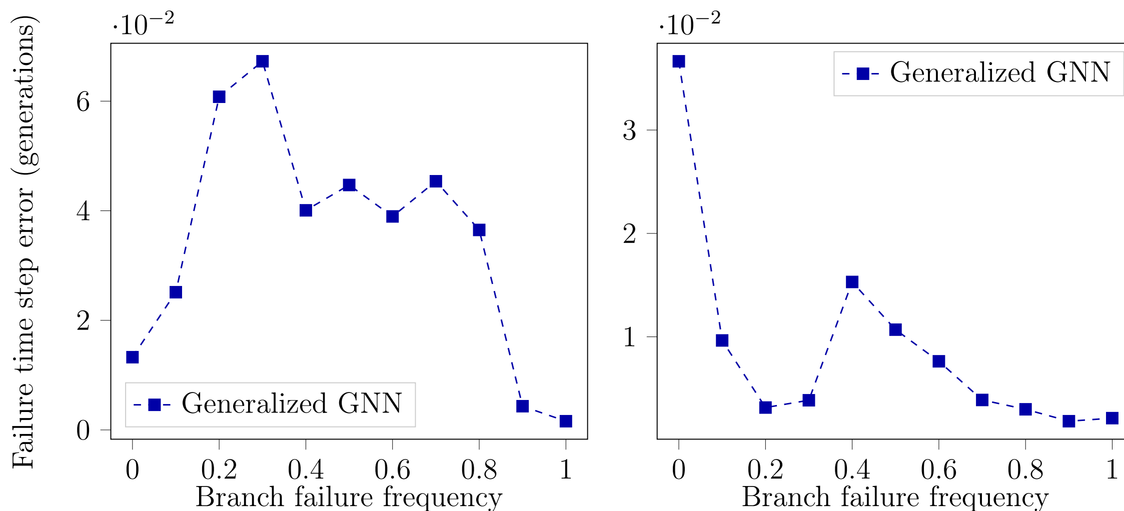
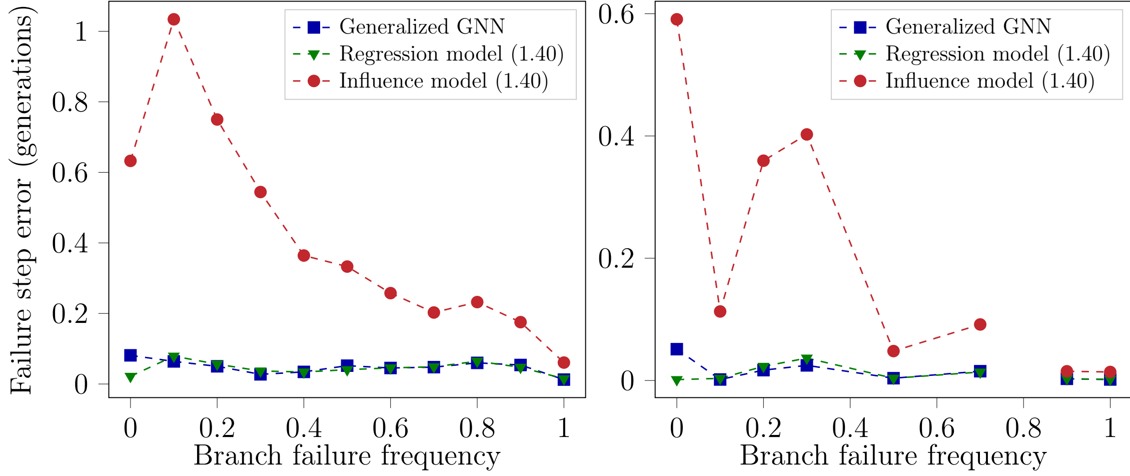


