VoltNet: A deep learning approach for on-chip voltage emergency prediction

Yi Shen

Master of Engineering

Department of Electrical and Computer Engineering

McGill University

Montreal,Quebec

2019-8-14

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering in Electrical Engineering

DEDICATION

To my parents.

ACKNOWLEDGEMENTS

I would like to appreciate the help from my supervisor Prof. Roni Khazaka. He has provided great insight, guidance and patience. I give special thanks to Prof. Brett Meyer for his invaluable assistance in Linux. I thanks Marco Kassis and Karanvir Sidhu for their accompany and help through my journey though Master of Engineering. Last but not the least, I thank open source community for their selfless contributions. This thesis would not complete without the help from Scikit-learn, Keras, TensorFlow, PARSEC, Numpy, Gem5, McPAT, VoltSpot, ArchFP and VScode

ABSTRACT

Voltage noise on the microprocessor power delivery network can be very dangerous. When the voltage exceeds a certain threshold, it threatens the correctness of the microprocessor's operation. Current methods aggressively lower the voltage of power supply so that the voltage emergency may never happen. However, the downside is that the microprocessors are not working at their maximum capability. If voltage emergencies could be predicted, then the potential of the microprocessors can be fully utilized. Some linear regression-based algorithms are proposed in the literature. The thesis proposes two related linear-regression algorithms along with a new deep-learning model to predict emergencies. The new deep-learning model yields promising results.

ABRÉGÉ

Le bruit de tension sur le réseau d'alimentation du microprocesseur peut être très dangereux. Lorsque la tension dépasse un certain seuil, elle menace l'exactitude du fonctionnement du microprocesseur. Les méthodes actuelles abaissent de manière agressive la tension d'alimentation de sorte que l'urgence de tension ne se produise jamais. Cependant, l'inconvénient est que les microprocesseurs ne fonctionnent pas à leur capacité maximale. Si des urgences de tension pouvaient être prévues, alors le potentiel des microprocesseurs peut être pleinement utilisé. Certains algorithmes basés sur la régression linéaire sont proposés dans la littérature. La thèse propose deux algorithmes de régression linéaire similaires ainsi qu'un nouveau modèle d'apprentissage en profondeur pour prédire les urgences. Le nouveau modèle d'apprentissage en profondeur donne des résultats prometteurs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Symbols

$\hat{\beta}^k$	kth column of matrix B
t	Constraint Parameter for the original group Lasso
T	Importance factor threshold for the original group Lasso
λ	Lagrange multiplier for the new group Lasso
D	data set
S	Candidate node set
C	Selected ndoe set
σ	standard devicaton or sigmoid function

CHAPTER 1
Introduction

1.1 Motivation for the Voltage Emergency Prediction

As the technology node pushes closer to the physical limit, power supply noise is a major threat to the accuracy and reliability of microprocessors' (CPU) operations. The power supply noise deteriorates CPU performance in many undesirable ways. Two glaring issues are its slowing effect on microprocessors' clock frequencies [3] and infringement on operation correctness [32]. One effective counter to noise-induced problems is calibrated power delivery network (PDN), which includes optimization of controlled collapse chip connect (C4) pads arrangement [9], increased layers of PDN [40], 3-dimensional stacking circuit [27], etc.

However, optimization of the physical PDN design would either require an enormous amount of engineering effort like [40, 27] or increased space for power-ground (P/G) pads which reduces the availability of signal I/O pads like [9]. Most importantly, even the most optimized PDN design cannot solely guarantee the security of CPU operations. A safety net is much needed.

One way to ensure the correctness of operations is a hybrid method of dynamic margin adaptation (DMA) and noise-induced error recovery [39]. DMA reduces the frequency and increases the voltage in an impulse manner to save operations from being corrupted. The impulse will have its delay, which sometimes will cause failure to save. Combining DMA with noise-induced error recovery is a smart way to overcome its drawback. Once DMA failed, the recovery will rollback the microprocessor to its previous state (usually 30 microprocessor cycles ago) and rerun the operations. The recovery will slow down the microprocessor, but the correctness will be guaranteed.

This thesis offers ways to predict voltage emergency so that the impulse from DMA can be triggered earlier to reduce the occurrence of recoveries. Thus, it can improve the performance of microprocessors.

1.2 Introduction to the Prediction Problem

Like any other prediction problems, a good predictor needs good input. The given input should contain enough information to support our confidence in predicted events. For CPU voltage emergency problem, we choose global CPU voltage profile as our input space. Voltage varies at different points on CPU die. We take all voltages on the CPU die over time as our potential inputs to the algorithm. Voltages are collected by voltage sensors. Therefore, it is a sensor-based approach. Each sensor has its cost to build and the cost becomes unaffordable if there are too many sensors. Consequently, only a few locations on die could be monitored by sensors. Choosing the right locations is a key research interest, which we call "sensor placement problem", which can also be considered as feature engineering for later prediction algorithm. The emergency predictor is viewed as a "binary classification problem". The accuracy of such predictor largely depends on the quality of feature selection. We call each input to the binary classification model as a feature. Then a placement algorithm aims to find the fewest features that would maximize the classification accuracy. Binary classification problem concerns classifying given features into two categories; specifically, those will lead to an emergency and those that will not. Note that, we are only interested in emergencies in near future, which is defined in a few CPU cycles ahead because CPU will inevitably confront the emergency in indefinite future.

There are three sensor-placement algorithms proposed in this thesis, along with two prediction models. The three proposed placement algorithms are unique in their definition of the optimality for sensor placements. Quantitatively, three algorithms represent three objective functions for the placement evaluation. As a topic for later discussion, the most challenging part of the sensor placement problem is indeed finding a suitable objective function, as there is no explicit way to evaluate the selection quality.

The most effort of this thesis are devoted to discussing different sensor placement algorithms. It is because sensor placement plays a crucial role in the whole scheme, and it is more interesting than the binary classification part. Prediction models can be directly evaluated by the accuracy metric, but there is no established metric for evaluating sensor quality. In addition, there are only a few sensors that can be fabricated on the real CPU, and optimality of the placement algorithm is the bottleneck of extracting information from the CPU.

1.3 Thesis Contribution

In summary, this thesis makes four contributions to the literature. They include the integration of data generation infrastructure, algorithm improvements, and a new algorithm, which can be concluded as:

1. A set of interfaces with established simulation software, which provides convenient automation for generating PDN voltage data on CPU.
2. An improved version of the group Lasso sensor placement algorithm. Compared to the original one, the improvement focuses on decreasing the human experience required to apply the algorithm successfully.
3. An improved version of an eagle-eye sensor placement algorithm. It is an improvement focusing on the scope of the algorithm, which aims to combat sensor-clustering problem. The consequence is that the majority of the sensors will concentrate in a small area on the integrated circuit (IC).
4. A new deep learning algorithm that yields the placement during the training phase and predicts the emergency once training completes. This is a preliminary inquiry into the application of deep learning to PDN domain which is destined to have ample room for optimization. We hope this work could inspire more machine learning applications in the field.

1.4 Thesis Organization

The rest of this thesis is organized in the following way. In chapter 2, we illustrate necessary background knowledge for understanding the proposed algorithms. In chapter 3, we explore the most current researches in related fields. We also describe two sensor placement algorithms in great detail because they are related to two of our proposed algorithms. In chapter 4, we present a highly automatic data generation pipeline and its configuration for the data used in this thesis. In chapter 5, two regression-based algorithms are documented in detail. In chapter 6, a deep-learning based algorithm is introduced. In chapter 7, we present the evaluations of three algorithms.

CHAPTER 2
Background Information for Proposed Algorithms

Before we dive into the literature, we firstly introduce key theories and mathematical models which are necessary to understand the literature. In this chapter, we categorize all background information into sections corresponding to their applications. It appears that those backgrounds are listed as if they are separated topics. In the chapter 3 and chapter 5, we will demonstrate how to assemble those building blocks into algorithms.

2.1 Background for Eagle-Eye

Eagle-eye is a statistical maximization algorithm. It greedily maximize the probability of detecting an emergency by the placed sensor. In this section, we will review the notations for probability and some useful properties.

2.1.1 Probability and Notation

For eagle-eye algorithm, we take a frequentist perspective. Given an event Z, we define the the probability of this event as its relative frequency after a large number of trials. Formally, we define:

$$P(Z) = \frac{q_Z}{q_t} \tag{2.1}$$

where q_Z is the number of trails where event Z occurs and q_t is the total trails.

To further illustrate the notations used, consider two events Z_1 and Z_2. Denote $Z_1 \bigcup Z_2$ the event where either Z_1 or Z_2 happens. Denote $Z_1 \bigcap Z_2$ the event where Z_1 and Z_2 both happens. Naturally, $P(Z_1 \bigcup Z_2)$ and $P(Z_1 \bigcap Z_2)$ are the probabilities of corresponding events.

Finally, we denote the conditional probability of Z_1 given the occurrence of Z_2 as $P(Z_1|Z_2)$.

2.2 Background for Group Lasso Optimization

Group Lasso is an optimization-based algorithm. It takes advantages of the sparsity of Lasso regularization to select sensors. Our improved group Lasso algorithm also uses Lagrange multiplier and K-means clustering for optimization and automation.

2.2.1 Lasso and Group Lasso

In the context of ordinary least squares (OLS) estimation, "Least absolute shrinkage and selection operator" (Lasso) is a popular and powerful tool for reducing the number of predictors [34]. Consider the following general univariate linear regression problem in equation 2.2 , where we assume there are N observations and x_i is the i-th observation of single variable, y are the dependent variables of x and ε is the residual. We call the dependent variable y as the "target". For each distinct variable x_i, there is corresponding weight parameter β_i that will be optimized.

$$y = \sum_{i=1}^{N} \beta_i x_i - \varepsilon \tag{2.2}$$

The multivariate version of this general problem can be easily derived from stacking equation 2.2. In other words, making it a matrix. Let's keep the independent variable set $[x_1, \ldots, x_i]$ unchanged and denote the total k multivariate targets as $\hat{y} = [y_1, \ldots, y_k]^T$. The corresponding $\hat{\beta^j} = [\beta_1^j, \ldots, \beta_i^j]$. Then we have:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} = \begin{bmatrix} \hat{\beta^1} \\ \vdots \\ \hat{\beta^k} \end{bmatrix} \begin{bmatrix} x_1 & \ldots & x_i \end{bmatrix} \tag{2.3}$$

Lasso method [35] solves the equation 2.3 along with the constraint to the $l1$ norm of $\hat{\beta^k}$, which effectively reduces the value of unknown factors β_i^k to zero if it is less a relevant factor. Denote $\hat{\beta} = [\hat{\beta^1}, \ldots, \hat{\beta^k}]^T$, the multivariate Lasso's formula can be written as:

$$(\varepsilon, \hat{\beta}) = \arg\min \left\{ \sum_{k=1}^{M} \left(y_k - \varepsilon - \hat{\beta}^k x \right)^2 \right\}$$
$$\text{subject to} \sum_{j}^{M} |\hat{\beta}^k| \le t \tag{2.4}$$

where t is a hyperparamter for tuning.

The objective function of group Lasso algorithm is a variant of equation 2.4. One drawback of equation 2.4 is that it assume every variable x_i is a separated individual. To generalize Lasso to grouped variables, a modified constraint was introduced which leads to group Lasso [2, 25]. The formulation can be written as:

$$(\varepsilon, \hat{\beta}) = \arg\min \left\{ \sum_{k=1}^{M} \left(y_k - \varepsilon - \hat{\beta}^k x \right)^2 \right\}$$
$$\text{subject to} \sum_{j}^{M} \sum_{g}^{G} |\hat{\beta}^k \chi_g| \le t \tag{2.5}$$

where χ_g is the index set belongs to the gth group.

2.2.2 Method of Lagrange Multipliers

We just introduced the objective function of group Lasso algorithm, which is an optimization problem. In this section, we concern how to solve that objective. The method of Lagrange multipliers is a commonly used technique for optimization. Consider a optimization problem with inequality constraint:

$$\begin{gathered} \text{Minimize: } f(x) \\ \text{Subject to: } g(x) \le \alpha \end{gathered} \tag{2.6}$$

Its Lagrange function is given by:

$$\mathcal{L}(x, \lambda) = f(x) - \lambda g(x) \tag{2.7}$$

The new introduced parameter λ is called Lagrange (undetermined) multiplier. It's been proved that if $f(x_0)$ is a minimum of $f(x)$, then there exists a λ_0 so that (x_0, λ_0) is a stationary point of the Lagrange function.

Then, finding the solution of eq.2.6 is transformed into finding λ_0 and stationary points of the eq.2.7. It is common to find ideal λ_0 by grid search and checking stationary points is a standard Calculus problem.

2.2.3 K-Means Clustering

The K-means[24] clustering is a popular unsupervised clustering machine learning algorithm. It takes a given number (k) clusters and tries to group given data into that many clusters. The algorithm is NP hard to solve but there exists heuristic ways for quick convergence to a local minimum. The objective function of k-means clustering can be formulated as:

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{z_j \in c_i} \|z_j - c_i\|^2 \tag{2.8}$$

where $Z = \{z_1, \dots\}$ is the data set where z_j is an observation vector with arbitrary dimensions, which will be grouped into clusters. k is the total number of clusters and $C = \{c_1, \dots, c_k\}$ is the cluster set where $M = \{\mu_1, \dots, \mu_k\}$ denotes the corresponding cluster center set. Denote $z_j \in c_i$ if z_j is labeled as a member of a cluster with center μ_i. One popular iterative method to find Z is as follows:

2.3 Background for Deep Learning Algorithm

In this section, we present the neural network structures and techniques used in the proposed deep learning algorithm, VoltNet.

2.3.1 Activation Functions and SELU

Starting from this section, the remainder of this chapter will introduce the background knowledge for our proposed deep learning model VoltNet for sensor placement and prediction. Let us begin with the most fundamental element of the neural network: activation function.

Algorithm 1: Iterative Solver for K-Means Clustering

```
Input: k⟵ number of clustering,    Z⟵ $\{z_1, \dots\}$, $z_j \in \mathbb{R}^n$
Output: C⟵ $\{c_1, \dots, c_k\}, c_1 \in \mathbb{R}^n$,   $M \longleftarrow \{\mu_1, \dots, \mu_k\}$
begin
    Initialize M randomly
    repeat
        for $z_j \in Z$ do
            $i' = \arg\min_i \|\mu_i - z_j\|$
            $c_{i'} \cup z_j$
        end
        $M_{old} = M$
        for $\mu_i \in M$ do
            $\mu_i = \sum_{z_i \in c_i} z_i \quad / \quad \sum_{z_i \in c_i} 1$
        end
    until $M = M_{old}$
end
```

Consider a fully connected neural network or multilayer perceptron (MLP) network. This neural network connects the inputs to the outputs with a user-defined number of hidden layers where each layer contains a certain number of nodes for processing. The activation function is the function used for that processing. For a single node, this processing can be formulated as:

$$y = f(\hat{W} * \hat{x} + b) \tag{2.9}$$

where $f(*)$ is the activation function, y is the output, $\hat{x}$ is the input vector, $\hat{W}$ is the corresponding weight and b is a constant for bias.