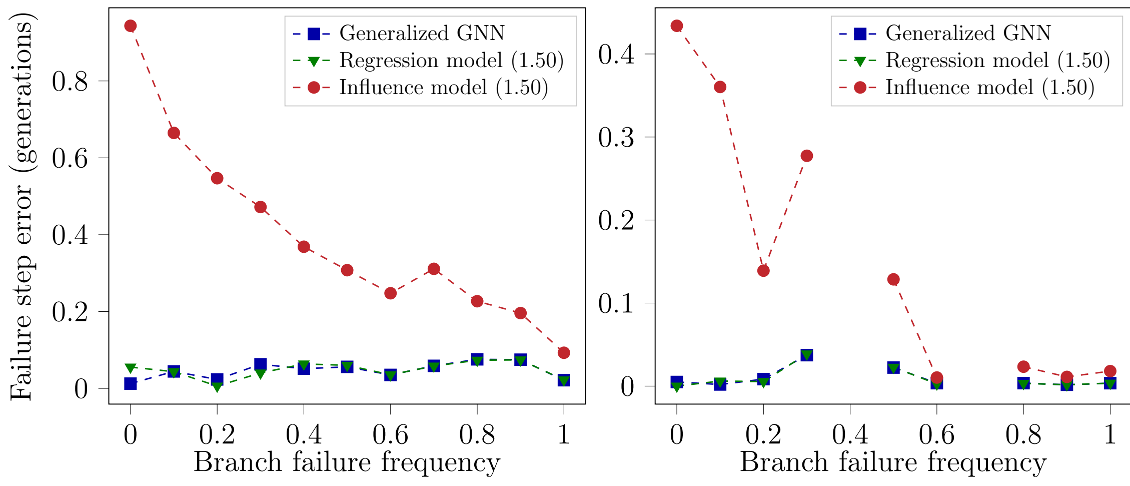
Figure 6-5: Error in prediction of failure time steps $l_{failure-step,e}$ for IEEE89 (left) and IEEE118 (right) averaged over all scaling $[1, 2]$ against failure frequencies $l_{freq,e}$.

As seen in the plot, the failure step error rate is in the order of 0.01 time steps. The significantly low error performance when averaged over random scaling values demonstrates the generalization capabilities of the GNN model.

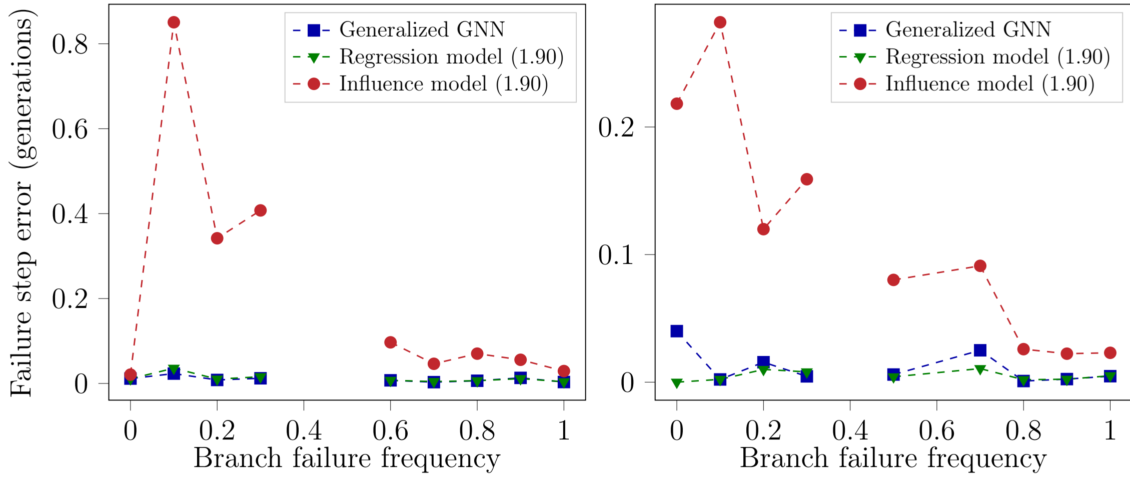
Similar to the previous metric, to benchmark the performance, Fig. 6-6 plots the



(a) Load scaling = 1.40 for IEEE89 (left) and IEEE118 (right).



(b) Load scaling = 1.50 for IEEE89 (left) and IEEE118 (right).



(c) Load scaling = 1.90 for IEEE89 (left) and IEEE118 (right).

Figure 6-6: Branch-average error in prediction of failure steps $l_{failure-step,e}$ for IEEE89 (left) and IEEE118 (right) for various scaling against branch failure frequencies $l_{freq,e}$.

branch failure step error rates of the same GNN model, tested on load scaling values of 1.40, 1.50, and 1.90, and compares it to different instances of the regression and the influence models built specifically for load scaling values of 1.40, 1.50, and 1.90. Note again that the GNN model works on any load scaling values in $[1, 2]$ as shown in Fig. 6-5, where the metric has been averaged over random scaling values.

The failure step prediction performance of the GNN model, despite being generic to all load scaling values, is very close to that of the load-specific regression models. Additionally, the performance of the generic GNN model is significantly better than the load-specific influence models. In the influence model, when doing state prediction in a step-by-step manner, the errors that occur in initial steps propagate to later steps, thereby accumulating to a high final error. This is completely avoided by the GNN model since it predicts the failure steps directly instead of predicting network states at each time steps individually. We believe this causes the GNN to outperform the influence models in these metrics.

Chapter 7

Results: Runtime Analysis

We perform a runtime analysis to demonstrate how the GNN and the machine learning models can harvest the power of GPUs to predict cascade sequences much faster than the flow-based simulation methods. We run cascade predictions on 11,000 test samples with the CFS oracle, the influence model, the GNN model, and the machine learning models. Table 7.1 presents the resulting runtime, in seconds per 1000 runs. The CFS oracle cannot be run on a GPU, hence it was tested in MATLAB 2019a on a Intel(R) Core(TM) i9-7920X CPU@2.90GHz processor with 128GB of installed memory. The DC power flow calculation in the CFS oracle was done by the MATPOWER toolbox. Further, the influence model, the machine learning models, and the GNN models were tested on an NVIDIA GeForce RTX 2080 Ti GPU with 11GB of total memory.

Table 7.1: Prediction time in seconds per 1000 samples

	CFS oracle	Influence	GNN	Naive Bayes	SVM	Regression
IEEE89	24.18	2.34	0.53	0.27	0.15	0.18
IEEE118	62.54	2.03	0.28	0.22	0.12	0.13

It can be seen that the time taken by the influence models, the machine learning models, and the GNN models are significantly lower than the CFS oracle. In the influence model, the matrix multiplications can be sped up using a GPU. However, because of its step-by-step prediction nature, each cascade prediction lasts for a vari-

able number of steps. Hence, we cannot run multiple predictions simultaneously with the influence model. Meanwhile, the GNN model can be fully parallelized allowing us to run thousands of cascade sequence predictions simultaneously. This makes predictions with the GNN model almost five times faster than the influence model. Further, the machine learning models are even faster as these simple models do not need to keep track of the graph topology like the GNN model. Among the three machine learning models, it can be seen that the regression model is the fastest.

Chapter 8

Conclusion

We considered the problem of predicting the failure cascade sequence due to branch failures given the initial contingency, the power injection values, and the grid topology. We first proposed several flow-free models based on machine learning techniques like support vector machines, naive Bayes classifiers, and logistic regression. These models predict the gird states at every generation of a cascade, without requiring power flow calculations. We also proposed a flow-free graph neural network model that can be generalized over variable load profiles. We trained the proposed models using simulated data and evaluated them at various levels of prediction granularity. We showed that the machine learning models have lower error performance than the influence model. We also showed that the GNN model, in addition to being generic over randomly scaled loading values, outperforms the influence models that were built specifically for their corresponding loading profiles. Finally, we presented a runtime analysis that showed the models' ability to harvest the power of GPUs and reduce the computational time by almost two orders of magnitude compared to the flow-based cascading failure simulator.