For the MLP model, mostly, only nonlinear functions are used. However, in general, activation function can be any differentiable function. Naturally, different activation functions have their own characteristics; therefore, their own application scenarios. For VoltNet, we used scaled exponential linear unit (SELU) for our fully connected layers.

SELU[16] is a self-normalizing activation function that can accelerate the training and improve prediction accuracy[28]. SELU is a piece-wise function as follows:

$$SELU(x) = \lambda \begin{cases} x & x > 0 \\ \alpha e^x - \alpha & x \leq 0 \end{cases}$$

where $\alpha =$ 1.6732 and $\lambda =$ 1.0507 for standardized inputs.

2.3.2 Sigmoid Function

Another activation function to introduce is the sigmoid function. Its formula is shown below.

$$S(x) = \frac{1}{1 + exp(-\hat{W}\hat{x} - b)} \tag{2.10}$$

where x is the input vector to the layer; W is the corresponding weight; and b is the bias.

The sigmoid function is suitable to model the probability of a class thanks to its close relationship with logistic regression. Logistic regression models the logit function $\ln p/(1-p)$ with linear combination. Consider the probability of target y being class 1 given x or formally as $P(y = 1|x)$, applying Bayes' theorem, we get:

$$\begin{aligned} P(y=1|x) &= \frac{P(x, y=1)}{P(x)} \\ &= \frac{P(x|y=1)P(y=1)}{P(x|y=1)P(y=1) + P(x|y=0)P(y=0)} \\ &= \frac{1}{1 + \frac{P(x|y=0)P(y=0)}{P(x|y=1)P(y=1)}} \\ &= \frac{1}{1 + \exp(\ln(\frac{P(x|y=0)P(y=0)}{P(x|y=1)P(y=1)}))} \end{aligned} \tag{2.11}$$

Applying Bayes' theorem again

$$= \frac{1}{1 + \exp(-\ln(\frac{P(y=1|x)}{P(y=0|x)}))}$$

Note that the natural logarithm term in the denominator is a logit function or logarithm of odds. Logistic regression models this logit function by linear combination of x which writes:

$$\ln(\frac{P(y=1|x)}{P(y=0|x)}) = Wx + b \tag{2.12}$$

Submitting equation 2.12 back to $P(y = 1|x)$ we get the sigmoid function. This means that sigmoid function models the probability directly.

2.3.3 Pruning

Neural network pruning or optimal brain damage [19] is a popular method for optimizing neural networks. Empirically, neural networks, especially deep networks, have millions of parameters to train and many of them contributes meagerly for the prediction. During the training, pruning method uses a chosen rank system to evaluate the importance of each parameters and remove the lesser ones. Intuitively, elimination of less important parameters would result in a reduced training time and faster prediction. There are also various evidence that a good pruning scheme can improve the prediction accuracy as well [26]. Less important parameters can act like a noise which may damage the accuracy overall.

Pruning is done in an iterative manner. Usually, it starts as normal training for some epoches. Later in the training, a parameters will be gradually removed as the training continues. Any removal would result in a temporary drop of the accuracy but can be recovered as the training continues. The flowchart is shown in fig. 2–1

In this thesis, the pruning is used for sensor placement and the evaluation of importance is measured by the magnitude of the parameter or the l_1 norm of the parameter. Rewrite the equation 2.9 for the whole hidden layer, we get:

2.3.4 LSTM

Recurrent neural network (RNN) [31] is a special network structure that is designed for temporal data analysis. When an input is given to RNN, it process the input one time-step at a time and the output is determined by both previous and current inputs, unlike

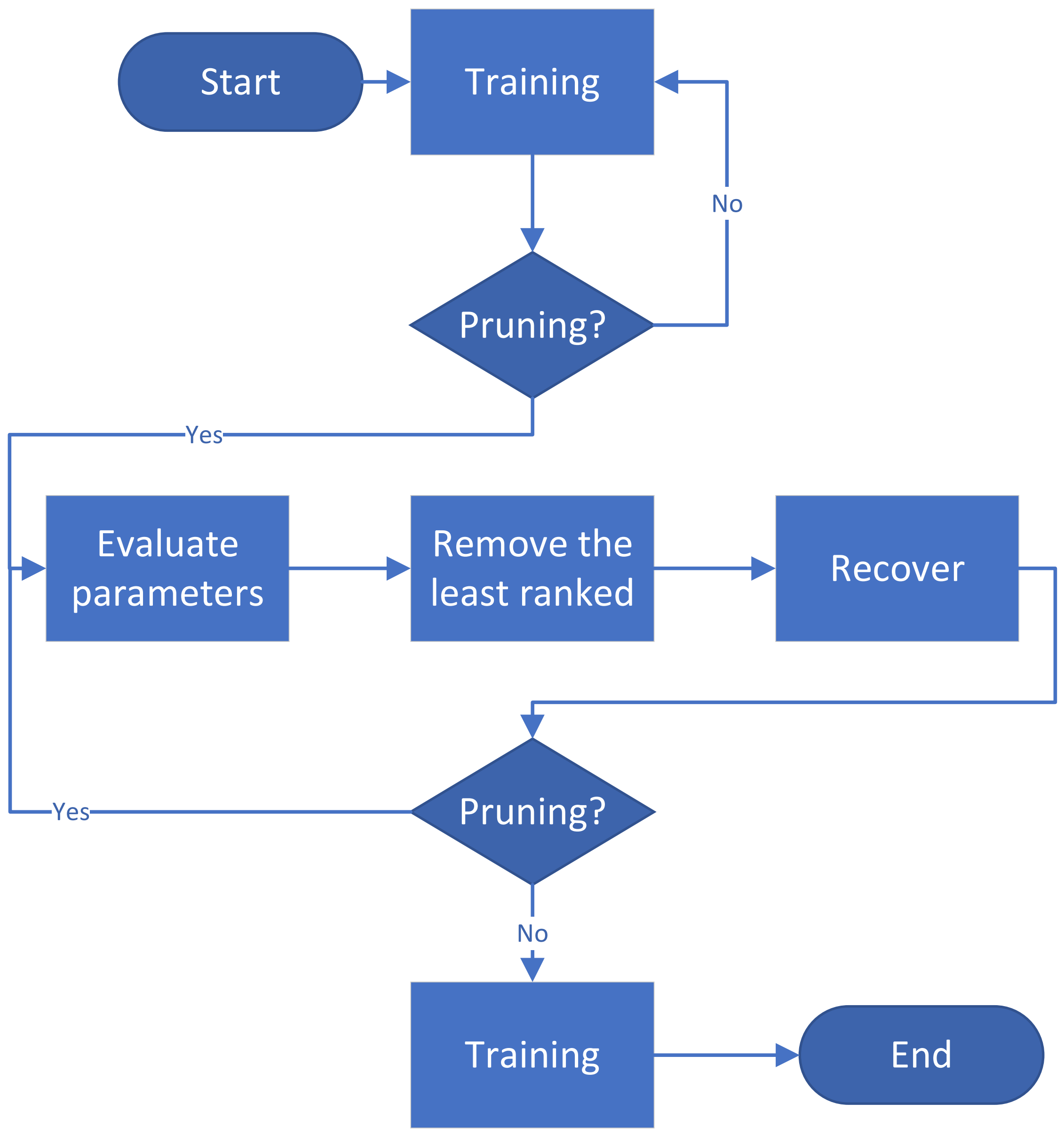

Figure 2–1: Pruning procedure

other structures, i.e. MLP, requires whole input at once. This characteristic is achieved by a recurrent cell structure which is dramatically different from the traditional layer-like structure. For a vanilla RNN, the hidden state is also its output and it is determined by both previous hidden state and the current input.

As shown in fig. 2–2, the cell is usually initialized with hidden states of 0. When an input is given, the first time step is passed to the cell and a hidden state is calculated. In next time step, the new hidden state is calculated by previous hidden state and current input. The previously mentioned cell is a group of neuron-like units where the number of

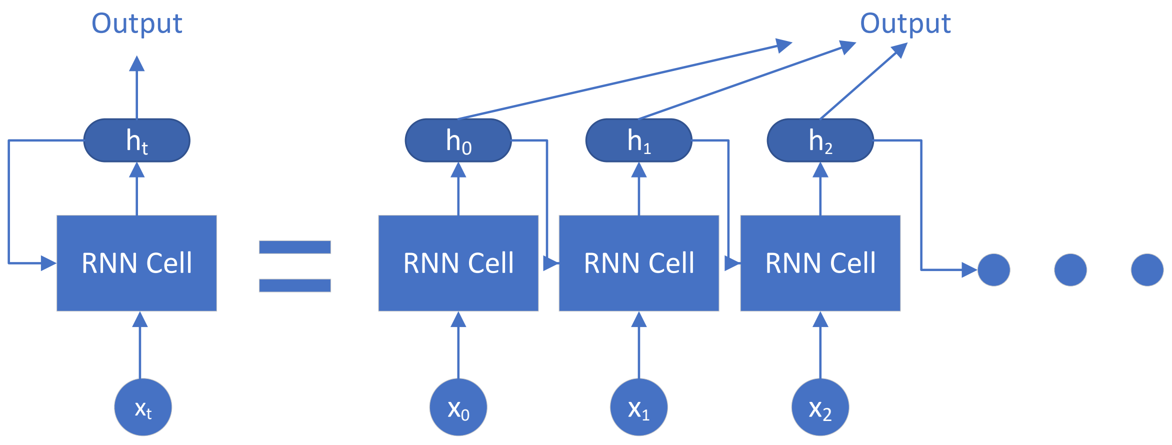

Figure 2–2: High level architecture of RNN; a recurrent representation is on the left hand side and a unrolled representation is on the right hand side.

those ”neurons” is a hyper parameter to set. Naturally, the hidden state produced by cell is a vector of that many dimensions. Fig. 2–3 shows a inner structure of RNN cell.

Long short-term memory (LSTM) [15] is a more complicated version of vanilla RNN. The main difference is the ”neuron” structure and the hidden state. In addition to the hidden state, LSTM has a cell state which is very similar to hidden state. The difference is that cell state is updated by gates and the hidden state is a function of cell state in addition to current inputs and previous hidden state. A standard LSTM ”neuron” can be derived by replacing the sigmoid function in fig. 2–3 with formula below.

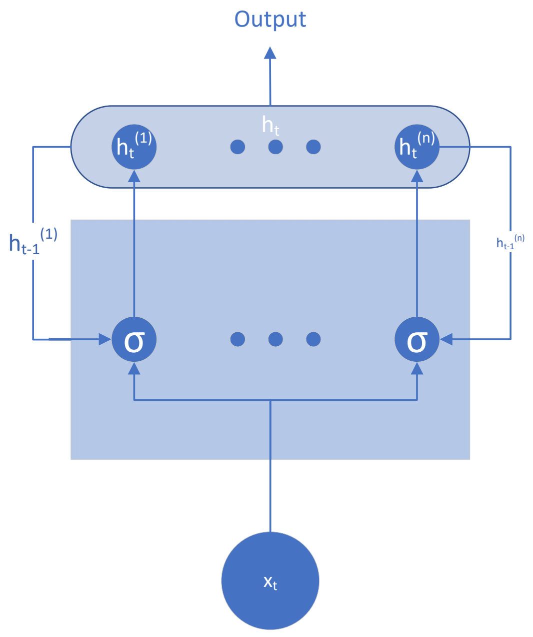

Figure 2–3: Structure of a RNN cell

$$\begin{aligned}
\vec{i} &= \sigma(\vec{h}_{t-1}\mathbf{U}_i + \vec{x_t}\mathbf{W}_i) \\
\vec{f} &= \sigma(\vec{h}_{t-1}\mathbf{U}_f + \vec{x_t}\mathbf{W}_f) \\
\vec{o} &= \sigma(\vec{h}_{t-1}\mathbf{U}_o + \vec{x_t}\mathbf{W}_o) \\
\vec{g} &= \tanh(\vec{h}_{t-1}\mathbf{U}_g + \vec{x_t}\mathbf{W}_g) \\
\vec{c_t} &= \vec{i} \circ \vec{c_{t-1}} + \vec{i} \circ \vec{g} \\
\vec{h_t} &= \vec{i} \circ \tanh(c_t)
\end{aligned} \tag{2.13}$$

where $\vec{i}, \vec{f}, \vec{o}, \vec{g}$ stands for input gate, forget gate, output gate and input modulation gate; c_t is the cell state at time step t and h_t is the hidden state at time step t; $\mathbf{W} = \{\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_o, \mathbf{U}_g, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_g\}$ is the weight of respective gates.

The LSTM architecture used in this thesis is called deep bidirectional LSTM (BLSTM)[18, 11, 33]. The rest of this section is devoted to illustrate the architecture and explain its improvement to the accuracy.

By being bidirectional, the hidden state at any given time step is determined by both the input from prior time steps and future time steps. This is achieved by stacking a forward LSTM and a backward LSTM together as shown in fig. 2–4. The backward LSTM takes the last time step of a given input sequence and recurrently continues to prior time steps, which is the opposite of the forward LSTM. Naturally, the hidden state of BLSTM consists that of the forward and backward LSTM.

One problem of BLSTM is that the forward and backward information is not distributed evenly to all hidden states. One extreme case is shown in fig. 2–5. At the first LSTM layer, the hidden state h_n has all the information from forwarding LSTM shown as yellow arrows but none from the backward. At the second LSTM layer, h'_n now has full information from backward LSTM shown in red arrow. Despite the problem, a significant improvement in accuracy has been observed from literature. Most state-of-the-art natural language processing models used some stacked RNN. A quick summary can be found in [10].

2.3.5 Skip Connections

Many neural networks are sequential, which means that the input of one layer is the output of its previous layer. To increase the capacity of the model, people tend to use deeper model rather than wider to reduce overfitting and training time. However, deeper architecture caused a well-known issue: vanishing gradient. The neural networks are mostly trained by backpropagation, which calculates the gradient by the chain rule.

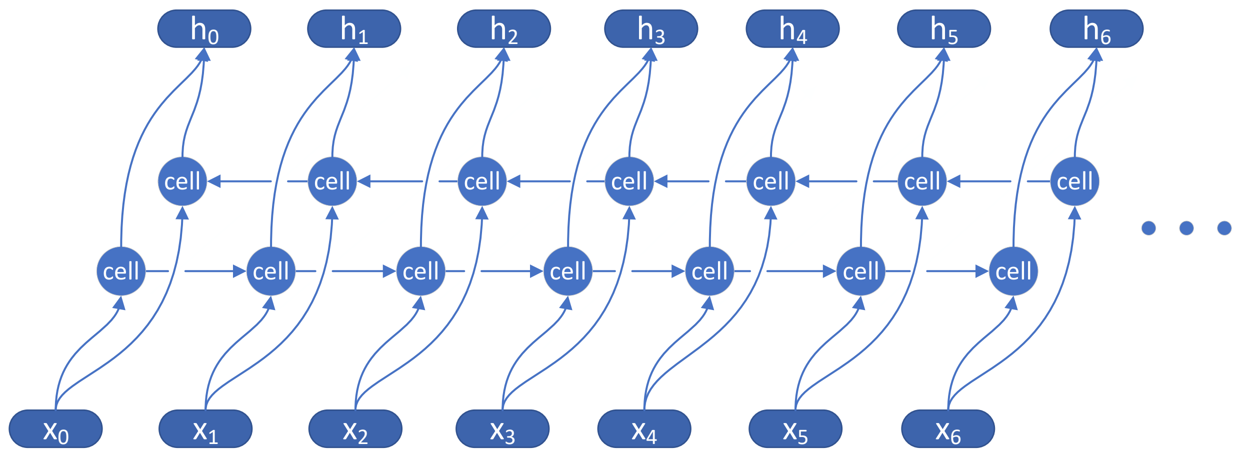

Figure 2–4: Typical architecture of bidirectional LSTM

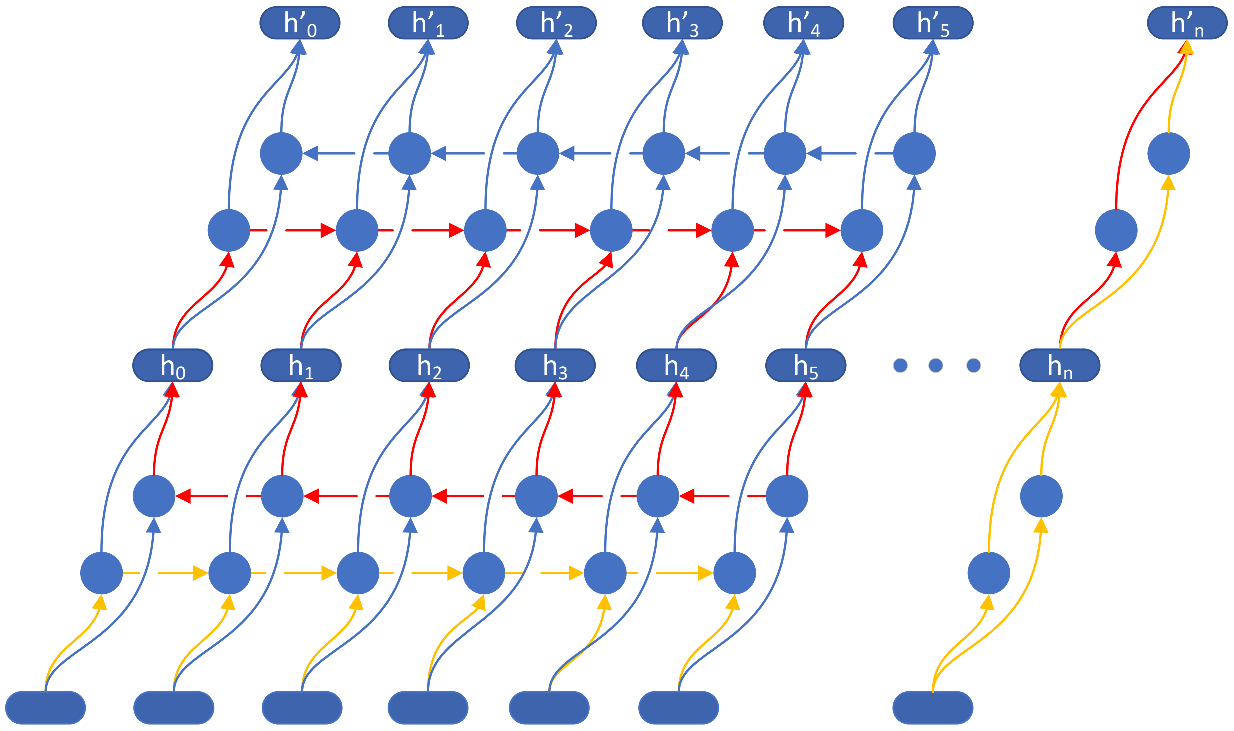

Figure 2–5: Forward and backward information flows in stacked bidirectional LSTM. Yellow arrow is the forward flow for h'_n; red arrow is the backward flow for h'_n.

The learning process is mostly updating the wights of layers by the gradient. By the chain rule, the calculation of one gradient involves multiplying the gradients of layers after it. Many popular activation functions usually yield gradients smaller than 1 and multiplying multiple of them will result in numerically negligible number. Hence, it directly slows or stops the learning, especially in deep networks.

Skip connection [13] is a unique bridge within the neural network that embeds the input of a previous layer to that of the following layer. This connection allows calculating gradients without considering the gradients of bridged layers, which counters the vanishing gradient problem. The embedding mechanism is usually implemented in one of two flavors. Namely, additive skip connection and concatenate skip connection.

As shown in fig. 2–6, the input to the later end of the connection comprises the output of the previous layer and the input on the other end of the connection. An additive

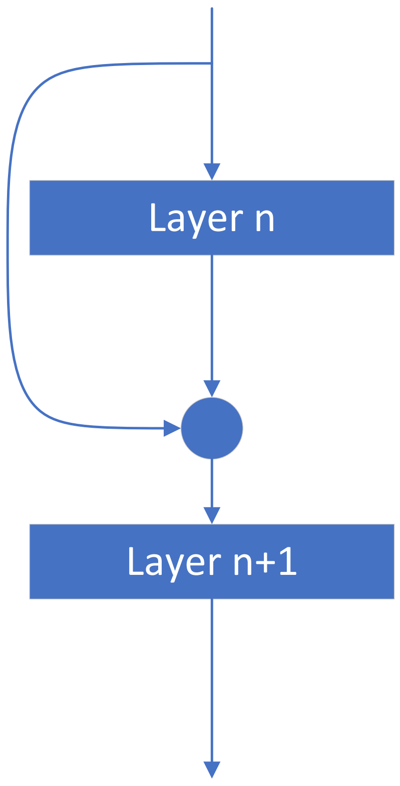

Figure 2–6: A typical skip connection architecture

skip connection adds the output and the input while a concatenate skip connection concatenate.

Concatenate skip connection produces very clean input to the following layer at the expense of expanding feature space. On the contrary, additive skip connection preserves the input space at the cost of noisy input.

CHAPTER 3
Preview of Related Algorithms in the Literature

3.1 Problem Formulation

It is requisite to predict voltage-noise emergency and throttle the voltage in time to ensure the correctness of CPU operation.

We took a divide-and-conquer way to predict the voltage noise emergency. The problem can be divided into 2 part, namely, voltage sensor placement and emergency prediction. The idea is to place a few sensors on the CPU to gather the necessary information for the prediction algorithm. With this division, the throttling scheme is:

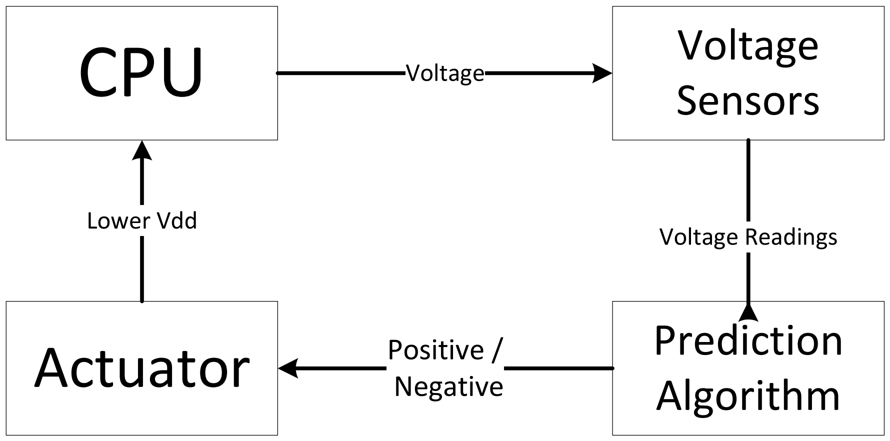

Figure 3–1: Throttling Scheme

In this section, a commonly accepted formal formulation of on-chip voltage sensor placement problem is presented [22, 37]. The notation used in this formulation will be consistent across the thesis.

With four listed inputs, the definition of on-chip voltage sensor placement problem can be described as follows.

1) A collected data set D.
2) A candidate node set $S = \{s_1, \ldots, s_m\}$ containing m candidate nodes for voltage sensor placement. In addition, we define all other nodes as target nodes.
3) A percentage $P = \pm p\%$ denoting the noise-margin for detecting a voltage emergency. Or as absolute value $t = (1 + P) * r$, where r is the supplied VDD voltage. Following [12], we chose $P = \pm 4\%$ for the rest of the thesis.
4) An integer N denoting the total number of sensors to be placed.

The objective is to find selected node set C with N optimal nodes in S so that the chosen loss function is minimized. The loss functions applied to the placement algorithms are different and will be introduced in their respective sections.

The prediction problem is a canonical supervised binary classification problem. Like the formulation for sensor placement problem, voltage-noise emergency prediction problem can be formulated as:

Given the inputs:

1) A set of time-series of voltage data $\mathbf{V} = \{\hat{v}, \ldots, \hat{v}\}$.
2) An integer O denoting the predictive power. It describes the maximum time that a prediction can be made ahead of the actual phenomenon. It is quantified by CPU cycles.

The objective is to predict whether an emergency will happen in O CPU cycles.

3.2 Previous Voltage Sensor Placement Techniques

The ultimate goal for voltage sensor placement is to optimize the voltage emergency detection/prediction. Although it is vital for most prediction algorithms, the problem is still open. From our perspective, the most challenging part is evaluating sensor placement algorithms.

Every placement algorithms have their unique evaluation metrics. Hence, it is impossible to compare them consistently. In principle, the best evaluation should be comparing their accuracy improvements for predicting the violations. However, this metric will inevitably introduce a prediction algorithm which may have an affinity for a specific placement algorithm. As a result, the prediction algorithm becomes a biased judge, which makes a direct comparison of sensor placement algorithms unfair. This conundrum raises the necessity for the exploration of new metrics.

In this section, two state-of-the-art sensor placement algorithms along with their evaluation metrics will be briefly introduced, which will be referenced heavily in later sections.

3.2.1 Eagle-Eye

Eagle-Eye [37] is a greedy algorithm that maximizes the probability of placing sensors right on top of the hot-spots of a given microprocessor. The algorithm straightforwardly maximizes the sensing quality metric (SQM). To better illustrate the algorithm, we first introduce the evaluation metrics they used - SQM and miss rate.

Miss Rate is the probability of the occurrence of voltage violation when all placed sensors failed to detect any violations at their monitored nodes. It can be formulated as:

$$\begin{aligned} Miss \quad Rate &= P(Z_{max} \geq t | Z_{r1} \leq t, Z_{r2} \leq t, \ldots, Z_{rs} \leq t) \\ &= P(Z_{max} \geq t | \max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t) \end{aligned} \tag{3.1}$$

where Z_{max} is the maximum noise among all nodes in the power grid and Z_{rn} is the voltage of nodes with sensors placed. It is obvious that to minimize the *Miss Rate*, one need to maximize the probability that nodes with sensors do monitor an emergency. The rigorous derivation was provided as:

$$\begin{aligned} & P(Z_{max} \geq t | \max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t) \\ = 1 - & P(Z_{max} \leq t | \max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t) \\ = 1 - & \frac{P(Z_{max} \leq t, \max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t)}{P(\max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t)} \\ = 1 - & \frac{P(Z_{max} \leq t)}{P(\max(Z_{r1}, Z_{r2}, \ldots, Z_{rs}) \leq t)} \end{aligned} \tag{3.2}$$

This is defined as *Sensing Quality Metric* (SQM) which is formulated as:

$$SQM = P(\max(Z_{r1}, \ldots, Z_{rs}) \geq t) \tag{3.3}$$

The Eagle-Eye greedily seeks the maximum of SQM by selecting the nodes. For n sensor budget, it selects 1 candidate node per iteration. Moreover, in each iteration, the additional node that increases SQM the most is selected. Denote the additional node as k and the selected node set as $\mathbb{S}$; this thesis calculates the new SQM for comparison as:

$$\text{SQM}(\mathbb{S} \cup k) = \text{SQM}(\mathbb{S}) + \text{SQM}(k) - \text{SQM}(\mathbb{S} \cap k) \tag{3.4}$$

3.2.2 Group Lasso

Briefly, the Group Lasso [23] placement algorithm substitutes the formulation 3.1 into the group lasso optimization formula and tries to derive a linear mapping between the candidate sensors and all the nodes on the grid. Candidates are ranked according to the linear mapping, and the top ranks exceeding a certain threshold P will be selected.

To illustrate the algorithm in detail, we briefly introduce the voltage grid here. The time-series voltage data for validating and training all mentioned algorithms is the final product from a simulation toolchain. For each time instance, the data includes voltages for each node on a CPU. Each node represents a small area of the CPU, and all the nodes comprise of a grid, as shown in the figure 3–2.

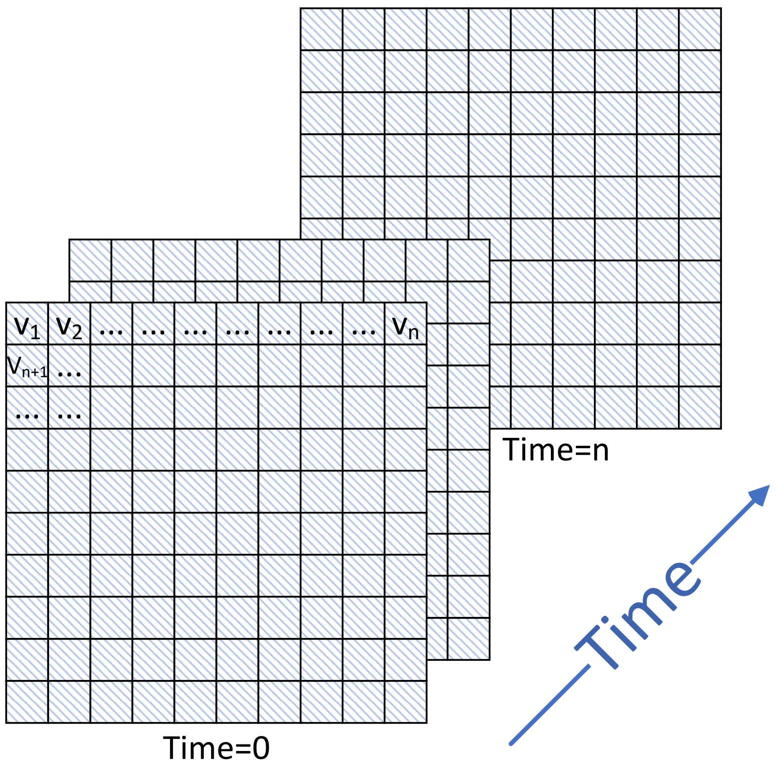

Figure 3–2: Voltage grids from the toolchain.