8.1 Future Directions

Some of the potential future directions to this work are listed below.

1. In the results, we train our models using data generated from DC power flow

calculations. In future, data samples generated from AC power flow can be used. We believe that the proposed models will definitely be faster than the AC cascade sequence generator as AC solvers are much slower than the DC solvers.

2. We can extend the data pool to large scale buses like IEEE 2383 bus system. This will require the GNN model to be more efficient so as to avoid GPU memory overflow issues. This can be done by using a more efficient attention mechanism technique like the graph attention networks [53].

3. The GNN model we proposed is limited to a specific graph topology. This is because the input dimension in the attention stage of the proposed model is fixed and is equal to the number of branches in the model. This limits our model to be trained over data samples generated from a specific topology. A natural extension to the proposed GNN model is to make the model generic over any given topology. This can again be done by changing the attention stage of the model using concepts from graph attention networks [53].

4. A study of interactions between various branches can also be conducted. One possible way to do this from our GNN model is to observe the attention coefficients. These coefficients can potentially signal the extent of correlation between any two branches in the system.

Bibliography

[1] J. Kim, "Increasing Power Outages Don't Hit Everyone Equally," 2023. Scientific American. Available at: https://www.scientificamerican.com/article/increasing-power-outages-dont-hit-everyone-equally1/#:~:text=Between%202013%20and%202021%2C%20the,events%20per%20customer%20per%20year (accessed: 08 August 2023).

[2] Climatecentral.org, "Surging Weather-related Power Outages," 2022. Online. Available at: https://www.climatecentral.org/climate-matters/surging-weather-related-power-outages (accessed: 08 August, 2023).

[3] J. Jiménez and R. Carballo, "Cleanup Begins After Severe Storms Tear Through Eastern U.S.," 2023. New York Times. Available at: https://www.nytimes.com/2023/08/08/us/us-severe-storms-damage.html (accessed: 08 August, 2023).

[4] C. Clifford, "Why america's outdated energy grid is a climate problem," 2023. CNBC. Available at: https://www.cnbc.com/2023/02/17/why-americas-outdated-energy-grid-is-a-climate-problem.html (accessed: 08 August 2023).

[5] K. H. LaCommare, J. H. Eto, L. N. Dunn, and M. D. Sohn, "Improving the estimated cost of sustained power interruptions to electricity customers," Energy, vol. 153. Elsevier BV, pp. 1038–1047, Jun. 2018. doi: 10.1016/j.energy.2018.04.082.

[6] B. Stone Jr. et al., "How Blackouts during Heat Waves Amplify Mortality and Morbidity Risk," Environmental Science & Technology, vol. 57, no. 22. American Chemical Society (ACS), pp. 8245–8255, May 23, 2023. doi: 10.1021/acs.est.2c09588.

[7] N. Popovich and B. Plumer, "How electrification became a major tool for fighting climate change," 2023. New York Times. Available at: https://www.nytimes.com/interactive/2023/04/14/climate/electric-car-heater-everything.html (accessed: 08 August 2023).

[8] Vaiman et al., "Risk Assessment of Cascading Outages: Methodologies and Challenges," in IEEE Transactions on Power Systems, vol. 27, no. 2, pp. 631-641, May 2012, doi: 10.1109/TPWRS.2011.2177868.

[9] I. Dobson, "Estimating the Propagation and Extent of Cascading Line Outages From Utility Data With a Branching Process," in IEEE Transactions on Power Systems, vol. 27, no. 4, pp. 2146-2155, Nov. 2012, doi: 10.1109/TPWRS.2012.2190112.

[10] H. Ren and I. Dobson, "Using Transmission Line Outage Data to Estimate Cascading Failure Propagation in an Electric Power System," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 55, no. 9, pp. 927-931, Sept. 2008, doi: 10.1109/TCSII.2008.924365.