Firstly We assume there are total g nodes in a given voltage grid. Then let $\hat{v}_k = [v_1, v_2, \dots, v_{g-m}]^T$ and $\hat{e}_k = [e_1, e_2, \dots, e_m]^T$ denote the voltages of target nodes on the grid and the candidate nodes at Time $= k$, respectively. Now we can denote the whole data set as $V = [\hat{v}_1, \hat{v}_2, \dots]$ and $E = [\hat{e}_1, \hat{e}_2, \dots]$.

Standardization is applied to both V and E to obtain zero mean and unit variance. Also let $Y = [\hat{y}_1, \hat{y}_2, \dots]$ and $X = [\hat{x}_1, \hat{x}_2, \dots]$ be the column-wise standardized V and E respectively. One critical assumption of this algorithm is that the voltage on the grid can be approximated as:

$$e_1 = \sum_{i=1}^{m} a_{1,i} v_i \tag{3.5}$$

where a is the weight of placed sensors.

Follow the above assumption for a single node, then the optimization objective for voltage sensor selection for the whole grid could be written as:

$$\hat{y}_k = \begin{bmatrix} \hat{\beta}^1 \\ \vdots \\ \hat{\beta}^k \end{bmatrix} \hat{x}_k \tag{3.6}$$

For all the time instances, above equation can be horizontally stacked as below:

$$\begin{bmatrix} \hat{y}_1 & \dots & \hat{y}_k \end{bmatrix} = \begin{bmatrix} \hat{\beta}^1 \\ \vdots \\ \hat{\beta}^k \end{bmatrix} \begin{bmatrix} \hat{x}_1 & \dots & \hat{x}_g \end{bmatrix} \tag{3.7}$$

This is solved using group lasso optimization. By apply equation 2.5 into equation 3.7, the group Lasso optimization formula for voltage sensor placement algorithm is:

$$(\varepsilon, \hat{\beta}) = \arg\min \left\{ \sum_{k=1}^{M} \left(Y - \varepsilon - \hat{\beta} X \right)^2 \right\}$$
$$\text{subject to} \sum_{j}^{M} \sum_{g}^{G} |\hat{\beta}^k \chi_g| \leq t \tag{3.8}$$

Furthermore, the weights' magnitude directly correlates with the importance of that candidate sensor as the larger the weight gets, the more contribution is from that node. Therefore, the score of ith sensors can be described as:

$$\text{importance} = \left\| \begin{bmatrix} \hat{\beta}_i^1 \\ \vdots \\ \hat{\beta}_i^k \end{bmatrix} \right\|^2 \tag{3.9}$$

All sensors with a score higher than T will be selected where T is a tuning parameter.

Unfortunately, this algorithm does not come with its unique metric function. It was evaluated by prediction accuracy where the prediction was made by an ordinary least square (OLS) optimization.

Pseudo-code is shown below:

```
Algorithm 2: Group Lasso with Cross Validation for Sensor Placement
Input: S⟵ Candidate node set,   P⟵ noise margin,   N⟵ number of sensors
       to be placed,   D⟵ voltage grid over time
Output: $C$ ⟵ selected node set
begin
    Separate D into $\mathbf{V}$ and $\mathbf{E}$
    $\mathbf{Y}$ = standardize($\mathbf{V}$)
    $\mathbf{X}$ = standardize($\mathbf{E}$)
    Select $t$ based on experience
    Solve group lasso optimization by eq. 3.8 with $t$
    Calculate importance for all nodes $\vec{I} = \{i_1, \dots, i_m\}$ by eq. 3.9
    Select $T$ based on experience
    $C = \{s_i | i_i > T\}$
end
```

3.3 Signature-Based Voltage Emergence Prediction Algorithms

Another route for predicting voltage emergency is by analyzing microprocessor trace. The trace is the CPU operations' statistics over time, i.e. the read of L2 cache over 100 CPU cycles. This type of algorithms try to find certain patterns that directly correlate to voltage emergencies.

According to the works in [12, 30], the main idea of this approach is to find a handful of very frequent CPU trace patterns that induce the majority of the voltage emergencies. In this way, the voltage emergencies can be predicted without the voltage sensor on IC. In addition, the trace patterns are supposed to have less noise than the patterns of voltage. On one hand, the CPU trace pattern comes from code loops which are inherently identical. On the other hand, the CPU traces are digital and do not suffer from process variation like the voltage sensors do.

In this direction, perhaps the correlation between machine code and voltage emergency could also be found.

3.4 Optimum Sensor Placement Problem in Other Areas

As the study for voltage sensor placement begins to increase, it can be helpful to explore the established solutions in similar problems. And the temperature sensor placement problem is extremely similar to the voltage noise counterpart. In this area, the algorithms of placing temperature sensors varies, but the prediction is mostly a regression problem that relies on the correlation between selected nodes and other nodes, which is backed up by the second law of thermodynamic. In the following section, 2 types of popular placement techniques are briefly introduced which inspired some algorithms proposed in this thesis.

3.4.1 Correlation-Based Placement

This type of techniques [29, 6] explore the correlation of all the nodes on the CPU. And the least correlated nodes are selected to maximize the information about the map because they are the hardest to reconstruct in the later prediction stage.

3.4.2 Clustering-Based Placement

This type of techniques [17, 24] considers geometric characteristic of nodes with previous “heat emergencies”. By placing sensors in the cluster, it would maximize the information from the hottest region.

CHAPTER 4
Automated Data Generation Flow

4.1 Introduction to the Data Generation Toolchain

Acquiring the data for analysis is one of the most critical problems in machine learning. Some researches tried to extrapolate the data and inevitably introducing additional variance to their model. This chapter will introduce the toolchain for per-CPU-cycle simulation and the simulated Penryn-like microprocessor architecture[39, 36]. The toolchain can automatically simulate all the voltage sensor data on a given microprocessor per microprocessor cycle and later used for our voltage sensor placement algorithms. The interfaces for those programs can be found here [1], which can automate the whole process.

The automated toolchain builds around VoltSpot which is the last stage of the simulation. VoltSpot models the CPU power delivery as a mesh grid comprised of lump components. It calculates the voltages at each node on the grid by transient analysis. The final product is three dimensional data where two of three dimensions are coordinates on the grid with an additional time dimension. The latter half of this chapter will illustrate all modifications to this raw data for training models.

4.2 Structure of the Toolchain

The toolchain is summarized in fig.4–1. For reproducibility, the detailed procedure is described below.

1. The Penryn-like microprocessor architecture is partially defined in Gem5 [5]. Full system simulations are run with workloads from PARSEC 2.0 [4].
2. The simulation result from Gem5 is processed by Gem5-McPAT interface.
3. McPAT [21] is used to get the power and area of the Penryn-like microprocessor.
4. The simulation result from McPAT is processed by McPAT-VoltSpot interface.

	2-core
CPU-clock (GHz)	3.7
Supply Voltage (V)	0.8
L1 data cache (kB)	32
L1 instruction cache (kB)	32
L2 cache (MB)	6

Table 4–1: Gem5 microprocessor setting

5. The geometric information of the CPU is described in the floorplan file, which is created using ArchFP.
6. The final voltage data is from a modified version of VoltSpot2.0 [41] with similar configurations in [39].

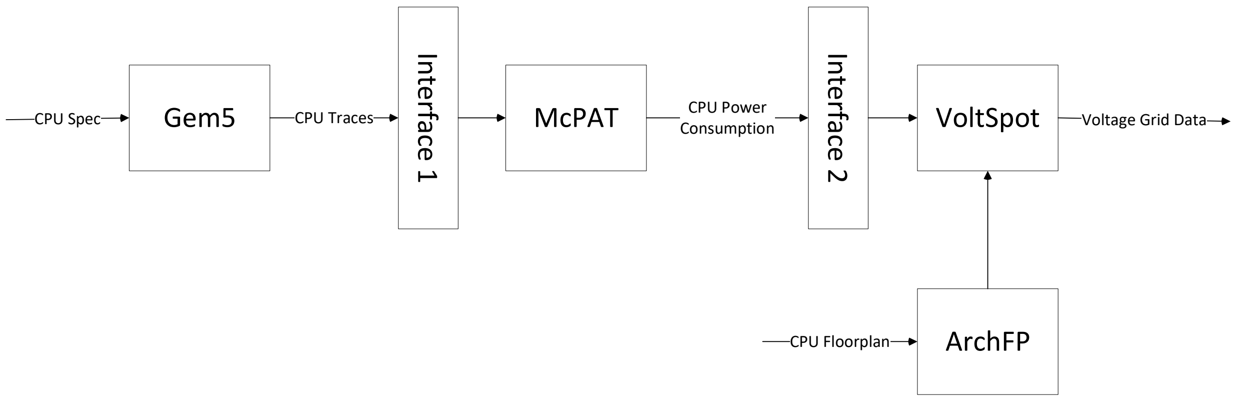

Figure 4–1: The flow chart for the simulation tool chain

The above procedure is automated by bash scripts where a template has been provided in [1]. Besides, the same script, with minor modification, can pipeline data for online analysis.

4.3 General Experiment Set Up

This thesis used a Penryn-like microprocessor architecture for demonstration purpose. A 2-core version of such microprocessor is used for illustrating the key points of algorithms. The architecture is partially defined in Gem5 and can be summarized as:

Table 4–1 only introduces 2-core architecture because the other versions are achieved by replicating the basic structure of 2-core microprocessor. The floorplans of all three microprocessors are shown below, which is a mandatory input to the VoltSpot.

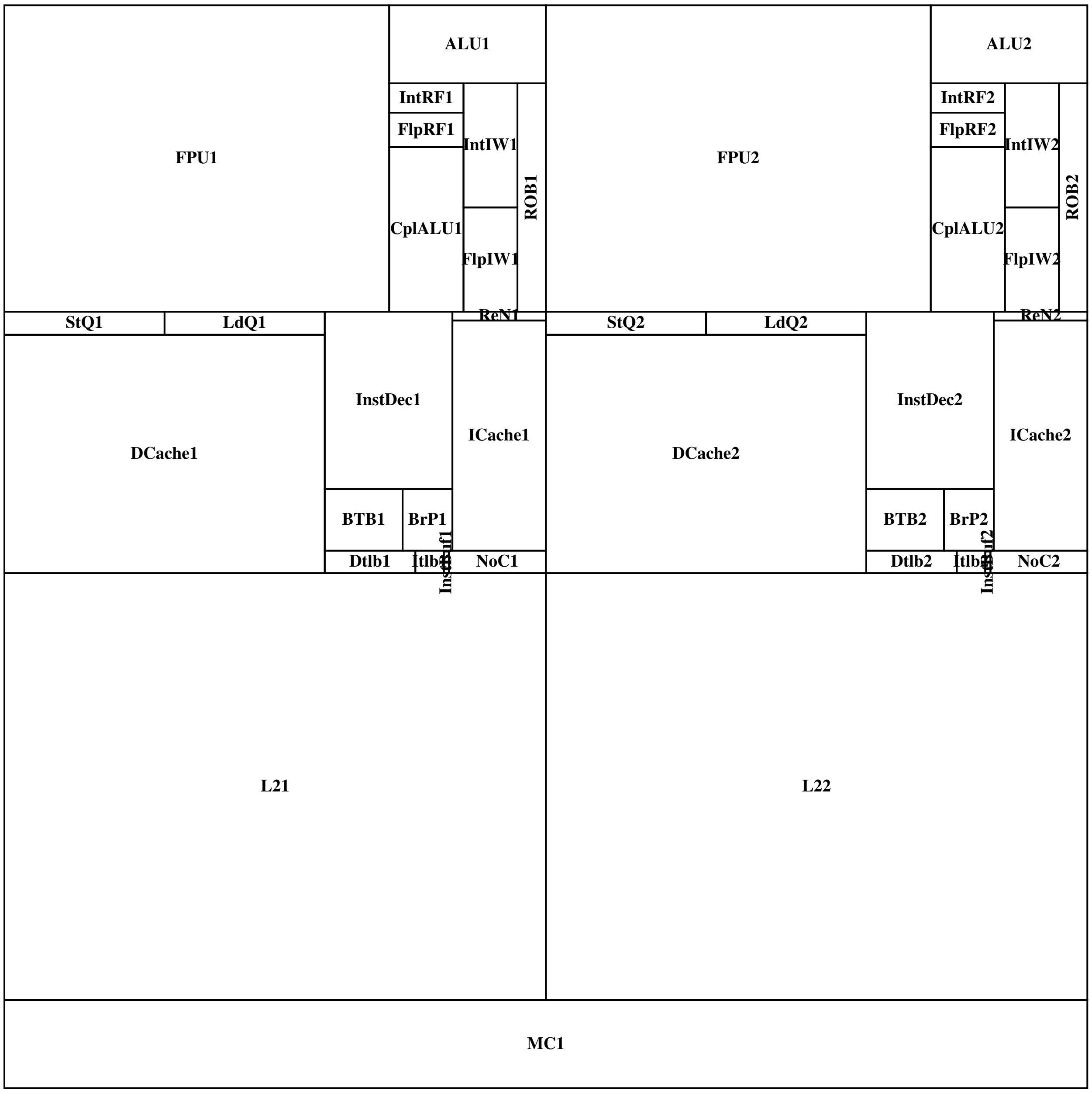

Figure 4–2: Floor plan of 2-core Penryn-like CPU

The concludes the configurations for Gem5. Because of the triviality of running McPAT, we directly skip to the configurations of VoltSpot. VoltSpot is essentially a grid-based lump-component model for on-chip microprocessors. In particular, the grid is formed by partition the microprocessor into cells with the size of the C4 pad. The voltage data is acquired by applying transient analysis to this model. For any given CPU cycle, the voltage data is the instantaneous voltage of each cell of the grid-based model. The critical configurations for setting up VoltSpot is the same as in [39].

4.4 Output Structure of Simulation

This section discusses the product of introduced toolchain. The direct product is a time-series voltage grid that has shown in Chapter 3. The unit for time is CPU cycles. As a recap, one can consider a voltage grid as the voltage readings across the CPU. Each data point in a voltage grid represents a node. A node is a small area on the CPU. In other words, a voltage reading on the voltage grid is an average voltage of a small area on the CPU. We call the direct product as the raw data.

As a topic for a later chapter, we discuss three prediction algorithms in this thesis. They require very different data structure for training. While the time-series grids from the toolchain are good enough for training linear regression algorithms, it is not enough for the neural network algorithm. For that, we need batches of time-series grids over a constant small time interval. Each prediction requires one batch of such data. We call such batches of data as voltage grid traces. Voltage grid traces are essentially chopped raw data. Within one grid voltage trace, the time-series voltage of a certain node is called the voltage trace.

4.5 Prepared Data Set for Training

In this section, the preparations of three different data set will be introduced. They all comes from the same simulation data but with different format, due to the varying input format requirements of models.

4.5.1 Time-series Voltage Grid

As one of the three training set, this one is essentially the vanilla output from out simulation toolchian. Both group Lasso and Eagle-Eye algorithms uses this training set.