[11] P. D. H. Hines, I. Dobson and P. Rezaei, "Cascading Power Outages Propagate Locally in an Influence Graph That is Not the Actual Grid Topology," in IEEE Transactions on Power Systems, vol. 32, no. 2, pp. 958-967, March 2017, doi: 10.1109/TPWRS.2016.2578259.

[12] A. Bernstein, D. Bienstock, D. Hay, M. Uzunoglu and G. Zussman, "Power grid vulnerability to geographically correlated failures — Analysis and control implications," IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, Toronto, ON, Canada, 2014, pp. 2634-2642, doi: 10.1109/INFOCOM.2014.6848211.

[13] S. Soltan, D. Mazauric and G. Zussman, "Analysis of Failures in Power Grids," in IEEE Transactions on Control of Network Systems, vol. 4, no. 2, pp. 288-300, June 2017, doi: 10.1109/TCNS.2015.2498464.

[14] H. Cetinay, S. Soltan, F. A. Kuipers, G. Zussman and P. Van Mieghem, "Comparing the Effects of Failures in Power Grids Under the AC and DC Power Flow Models," in IEEE Transactions on Network Science and Engineering, vol. 5, no. 4, pp. 301-312, 1 Oct.-Dec. 2018, doi: 10.1109/TNSE.2017.2763746.

[15] M. J. Eppstein and P. D. H. Hines, "A "Random Chemistry" Algorithm for Identifying Collections of Multiple Contingencies That Initiate Cascading Failure," in IEEE Transactions on Power Systems, vol. 27, no. 3, pp. 1698-1705, Aug. 2012, doi: 10.1109/TPWRS.2012.2183624.

[16] P. Rezaei, P. D. H. Hines and M. J. Eppstein, "Estimating Cascading Failure Risk With Random Chemistry," in IEEE Transactions on Power Systems, vol. 30, no. 5, pp. 2726-2735, Sept. 2015, doi: 10.1109/TPWRS.2014.2361735.

[17] X. Zhang, C. Zhan and C. K. Tse, "Modeling the Dynamics of Cascading Failures in Power Systems," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 7, no. 2, pp. 192-204, June 2017, doi: 10.1109/JETCAS.2017.2671354.

[18] J. Qi, J. Wang and K. Sun, "Efficient Estimation of Component Interactions for Cascading Failure Analysis by EM Algorithm," in IEEE Transactions on Power Systems, vol. 33, no. 3, pp. 3153-3161, May 2018, doi: 10.1109/TPWRS.2017.2764041.

[19] J. Qi, K. Sun and S. Mei, “An Interaction Model for Simulation and Mitigation of Cascading Failures,” in IEEE Transactions on Power Systems, vol. 30, no. 2, pp. 804-819, March 2015, doi: 10.1109/TPWRS.2014.2337284.

[20] X. Wu, D. Wu and E. Modiano, “Predicting Failure Cascades in Large Scale Power Systems via the Influence Model Framework,” in IEEE Transactions on Power Systems, vol. 36, no. 5, pp. 4778-4790, Sept. 2021, doi: 10.1109/TPWRS.2021.3068409.

[21] J. Xie, I. Alvarez-Fernandez and W. Sun, “A Review of Machine Learning Applications in Power System Resilience,” 2020 IEEE Power & Energy Society General Meeting (PESGM), Montreal, QC, Canada, 2020, pp. 1-5, doi: 10.1109/PESGM41954.2020.9282137.

[22] L. Duchesne, E. Karangelos and L. Wehenkel, “Recent Developments in Machine Learning for Energy Systems Reliability Management,” in Proceedings of the IEEE, vol. 108, no. 9, pp. 1656-1676, Sept. 2020, doi: 10.1109/JPROC.2020.2988715.

[23] N. M. Sami and M. Naeini, “Machine Learning Applications in Cascading Failure Analysis in Power Systems: A Review.” arXiv, 2023. doi: 10.48550/ARXIV.2305.19390.