Benchmarks Name	Benchmarks Application
blackscholes	Option pricing with Black-Scholes Partial Differential Equation (PDE)
bodytrack	Body tracking of a person
freqmine	Frequent itemset mining

Table 4–2: Summary of benchmark purpose

This data set is a collection of three continuous simulation toolchain outputs. The simulations respectively simulates three different programs from PARSEC benchmarks suite [4], namely, blackscholes, bodytrack and freqmine. The benchmarks covers different areas and they can be summarized in table 4–2

As introduced before, the data from the simulation toolchain are voltage grids over simulated time. In other words, a complete collection of voltage trace over the whole CPU. There are totally 5,005,062 CPU cycles in this data set and each benchmark takes approximately 1/3 of the total cycles.

The data set has a shape of [samples, nodes] where a 2D voltage grid is flattened to a vector in the second dimension.

4.5.2 Voltage Grid Trace

This data set is used for classification purpose, which means there is a label for each trace denoting whether an emergency would happen, a few CPU cycles later, after the trace. The gap between the end of the trace and the emergency-checking time point is called prediction capability denoted as γ.

The voltage grid traces are continuous and 50 CPU cycles long chopped from time-series voltage grid data set. However, many data is discarded when preparing. This data set collects only voltage grid traces that would induce an emergency and an equal amount of randomly sampled "normal" traces. It is balanced with equal amount of positive and negative samples. Emergencies are rare, so this set is substantially smaller than the previous data set with roughly 250,000 traces.

The data set has a shape of [samples, nodes, trace] where "trace" is the 50 CPU cycles. In addition, it has a separate label vector with a label for each sample.

4.5.3 Voltage Trace

This data set is for training a recurrent neural network model. It is also for classification. Therefore, each sample is labeled, too. To meet the architecture requirement, each sample is voltages per node. In other words, they are voltage traces introduced in the previous section. The voltage grid trace data set is decomposed into voltage traces to form this data set. In addition, all the nodes responsible for the emergency are preserved and labeled as positive. Other ”normal” nodes are randomly discarded. The final data set keeps the positive to negative ratio to 1:3.

The data set has a shape of [samples, trace]. In addition, it has a separate label vector with a label for each sample.

4.6 Extra Contribution and Unsolved Challenges

Apart from the automation scripts, another contribution of the proposed toolchain is the per-CPU-cycle workaround for Gem5. Although Gem5 is per-CPU-cycle accurate, it does not fully support per-CPU-cycle output. Without modification, the per-CPU-cycle output from Gem5 would be corrupted. The proposed toolchain provides a customized simulation script for Gem5 that can bypass this limitation. A collect-and-pause sampling scheme achieves the workaround. For instance, it collects 5000 cycles of continuous data and then pauses for another 5000 cycles. This process repeats until the end.

The proposed toolchain facilitates the workflow and provides templates for quick configurations. Unfortunately, it cannot complete the configuration for the user. Manual configuration leaves a glaring issue unsolved. For Gem5, the configuration mostly defines the CPU of interest. The Gem5 configuration requires python programming, and it is relatively hard to discover the logic bug of programmed CPU architecture.

For VoltSpot, the most challenging part is knowing the floorplan for interested CPU. Many interesting CPUs do not have floorplans available to the public. Most open-sourced CPU designs are RISC-V architecture based which is not fully supported by Gem5.

CHAPTER 5
Regression-Based Methods

5.1 Introduction to Two Improved Methods

In this chapter, we are going to explain two regression-based prediction methods. They are similar to the methods for predicting thermal emergency on CPU. Unanimously, they use linear regression to predict the exact voltages in the future. We check if any prediction violate the preset threshold to convert it to a binary classification problem. The two methods differ in the way they select the optimum sensor locations. Group Lasso method takes advantage of the sparsity created by group Lasso regression to select sensors. Instead of regression analysis, Eagle-Eye maximizes the probability of a sensor catching an emergency. Note that regression-based methods completely separate the selection and the prediction problems and handles two problems with different algorithms.

5.2 Correlation Guided Group Lasso with Grid Search Cross Validation

In this section, we propose an improved version of group Lasso algorithm for sensor selection, which was introduced in 3.2.2. The improvement is mainly for automating the original algorithm. In this section, we will focus on the implementation of our modified algorithm. Before that, let's review the key parameter and steps in the original group Lasso algorithm.

The original group lasso algorithm select a subset of nodes C from candidate node set S to achieve maximum prediction accuracy. To achieve this goal, it optimize under constraint for a weight matrix B. The tightness of the constraint is manually defined by t. Matrix B establishes a linear relationship of voltages between nodes in candidate set (X) and all other nodes on the grid (Y). Each column in X and Y contains voltages of corresponding sensors sampled over time. The relationship between X and Y is formulated as $Y = BX$. The columns in B are the importance vectors for corresponding sensors.

The importance index of individual candidate node is evaluated by taking the norm of its importance vector in B. For selecting the best nodes, a manually selected threshold T is introduced, where any node with importance index higher than the threshold is selected.

5.2.1 Discussion of Original Algorithm

One serious drawback of the original group lasso algorithm is the difficulty of selecting t and T. We found those 2 parameters are problem dependent. There are three major concerns which complicate their selection.

1. The sensor ranking threshold T is S dependent. Even for the same training data, if the size of the candidate node set S changes, i.e., the area available for placing the sensors are changed, T would be different. It is usual to limit sensor placement to certain locations on CPU. For example, consider designing a dual core CPU with a L2 cache for each core, one may put voltage sensors on L2 caches for each core at first but later decided to put sensors only on 1 of 2 L2 caches. For the original algorithm, designer has to manually re-calibrate parameter T. As Shown in figure 5–1, both the maximum and the average importance factors of sensors are approximately doubled when the size of S decreases from 50% to 20% of the total die area. Designer has to figure out the threshold T case by case. This represents poor transferability.
2. The metrics for choosing t is limited. The only two constraints being considered in [23] are the number of sensors and "relative error," which is the accuracy for prediction. The original paper showed a positive correlation, within a small range of t, between the number of selected sensors and the accuracy of prediction. Therefore, they suggest the selection of t is a balance between accuracy and the cost of sensors. However, there is no quantitative upper or lower boundaries. Without mathematical tools to set the parameter t, balancing becomes an art of experience.
3. No quantifiable relationship between t and the number of sensors. It is because the number of sensors is determined by both t and T. The conundrum here is that it is

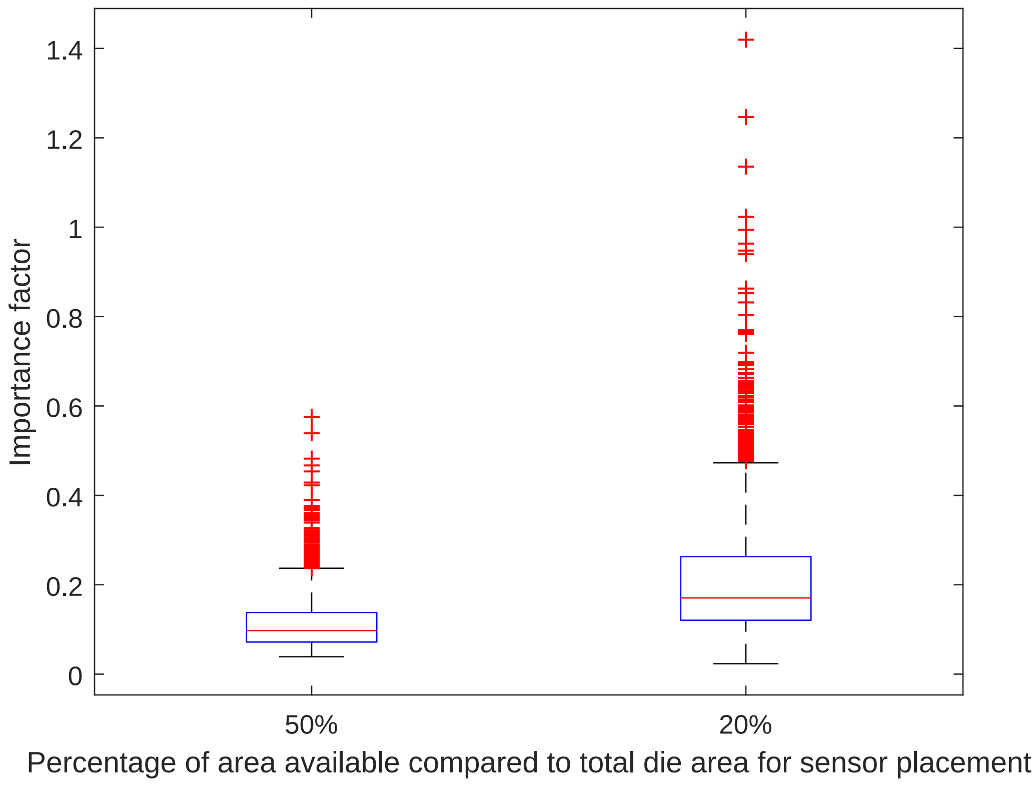

Figure 5–1: A demonstration of T variation with different S. The distribution of importance factors changes dramatically when S changes. As a result, designer has to find the optimal T case by case.

unclear how to reach a given number of sensors weather by relaxing the regulation t during optimization or lowering the quality factor T after optimization.

5.2.2 Clustering, Cost function and Cross Validation

In section 3.2.2, the main issue about the original algorithm is its poor transferability. A suitable parameter for one problem can be totally off for another. To solve this problem, we introduce three changes for standardizing the algorithm to minimize the effort of applying it to different applications. Those changes are:

1. Elimination of threshold T. We choose sensors from the most important to the least. In addition, We apply a 2-means cluster algorithm to the importance factor, and the cluster with higher importance is the upper bound of available sensors.

2. Addition of cross-validation for searching the t with the lowest loss. Theoretically, cross-validation can also be applied to the old algorithm. However, in our case, we don't have to train a prediction model to evaluate the selection, which essentially half the training time.
3. Adoption of a loss function for sensors. Different candidate sets can be evaluated by how much information they provide.
4. Optimizing by the average voltage grid instead of stacking samples. In the original paper, the target nodes Y and candidate nodes X for group Lasso optimization have the shape similar to $m \times n$, where n is the number of samples, and m is the number of nodes in the corresponding set. We take the average of all samples and result in a shape of $m \times 1$. This change has been applied to both sensor selection and prediction models.

Those changes substantially minimize the human expertise required for applying the algorithm. For the new algorithm, we cross-validate the group lasso optimization with various t values. During the cross-validation, sensor locations will be selected by the user-specified number and 2-means clustering, and the new loss function will evaluate the selection. Consequently, the selection with the lowest loss will be the final choice. Note that we can also apply cross-validation to the original algorithm to find a decent t. However, this approach would require training a prediction model in each validation iteration because the original algorithm does not have any metrics to evaluate the sensor quality other than the final prediction accuracy. Theoretically, the average operation should make the averaged data point falls right onto the estimated regression line. A quick proof is provided below. The main benefit of averaging is the greatly reduced training time, which makes the cross-validation over many t's practical.

Consider a general optimization objective function min $Y - BX$, where Y is a $m_1 \times n$ matrix and X is a $m_2 \times n$ matrix. Let $x_{i,j}, y_{i,j}$ denote the element in ith row and jth column in X and Y, respectively. Let $\overline{Y} = [\overline{y_1}, \ldots, \overline{y_{m1}}]^T$, $\overline{X} = [\overline{x_1}, \ldots, \overline{x_{m2}}]^T$, and ϵ be the error. In addition, the optimization of stacked samples yields B, and we have:

$$\begin{aligned}
\overline{y_{m1}} &= \frac{1}{n}\sum_{j}^{n} y_{m1,j} \\
&= \frac{1}{n}\sum_{j}^{n}(\sum_{i}^{m_2}\beta_{m1,i}x_{i,j} + \epsilon) \\
&\text{The average regression error for all samples is 0} \\
&= \sum_{i}^{m_2}\beta_{m1,i}\frac{1}{n}\sum_{j}^{n}x_{i,j} \\
&= \sum_{i}^{m_2}\beta_{m1,i}\overline{x_i}
\end{aligned}$$

This proves that the average $\overline{Y}$ and $\overline{X}$ is on the regression line.

Those changes would enable a group-lasso-based algorithm to automatically find a decent t as well as fine-tuning of t afterward. On top of that, it also offers the ability to evaluate the candidate set S before any prediction model applied, which effectively decouples the process of selecting sensors (feature selection) and selecting models. Also, the implementations of those sub-modules are introduced in the rest of this section, followed by a discussion about alternative implementations.

The proposed loss function essentially averages the correlation matrix of the chosen sensors set. However, calculating the average is not a straightforward process. 2-means clustering does not guarantee a constant size of clusters. Hence, the size of the candidate set will vary. So does the size of the correlation matrix. Once the Pearson correlation matrix is calculated as eq.5.1, the correlation matrix will be symmetric with all 1's a diagonal. Besides, the size of the correlation matrix will vary by the different ts. To reasonably compare different correlation matrices, only the lower triangular section of the matrix is considered for the following transformation. To further reduce the bias, the considered coefficients from the correlation matrix are transformed by Fisher z transformation, averaged and transformed back[8].

$$\rho_{i,j} = \frac{\text{cov}(c_i, c_j)}{\sigma c_i \sigma c_j} \tag{5.1}$$

where ρ is the correlation coefficient between voltage traces on node c_i and c_j; C is the selected node set and $c_i, c_j \in C$; cov(*) denotes the covariance and $\sigma*$ denotes the standard deviation.

The Fisher z transformation and its inverse can be described as:

$$z_{i,j} = \frac{1}{2} \ln\left(\frac{1+\rho_{i,j}}{1-\rho_{i,j}}\right) = \text{arctan}\left(\rho_{i,j}\right) \tag{5.2}$$

$$\rho_{i,j} = \frac{\exp\left(2z_{i,j}\right) - 1}{\exp\left(2z_{i,j}\right) + 1} = \tanh\left(z_{i,j}\right) \tag{5.3}$$

Discussion:

1. Correlation describes the similarity of two signals. For selecting nodes, we want to harvest as much information about the grid as possible. Therefore, the lower the correlation, the more extra information we would get. This idea also can be seen in the heat sensor selection [6].
2. Apart from correlation, mutual information can also be used to measure the similarity of signals. However, multivariate mutual information has not been fully understood yet, which makes it less favorable than correlation.
3. The proposed loss function can be solely applied to select sensor placement, which would maximize the information gained from a certain number of nodes. Its selection capability is the main reason for choosing it as a loss function for cross-validation, where it serves as a refinement of many candidate sets produced by various ts.

5.2.3 Overall algorithm

In this section, we recap and formalize the cross validated group Lasso algorithm. Before proceeding to the pseudo-code, we introduce one last alternation on the original algorithm. We optimize the Lagrange function of the group Lasso instead of the original optimization problem. The Lagrange function of group Lasso is given by:

$$\|Y - BX\|_{fro}^2 - \lambda \sum \left\| \hat{\beta}^k \right\|_1 \tag{5.4}$$

where $\hat{\beta}^k$ is the kth columen of B.

Consequently, instead of the optimizing with various ts, we solve for various λs. The major difference is the magnitude of two parameters are dramatic, where λs are usually between 0 and 1 but t can be arbitrarily large.

Algorithm 3: Group Lasso with Cross Validation for Sensor Placement

```
Input: S⟵ Candidate node set,   P⟵ noise margin,   N⟵ number of sensors
       to be placed,   D⟵ voltage grid over time
, Output: C⟵ selected node set
begin
    Separate D into $\mathbf{V}$ and $\mathbf{E}$
    Average all samples in $\mathbf{V}$ and $\mathbf{E}$, get $\overline{\mathbf{V}}$ and $\overline{\mathbf{E}}$
    $\mathbf{Y}$ = standardize($\mathbf{V}$)
    $\mathbf{X}$ = standardize($\mathbf{E}$)
    Initialize $\vec{\lambda} = \{\lambda_0, \lambda_1, \ldots, \lambda_r\}$ covering a range of reasonable $\lambda$s for group lasso
      optimization
    for $\lambda_r \in \vec{\lambda}$ do
        Solve group lasso optimization by eq. 3.8 with $t_r$
        Calculate importance for all nodes $\vec{I_r}$ by eq. 3.9
        Perform 2-means clustering on $\vec{I_r}$, let $C_R$ be the cluster with higher mean
        $C_r$ is the N most important nodes in $C'_r$, if $N > \|C_R\|, C_r = C_R$
        Calculate the correlation matrix $M_r$ of time-series voltage trace of nodes in
          $C_r$.
        Calculate the Fisher z transformation of the lower triangular section of $M_r$
          and denote the average as $a_r$.
        Calculate the inverse Fisher z transformation of $a_r$ and denote the result as
          $s_r$.
    end
    $r' = \arg\min_r([s_1, s_2, \ldots, s_r])$
    C = $C'_r$
end
```

5.3 Segmented Eagle-Eye

Eagle-eye is very effective for optimizing its loss function. In a grossly simplified prescriptive, it chooses locations where violations most likely to happen in the simulation data. We implemented this algorithm mainly for comparison reason. Additionally, a subtle change has been made for generalizing the algorithm for practical use.

One particular issue about the original eagle-eye algorithm is ”hot-spot clustering.” Our observation is that violations tend to happen more likely in some hot-spots rather than other areas. For example, in one experiment, most violations happen in the FPU cache are shown in fig. 5–2. If we directly apply the original algorithm, due to its greedy nature, most sensors will concentrate on hot-spot (FPU) and other integrated circuit blocks (IC) risk not having any sensor. Note that simply increasing total sensor number N is not a solution, because violations in hot-spot also have higher frequencies than the violations in other spot and therefore gains selection priority. As a result, one has to choose a large N to reach out to other IC blocks and inevitably with many sensors placed in hot-spot. This approach beats the purpose of selecting sensors.

Our implementation divides the IC area by IC blocks defined in the floorplan and applies eagle-eye to each segment individually. In this way, one can choose the desired number of sensors for each IC block. The desired number can be quickly decided either by domain knowledge or by setting it to proportionate to the violation frequency in each block.

5.3.1 Floorplan-Based Segmentation

In this section, we demonstrate the clustering effect and compare the outputs from the original eagle-eye and segmented eagle-eye.

As shown in fig.5–2, the original algorithm places sensors only on the FPU1 and nowhere else. The segmented placement algorithm randomly spreads the sensor to other user-controlled areas while keeping 2 sensors at the original cluster.

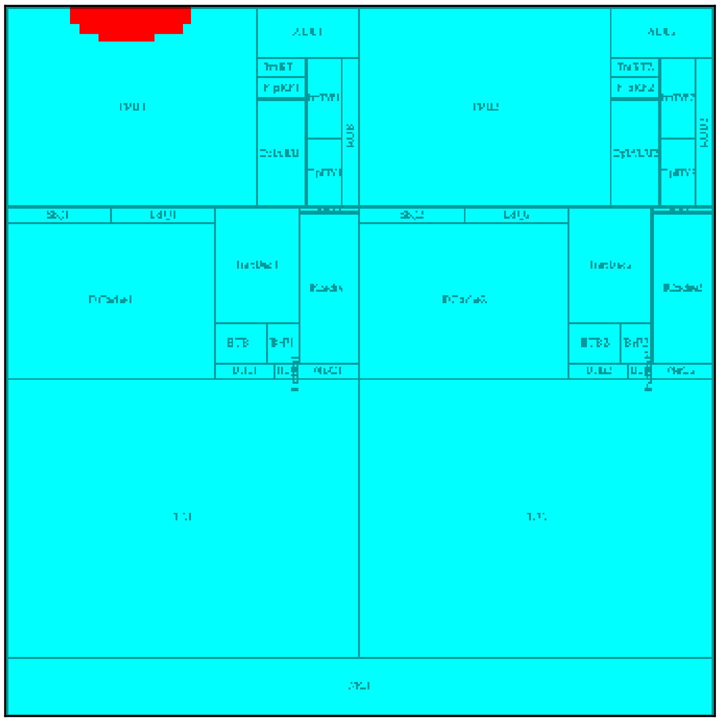

Figure 5–2: Sensor selection from original Eagle-Eye algorithm

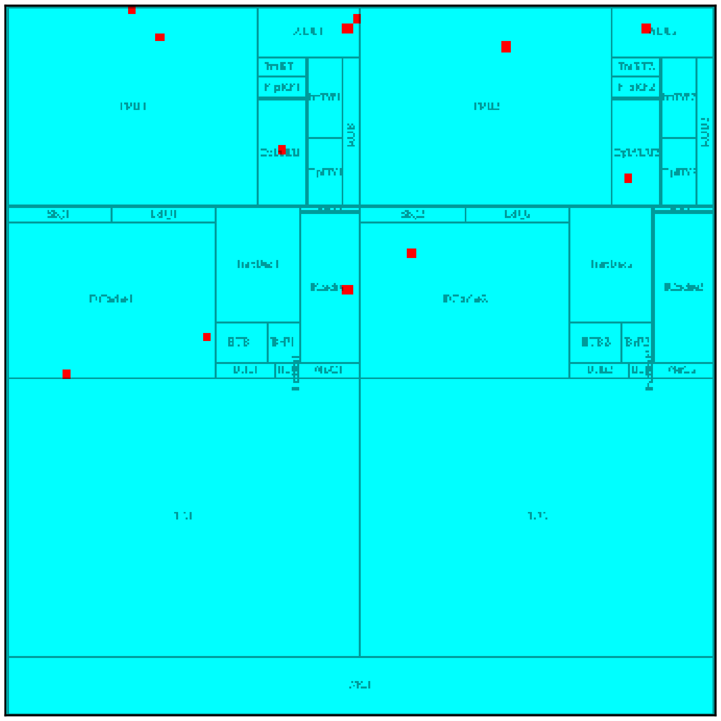

Figure 5–3: Sensor selection from segmented Eagle-Eye algorithm

5.4 Prediction Models

The prediction models for voltage/thermal emergencies are similar. They both require capability to prediction global extrema without complete spacial information. Many prediction models in the literature tries to capture the relationship between selected nodes and all other nodes with linear relationship [7, 38, 20]. This section presents a common regression model. Both group Lasso and Eagle-Eye use this algorithm to handle the prediction problem.

5.4.1 Linear Regression

The regression model solves an unconstrained OLS problem. In this case, we assume the voltage of nodes without sensor is a linear weighted sum of nodes with sensors that has been formulated in 2.2.1. To get the weights, we proceeds to solve:

$$\hat{\beta} = \arg\min \left\{ \sum_{k=1}^{M} \left(y_k - \hat{\beta}^k x \right)^2 \right\}$$

where $\hat{\beta}^k$ is the unique weights for target node y_k; $\hat{\beta}$ is the collection of all the weights; x are all the selected nodes with sensors.

For a regression model, the direct output of this prediction model will be the predicted voltage values for all target nodes. To find whether an emergency occurs, we examine if the lowest and the highest predicted voltage stays in a pre-defined voltage range.

CHAPTER 6
VoltNet

6.1 Introduction to Deep Learning Model: VoltNet

In this section, we propose a deep learning algorithm, VoltNet. We hope our work can inspire more deep learning applications in this field.

VoltNet is a deep neural network that solves both selection and prediction problems together. Instead of two independent problems, VoltNet considers the prediction as the feedback to optimize the sensor selection through backpropagation during the training phase. Related to the group Lasso, VoltNet select sensors by introducing sparsity to the weights for each sensor. For VoltNet, the sparsity comes from pruning. The prediction is a natural result from the trained model.

A concern we had when experimenting with previous algorithms is that it is extremely difficult to predict future based on a single freshly sampled data. Generally, good prediction comes from legitimate utilization of history. We expect a model utilizing history data could make better predictions thanks to the additional information.

So far, all the introduced sensor placement algorithms act like a feature selection process. The prediction of emergency relies on an additional linear regression model. This has two problems. First, intuitively, the voltage in the future likely does not have a linear relationship with current voltages. Second, the training of prediction model gives no feedback to the parameter selection of placement algorithms. The quality of the result purely depends on trial-and-error attempts regardless whether it is automated or not. A nonlinear feedback-based model is needed.

When training the two introduced algorithms, Eagle-Eye and Group Lasso, it is mandatory to have voltage traces from all nodes on the grids. Because we want the emergency prediction to be global on the CPU die and regression model needs global information for that. To capture the emergency voltage physically, a very large, if not all, area of the CPU has to be sensed. It is simply impossible. It means those algorithms only applies to simulation data. Consequently, it is impossible to apply them to real voltages. The gap between simulation and reality is wide open.

To solve those problems, we propose a deep neural network model, VoltNet.

6.1.1 Overall Description of VoltNet

As shown in fig.6–1, VoltNet combines time distributed BLSTM and MLP architectures. In time distributed BLSTM layer, a single BLSTM model is responsible for processing all the nodes provided. We interpret that BLSTM is trained to predict the probability of emergency per node and the MLP processes all the probabilities per node and predicts the overall probability of emergency. There are three training phase for VoltNet. In the first training phase, a BLSTM model is trained to process single voltage trace. Since then, the BLSTM will be frozen and remain unchanged in the following sessions. In the second training phase, the whole model is trained to predict if an emerngy would happen. The BLSTM model is applied to all nodes available by time distributed layer. The results are passed to MLP layer with pruning. The pruning yields the sensor selection. In the third phase, the whole model is trained again with the data size changed to match the selection so that only the selected nodes are considered.

BLSTM model is designed specifically to handle time-series data. The idea using one BLSTM for all nodes is inspired by the work in [12], where they showed that a small amount of CPU operations are responsible for the most voltage violations. Hence, we expect the voltage trace for violation could be summarized to a few patterns, too. A single BLSTM should be capable of handling that.

The benefit of having only one BLSTM for all nodes is the boosted technical and architectural flexibility. For technical flexibility, it dramatically expands the training data set. Emergency is scarce. Some nodes may never have an emergency during the simulation. An one-for-all BLSTM is trained with all traces from every node combined.

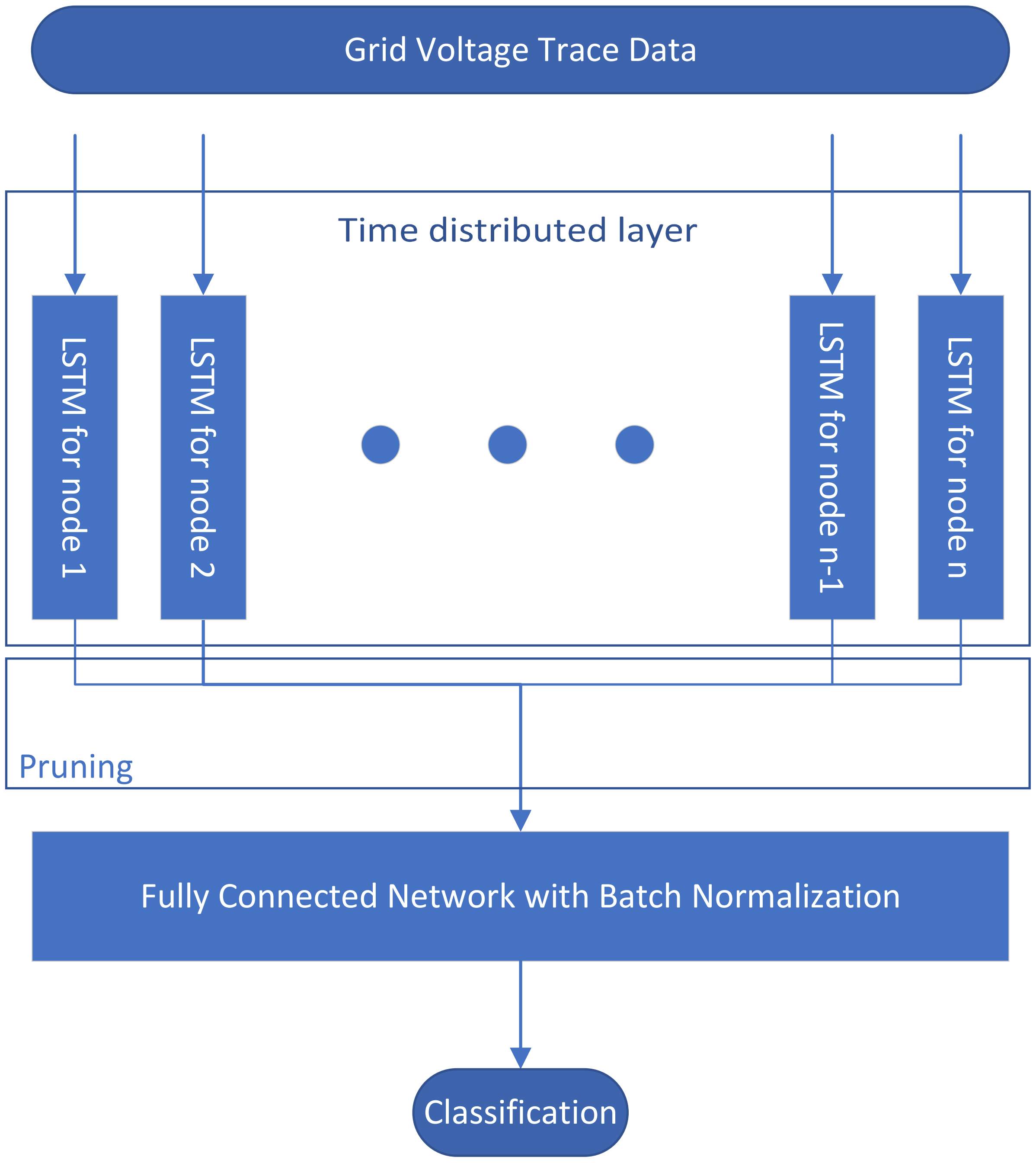

Figure 6–1: The VoltNet architecture for sensor placement

It solves the problem easily while enjoying an accuracy improvement thanks to the increased training data. In addition, more extensive training set allows deeper structures without overfitting. Moreover, a deeper structure introduces a higher level of abstraction [14]; therefore, better information extraction. For architecture flexibility, it allows the model to handle both simulation data and physical data alike. The source for physical voltage data is limited because there are only a few voltage sensors on the CPU. The training mixes voltages traces so there is no restriction on the source of the voltage trace. It makes no difference if the training set comprises a full grid or a portion of the grid. More importantly, this is a classification model which means exact emergency voltage is unnecessary. The training can proceed as long as emergencies are detected somehow, i.e. by finding unexpected CPU operations.