[24] Y. Yang, Z. Yang, J. Yu, B. Zhang, Y. Zhang, and H. Yu, “Fast Calculation of Probabilistic Power Flow: A Model-Based Deep Learning Approach,” IEEE Transactions on Smart Grid, vol. 11, no. 3. Institute of Electrical and Electronics Engineers (IEEE), pp. 2235–2244, May 2020. doi: 10.1109/tsg.2019.2950115.

[25] Y. Du, F. Li, J. Li and T. Zheng, “Achieving 100x Acceleration for N-1 Contingency Screening With Uncertain Scenarios Using Deep Convolutional Neural Network,” in IEEE Transactions on Power Systems, vol. 34, no. 4, pp. 3303-3305, July 2019, doi: 10.1109/TPWRS.2019.2914860.

[26] S. Gupta, R. Kambli, S. Wagh and F. Kazi, “Support-Vector-Machine-Based Proactive Cascade Prediction in Smart Grid Using Probabilistic Framework,” in IEEE Transactions on Industrial Electronics, vol. 62, no. 4, pp. 2478-2486, April 2015, doi: 10.1109/TIE.2014.2361493.

[27] R. A. Shuvro, P. Das, M. M. Hayat and M. Talukder, “Predicting Cascading Failures in Power Grids using Machine Learning Algorithms,” 2019 North American Power Symposium (NAPS), Wichita, KS, USA, 2019, pp. 1-6, doi: 10.1109/NAPS46351.2019.9000379.

[28] H. Zhang, T. Ding, J. Qi, W. Wei, J. P. S. Catalão and M. Shahidehpour, “Model and Data Driven Machine Learning Approach for Analyzing the Vulnerability to Cascading Outages With Random Initial States in Power Systems,” in IEEE Transactions on Automation Science and Engineering, 2022, doi: 10.1109/TASE.2022.3204273.

[29] R. Pi, Y. Cai, Y. Li and Y. Cao, "Machine Learning Based on Bayes Networks to Predict the Cascading Failure Propagation," in IEEE Access, vol. 6, pp. 44815-44823, 2018, doi: 10.1109/ACCESS.2018.2858838.

[30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, "The Graph Neural Network Model," in IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61-80, Jan. 2009, doi: 10.1109/TNN.2008.2005605.

[31] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks." arXiv, 2016. doi: 10.48550/ARXIV.1609.02907.

[32] B. Donon, B. Donnot, I. Guyon, and A. Marot, "Graph Neural Solver for Power Systems," 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, Jul. 2019. doi: 10.1109/ijcnn.2019.8851855.

[33] D. Wang, K. Zheng, Q. Chen, G. Luo, and X. Zhang, "Probabilistic Power Flow Solution with Graph Convolutional Network," 2020 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe). IEEE, Oct. 26, 2020. doi: 10.1109/isgt-europe47291.2020.9248786.

[34] J. B. Hansen, S. N. Anfinsen, and F. M. Bianchi, "Power Flow Balancing with Decentralized Graph Neural Networks," arXiv, 2021, doi: 10.48550/ARXIV.2111.02169.

[35] B. Donon, R. Clément, B. Donnot, A. Marot, I. Guyon, and M. Schoenauer, "Neural networks for power flow: Graph neural solver," Electric Power Systems Research, vol. 189. Elsevier BV, p. 106547, Dec. 2020. doi: 10.1016/j.epsr.2020.106547.

[36] A. B. Jeddi and A. Shafieezadeh, "A Physics-Informed Graph Attention-based Approach for Power Flow Analysis," 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, Dec. 2021. doi: 10.1109/icmla52953.2021.00261.

[37] D. Owerko, F. Gama and A. Ribeiro, "Optimal Power Flow Using Graph Neural Networks," ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Barcelona, Spain, 2020, pp. 5930-5934, doi: 10.1109/ICASSP40776.2020.9053140.