The sensor selection is produced by pruning the weights of a special MLP during the second training phase. The effect of pruning is comparable to that of group Lasso, it creates sparsity in the weight matrix. Instead of the B in group Lasso, we choose sensors based on the weight matrix of pruned layer. The fundamental difference between the two algorithms is the feedback. In VoltNet, a weight is pruned if it is too small. All wights are updated by backpropagation to maximize the prediction accuracy. The nodes are selected while keeping the accuracy maximized.

In the following section, we will describe each layer in detail.

6.1.2 Deep BLSTM Voltage Trace Analysis

This section discusses the time distributed BLSTM layer. The time distributed part is basically a wrapper to apply the BLSTM to every node. With this trick, all nodes can be processed together to take advantages of parallel computing.

The most important part is the node BLSTM model. It takes voltage trace as input and is trained to predict the probability of emergency. This is achieved by setting the output neural to a single sigmoid neuron.

Table 6–1: Coarsely tuned hyper parameter for node BLSTM model

Hyper parameters	Value
Length of input voltage trace	50 CPU cycles
Number of cells in BLSTM	50
BLSTM dropout	0.4
Number of SELU neurons	32
Number of output sigmoid neurons	1
Batch size for training	128
Epochs for training	20

The architecture of proposed node BLSTM is a deep residual bidirectional LSTM network. As shown in fig. 6–2, it contains 3 residual blocks connected to shallow SELU layer and batch normalization layer followed by an output layer of a single sigmoid neuron. The residual block has two components. The first one is a bidirectional LSTM layer. The other one is an additive skip connection layer that adds the input of the previous LSTM with the output of the LSTM. For stacked bidirectional LSTM, additive skip connection is chosen over the concatenate skip connection.Because, for stacked LSTM, the hidden state of previous layer becomes the input of the next. Bidirectional LSTM doubles the hidden state, which approximately doubles the input space of the next layer. And we want to avoid further increase the input space which may increase the training time.

One major improvement over the group Lasso and Eagle-Eye algorithms is that Volt-Net can be applied to the data gathered physically from the CPU. For a physical CPU, the sensors are already fabricated on the IC. Given the means to detect the occurrence of emergency is available, only the phase 1 and phase 3 training are necessary to train the VoltNet. Because the time distributed layer can handle any number of nodes it does matter if the input is a grid of nodes or a limited number of nodes. In comparison, the group Lasso algorithm require the exact voltage of emergency which is much harder, if even possible, to acquire.

We didn't fine tune the model to its best possible performance. But the following coarsely tuned hyper parameter, provided in table 6–1, yields decent results.

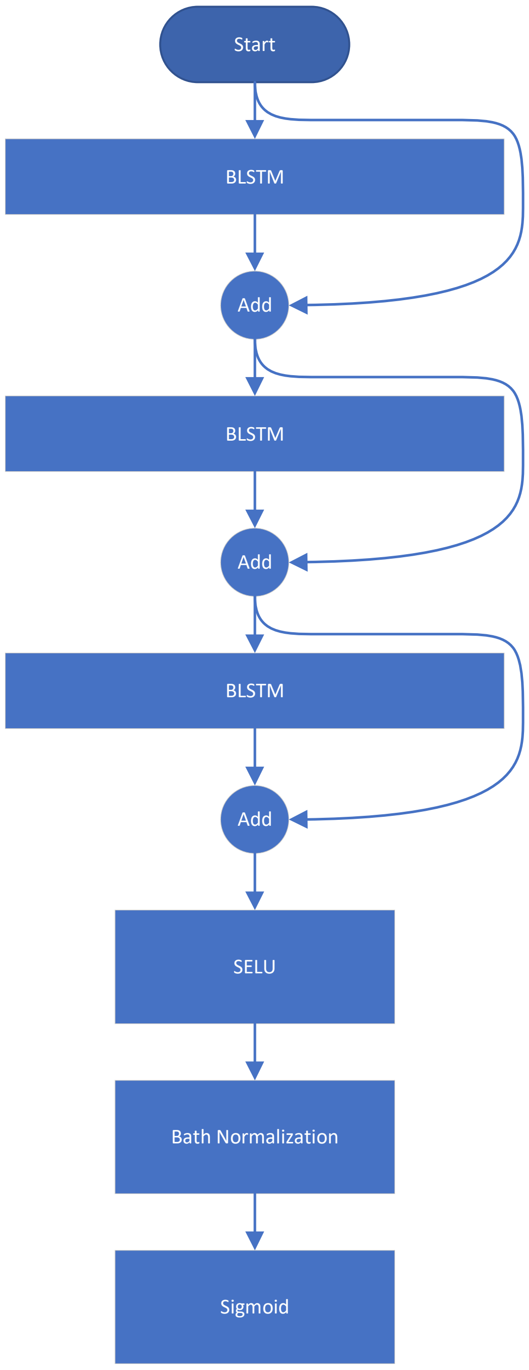

Figure 6–2: The architecture of node BLSTM

To train this model, we use the voltage trace data set introduced in chapter 4. The first 10,000 samples are used to train, then the next 3,000 samples are used for validation and the following 5,000 samples are used for testing.

6.1.3 Three Phase Training with Pruning

The pruning enables the sensor selection in VoltNet. As stated, there are totally 3 training session to train the VoltNet. In the previous section, we introduced the first stage which trains the BLSTM. The remaining stages are introduced in this section.

The second performs the pruning. The third completes the training. In the second session, the MLP layer after the time distributed layer is different from that of the third session.

The second stage is pruning. It forcibly sets insignificant weights to zero during the training. In this stage, we make a single neuron as the final layer of the VoltNet, which is architecturally different from the VoltNet in the third stage. As seen from group Lasso algorithm, if we have more than 1 weight representing the importance of a node, then many nodes would have a non-zero importance factor. This leads to a mandatory threshold. To avoid this, the MLP in the second training phase only has 1 layer and 1 sigmoid neuron. This setup gives exactly 1 weight for each node. If any weight is set to zero, that node is automatically discarded. As a result, sensor selection is the process selecting all nodes with non-zero weights and there is no need for a threshold. Additionally, it gives direct control over the number of selected sensors. During the pruning, one can specify the desired sparsity and the the number of sensors is controlled by the sparsity. Given a grid of 100 nodes, we can select 5 nodes by setting the sparsity to 0.95.

Note that the objective function of VoltNet is different from that of the group Lasso method. The weight update in neural network is carried out by backward propagation. The update is proportionate to the discrepancy between expected classification and the prediction. Therefore, the objective function can be interpreted as prediction accuracy.

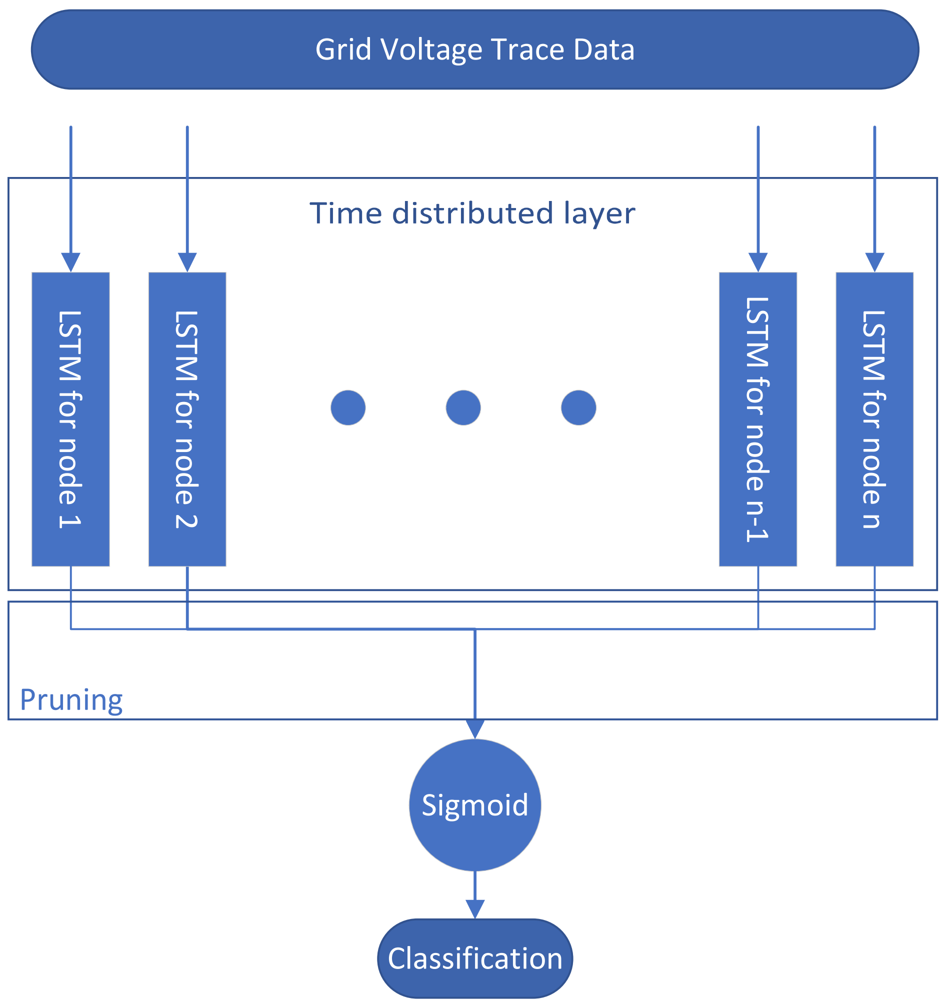

Figure 6–3: Architecture for the second training phase

In comparison, the original group Lasso method also tries to maximize the prediction accuracy but only indirectly. Moreover, the modified group Lasso method maximize the average correlation coefficient of selected sensors, which takes a different route.

In the second training phase, the architecture of the VoltNet is shown in the fig.6–3

After the second training phase, the sensor selection completes. The VoltNet will adjust its size to match the size of selected sensors and retrain. In the third training, the MLP has 3 layers. The first layer has SELU as activation function and initiated with 0 means and 1/34 standard deviation. It is followed by a bathnormalization layer and then the last layer only has 1 sigmoid neuron. We believe the extra layers help reducing the

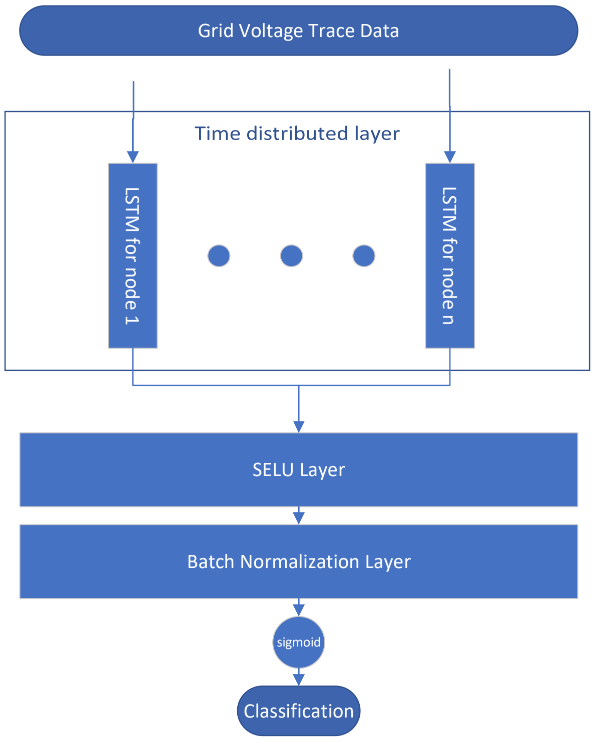

Figure 6–4: Architecture for the third training phase

noise produced by the previous node BLSTM and performs better than a single neuron. No model can predict 100% correctly, therefore the incorrect results from node BLSTM act as noises. It is a legit concern that the optimal selection for VoltNet stage 2 is different from that of stage 3. However, the modification is small and we observe an increase of the performance. We conclude the concern is not threatening.

In the third traning phase, the architecture of the VoltNet is shown in the fig.6–4.

The hyper parameters for the MLP are provided below.

Phase	Hyper parameters	Values
2	initial sparsity	0.50
	final sparsity	0.98
	begin step	500
	end step	1500
	prune frequency	100
	batch size	32
	epochs	15
3	Number of neurons	64
	batch size	32
	epochs	15

Table 6–2: Hyper parameters for training the MLP layer

CHAPTER 7
Model Evaluation

7.1 Key Configurations for the Prediction Models

We are comparing 5 different models for their performance predicting emergency. It is very align those algorithms to a similar level so that the comparison makes sense. To compare them, we have to use similar amount of sensors, uniformed outputs and same amount of training. There are a lot of configurations to go through.

7.1.1 Notation: Marking Algorithms

There are totally 5 different algorithms introduced in this thesis. We denote group Lasso model as $f(x)$, Eagle-Eye model as $g(x)$ and VoltNet as $h(x)$. For the improved version of group Lasso and Eagle-Eye we denote them as $\bar{f}(x)$ and $\bar{g}(x)$, respectively.

7.1.2 Parameter: Time to Event γ

One common critical configuration of all models needs to be addressed here. Our models predict the occurrence of voltage emergency a few CPU cycles ahead of the time when the inputs are sampled. But the original implementation in [23] does not. The original implementation predict the emergency at the instant of sampling. Let $f(\cdot)$ be the prediction model, x_t, y_t be the input and output at time t respectively, then the original prediction model can be expressed as:

$$y_t = f(x_t) \tag{7.1}$$

To actually predict the emergency, we use multi-step ahead prediction. We define an integer γ or "time to event" to indicate how many cycles ahead is the prediction. The prediction model can be expressed as:

$$y_{t+\gamma} = f(x_t) \tag{7.2}$$

where $\gamma >= 1$

This is main contributor to the performance difference between the original paper and the result from this thesis. In this thesis, we have train models with prediction capability of 5,10,20,40 for all models and additional 0 capability for regression models. The addition 0 capability models are provided to compare with the set up in the [23].

7.1.3 Parameter: Sensor Count N

How many sensors do we need? We chose 50 sensors as the target number of sensors. From literature, this number is decent to produce good predictions. Unfortunately, We cannot meet the target perfectly because some algorithms have very weak control over sensor count. We did successfully manage to control sensor count to a range of 40 - 60. In the following, We list the technical difficulties for controlling sensor count for each method.

For Eagle-Eye algorithm family, the number of sensors is bounded by the training data. The selected sensor count will not exceeds the the number of emergency nodes in the data set, due to the nature of the algorithm. To maximize the number of sensors, we have to use all but the last 5000 CPU cycles of data for sensor selection.