[38] T. Pham and X. Li, "Reduced Optimal Power Flow Using Graph Neural Network," 2022 North American Power Symposium (NAPS), Salt Lake City, UT, USA, 2022, pp. 1-6, doi: 10.1109/NAPS56150.2022.10012256.

[39] A. Varbella, B. Gjorgiev, and G. Sansavini, "Geometric deep learning for online prediction of cascading failures in power grids," Reliability Engineering and System Safety, vol. 237. Elsevier BV, p. 109341, Sep. 2023. doi: 10.1016/j.ress.2023.109341.

[40] S. Gupta, F. Kazi, S. Wagh, and R. Kambli, "Neural Network Based Early Warning System for an Emerging Blackout in Smart Grid Power Networks," Intelligent Distributed Computing. Springer International Publishing, pp. 173–183, 2015. doi: 10.1007/978-3-319-11227-5_16.

[41] C. Kim, K. Kim, P. Balaprakash, and M. Anitescu, "Graph Convolutional Neural Networks for Optimal Load Shedding under Line Contingency," 2019 IEEE Power & Energy Society General Meeting (PESGM). IEEE, Aug. 2019. doi: 10.1109/pesgm40551.2019.8973468.

[42] Y. Zhu, Y. Zhou, W. Wei, and L. Zhang, "Real-Time Cascading Failure Risk Evaluation With High Penetration of Renewable Energy Based on a Graph Convolutional Network," IEEE Transactions on Power Systems. Institute of Electrical and Electronics Engineers (IEEE), pp. 1–12, 2022. doi: 10.1109/tpwrs.2022.3213800.

[43] S. Lonapalawong, C. Chen, C. Wang, and W. Chen, "Interpreting the vulnerability of power systems in cascading failures using multi-graph convolutional networks," Frontiers of Information Technology and Electronic Engineering, vol. 23, no. 12. Zhejiang University Press, pp. 1848–1861, Jun. 20, 2022. doi: 10.1631/fitee.2200035.

[44] B. Jhun, H. Choi, Y. Lee, J. Lee, C. H. Kim, and B. Kahng, "Prediction and mitigation of nonlocal cascading failures using graph neural networks," Chaos: An Interdisciplinary Journal of Nonlinear Science, vol. 33, no. 1. AIP Publishing, p. 013115, Jan. 2023. doi: 10.1063/5.0107420.

[45] Y. Zhu, Y. Zhou, W. Wei, and N. Wang, "Cascading Failure Analysis Based on a Physics-Informed Graph Neural Network," IEEE Transactions on Power Systems. Institute of Electrical and Electronics Engineers (IEEE), pp. 1–10, 2022. doi: 10.1109/tpwrs.2022.3205043.

[46] H. Zhang, "The Optimality of Naive Bayes," in Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004), Miami Beach, Florida, USA, 2004.

[47] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," Statistics and Computing, vol. 14, no. 3. Springer Science and Business Media LLC, pp. 199–222, Aug. 2004. doi: 10.1023/b:stco.0000035301.49549.88.

[48] C. M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics). Berlin, Heidelberg: Springer-Verlag, 2006.

[49] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization." arXiv, 2014. doi: 10.48550/ARXIV.1412.6980.

[50] R. D. Zimmerman, C. E. Murillo-Sanchez, and R. J. Thomas, "Matpower: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," Power Systems, IEEE Transactions on, vol. 26, no. 1, pp. 12–19, Feb. 2011. doi: 10.1109/TPWRS.2010.2051168.

[51] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” arXiv, 2019. doi: 10.48550/ARXIV.1912.01703.

[52] F. Pedregosa et al., “Scikit-Learn: Machine Learning in Python”, J. Mach. Learn. Res., vol. 12, no. null, pp. 2825–2830, Nov. 2011.

[53] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2017). “Graph Attention Networks” (Version 3). arXiv. https://doi.org/10.48550/ARXIV.1710.10903.