For Group Lasso algorithm family, the difficulty is to select fewer sensors but not so few that the population no longer makes sense. In our experiment, the simulated CPU has 5776 nodes. Ideally, we want groups Lasso introduce as much sparsity as possible so that the impact of the "manual" selection is minimized. "Manual" selection cannot guarantee the optimality regardless whether it is carried out by a threshold or a top-down selection. By setting λ between 0.65 to 0.75, the algorithm gives about 400 selections and we have to manually filter some of them. However, if λ grows, the selection population quickly drops to single digit. For an optimization of a few thousands variables, it seems that a small increase of the regularization can constraint the outcome too much.

VoltNet has no problem controlling the selection size, thanks to the single layer design which enables direct translation of sparsity to the selection size.

We denote the sensor count as N.

7.1.4 Configuration: Comparable Training

In short, there is no way to ensure same amount of training for so many different algorithms. It is especially true for regression models and neural network models. The former option trains forever given a large training data, whereas the latter directly benefit from the increased size of the data set. Note that we use the average of data to train the regression model, it does not take much time at the cost of indistinguishable models given a large data set, i.e. models with prediction capability of 1 and 2 will be the same because the average will be the same.

Although we cannot train all models with the same amount of data, the limitation is not imposed by us. Those are their intrinsic limitations, which honestly shows their usefulness. What we can guarantee in this thesis is that we have used sufficient samples to reach their maximum capability. No significant improvement can be made by simply increase the amount of data for training.

7.2 Comparing Prediction Models

In this section, we are evaluating the performance of all models. We employ a few metrics from general to detail to evaluate the performance.

The data used in evaluating models is the test set from voltage grid trace data set introduced in chapter 4. There are equal amount of positive and negative samples in this data set. The occurrence of the voltage emergency is rare, therefore the samples in the data set have great temporal distance with each other, which means they are likely from different parts of the program. The sample balance and temporal distance make it ideal for evaluating performance in general.

7.2.1 Evaluation by Statistical Metrics

Before we convert regression models to classification models, it is worthwhile to examine the regression performance. To evaluate the regression result, we compare the statistical characteristics of predicted voltages and true voltages. The statistical metrics used are mean squared error and averaged R^2. The mean squared error (MSE) is calculated by following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i^t - y_i^p)^2 \tag{7.3}$$

The R^2 is calculated per node and then averaged with the same weight. The R^2 for a single node is calculated as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i^t - y_i^p)^2}{\sum_{i=1}^{n}(y_i^t - \hat{y})^2} \tag{7.4}$$

We show the statistical results in table 7–1. Note that this is for evaluating regression models so VoltNet is not included. Generally, as target prediction capability increases, the regression result gets worse as expected. For all algorithms, the best result is always the model with target 0 prediction capability, which only infers if there is an emergency at current time. As the target prediction capability increases, the mean squared error tends to increase and R^2 tends to decrease with a few exceptions.

If we consider the implication of the result, it is easy to conclude that linear regression models are not suitable for voltage emergency prediction, because only the none-prediction models can achieve a R^2 above 0. R^2 has to be positive to be considered not a failure.

Given the result presented, there is no solid conclusion to determine the performance of each sensor placement algorithms as the performances are marred by the linear prediction prediction models.

7.2.2 Evaluation by Prediction Accuracy

The prediction accuracy further proves the conclusion reached by the previous discussion. None of the regression models are capable predicting the positive case in our data set. The accuracy for linear regression models are around 50% which is no better than random guessing.

VoltNet's accuracy decreases along with increasing target prediction capability, which is expected. As further down the future, the more difficult to predict.

Table 7–1: Table for Statistical Metrics

Models	γ	Mean_Squared_Error	R^2
$f(x)$	0	0.000204671	-10.57871652
$f(x)$	5	0.001279349	-217.8130268
$f(x)$	10	0.002255958	-512.1550495
$f(x)$	20	0.047019879	-10231.79673
$f(x)$	40	0.002901609	-895.0097982
$\bar{f}(x)$	0	6.15E-06	0.260527767
$\bar{f}(x)$	5	3.32E-05	-4.122490523
$\bar{f}(x)$	10	3.29E-05	-4.725554785
$\bar{f}(x)$	20	2.25E-05	-1.834347755
$\bar{f}(x)$	40	3.42E-05	-4.467721721
$g(x)$	0	1.02E-05	0.216498285
$g(x)$	5	3.08E-05	-3.551376279
$g(x)$	10	2.88E-05	-2.84174014
$g(x)$	20	2.18E-05	-2.031892097
$g(x)$	40	2.55E-05	-2.204592264
$\bar{g}(x)$	0	1.21E-05	0.206824691
$\bar{g}(x)$	5	3.14E-05	-3.536477787
$\bar{g}(x)$	10	3.05E-05	-3.466492877
$\bar{g}(x)$	20	2.36E-05	-1.949371465
$\bar{g}(x)$	40	3.57E-05	-4.471307947

Models	γ	Acc(%)
$f(x)$	0	28.52
$f(x)$	5	46.96
$f(x)$	10	40.82
$f(x)$	20	44.9
$f(x)$	40	52.68
$\bar{f}(x)$	0	28.82
$\bar{f}(x)$	5	50.48
$\bar{f}(x)$	10	49.96
$\bar{f}(x)$	20	49.96
$\bar{f}(x)$	40	49.96
$g(x)$	0	49.96
$g(x)$	5	49.96
$g(x)$	10	49.96
$g(x)$	20	49.96
$g(x)$	40	49.96
$\bar{g}(x)$	0	49.96
$\bar{g}(x)$	5	49.96
$\bar{g}(x)$	10	49.96
$\bar{g}(x)$	20	49.96
$\bar{g}(x)$	40	49.96
$h(x)$	5	90.3
$h(x)$	10	86.94
$h(x)$	20	73.02
$h(x)$	40	60.74

7.2.3 Recall and Precision of VoltNet

Apart form accuracy, we introduce recall and precision to evaluate VoltNet. The formula for recall and precision are:

$$Recall = \frac{TP}{TP + FN} \tag{7.5}$$

$$Precision = \frac{TP}{TP + FP} \tag{7.6}$$

where TP is ture positive, FP is false positive and FN is false negative.

Before we present the result, we would argue that the performance is underestimated. We trained the model as exactly we described in section IV. No fine-tuning, no tricks or heuristics used. Many of those can boost the performance greatly. We chose not to use them because we want the evaluation as objective as possible. If we put any effort into manually increasing the performance, it is impossible to ensure to put the same amount of effort into all configurations with various signal lengths and various prediction intervals. That been said, our provided parameters inevitably favors some configurations than the others. This accounts for most abnormalities in the performance.

For a successful prediction model, we want the input information as little as possible and the prediction as far ahead as possible. We call the time interval from now to prediction as to the prediction interval. Let the length of input voltage reading from one sensor be the signal length. Fig.7–1 shows a decreasing tendency in the prediction accuracy given an increasing prediction interval. The accuracy is the average from various signal length. However, there is an abnormal bump for the prediction inter with 20 CPU cycles. This is most likely because our coarse hyperparameter favors this particular setup. Fig.7–2 tells another story. The accuracy increases if the signal length increases. Similar to the previous case, the performance bump for trace length 20 is likely due to the hyperparameter. Both cases match our intuitive expectations.

Accuracy should not be the sole metric to evaluate performance. Another key factor is recall. It tells how many emergencies happened we failed to predict during the test. Ideally, we should never miss because the cost of recovering from a mistaken CPU operation is much higher than preventing it. Table 7–3 tells the performance is less than satisfactory, especially when the prediction configuration becomes more challenging. This indicates that the positive case in the training set is under-represented. Many standard balancing techniques could be used like weighted classes, to mitigate the problem.

The precision is shown in table 7–4 is a less critical factor to consider. Statically, there are far more negative cases than in positive cases. As a result, a small percentage of negative cases predicted wrong would result in a low precision regardless.

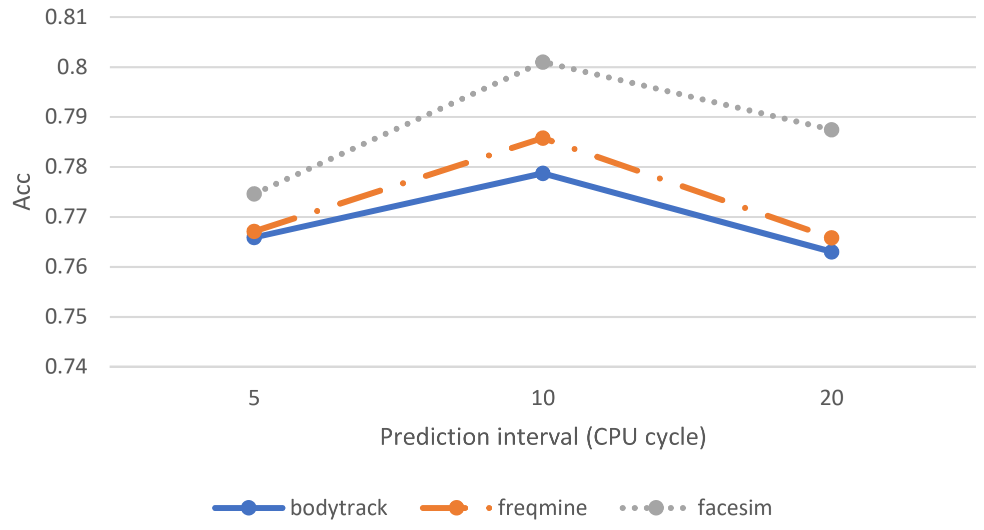

Figure 7–1: Accuracy slightly decreases as prediction interval increases

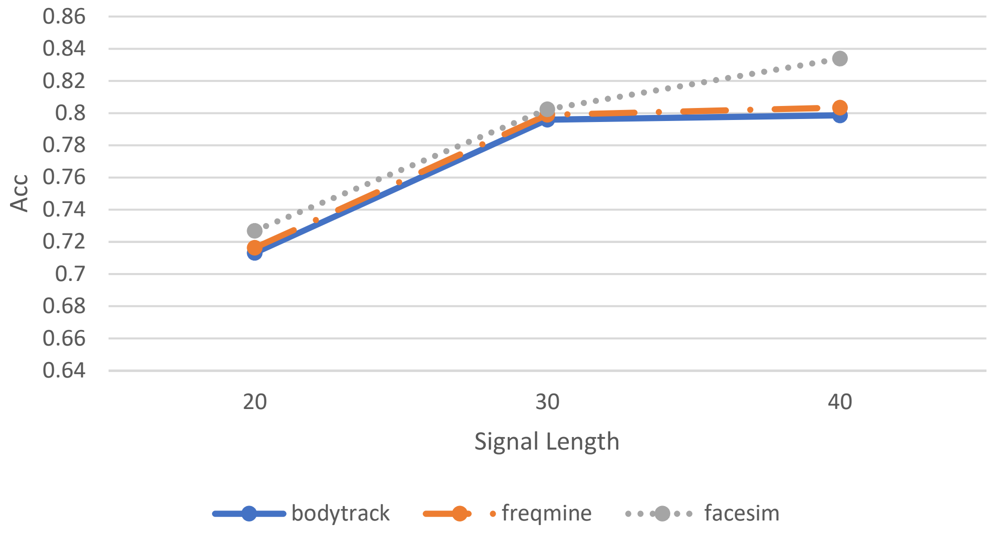

Figure 7–2: Accuracy greatly increases as signal length increases

Table 7–2: Accuracy for various configurations

Predition Interval	5	5	5	10	10	10	20	20	20
Trace Length	20	30	40	20	30	40	20	30	40
bodytrack2c	0.6032	0.861	0.8334	0.7985	0.764	0.7737	0.7377	0.7626	0.7888
freqmine2c	0.6048	0.8606	0.8359	0.8025	0.7719	0.783	0.7417	0.7643	0.7914
facesim2c	0.611538	0.858462	0.853846	0.801538	0.776154	0.825385	0.767692	0.772308	0.822308

Table 7–3: Recall for various configurations

Predition Interval	5	5	5	10	10	10	20	20	20
Trace Length	20	30	40	20	30	40	20	30	40
bodytrack2c	0.570439	0.741339	0.795612	0.599307	0.646651	0.670901	0.486143	0.562356	0.524249
freqmine2c	0.575141	0.737853	0.766102	0.567232	0.620339	0.622599	0.503955	0.524294	0.512994
facesim2c	0.583333	0.75	0.824074	0.62037	0.583333	0.62037	0.462963	0.398148	0.472222

Table 7–4: Precision for various configurations

Predition Interval	5	5	5	10	10	10	20	20	20
Trace Length	20	30	40	20	30	40	20	30	40
bodytrack2c	0.120782	0.355088	0.316345	0.237311	0.214231	0.227042	0.161985	0.196213	0.210771
freqmine2c	0.124602	0.35978	0.321023	0.239733	0.220128	0.230834	0.172201	0.193333	0.215268
facesim2c	0.120459	0.340336	0.342308	0.235915	0.203883	0.264822	0.170068	0.156934	0.226667

CHAPTER 8
Conclusion

This thesis have propose an automated toolchain for simulating voltages across the CPU die area. Those data are necessary to study the voltage sensor placement problem and voltage emergency prediction problems.

Overall, there are 3 new algorithms proposed in this thesis. For sensor placement problem, original group Lasso algorithm has received a new loss function along with many other changes. As a result, the new group Lasso can be applied to different CPUs with minor human intervention. In addition, we observed and solved sensor clustering problem for the original Eagle-Eye algorithm. We also proposed VoltNet as a machine learning algorithm that embedded the sensor selection process into the model training.

Finally, we proceed to compare the performance of different models. It turns out that the group Lasso family and Eagle-Eye family are severely bounded by the poor performance of the linear regression prediction model. No solid conclusion can be drawn to evaluate the placement algorithms. This finding invites new endeavours to find new prediction model for capturing the non-linearity of voltage trace. Furthermore, we have shown that VoltNet yields very promising results. With only coarse training, it reaches accuracy of 90.3% and 86.94% for prediction capability of 5 and 10 CPU cycles, respectively.

As for future work, a lot of directions can be pursued. One obvious uncharted domain is the non-linear models for group Lasso and Eagle-Eye. Many seems fit, probably few would work out. Another interesting path is applying VoltNet to physical CPU. As stated before, there is no theoretical limitation bars VoltNet from simulation to reality. After some literature research, another popular way to predict voltage emergency is not to look at voltage at all. It turns out that the CPU signatures are also a good information source to predict emergency. We wonder what would be the outcome if we apply VoltNet to a digital signal like CPU signature instead of analog voltage signal.

References

[1] wornbb/Dev-Toolbox: Toolbox for creating Gem5-McPAT interface.

[2] Sergey Bakin. *Adaptive regression and model selection in data mining problems*. PhD thesis, School of Mathematical Sciences, Australian National University, 1999.

[3] B K Bhattacharyya, A Levin, and Gang Huo. A semi-empirical approach to determine the effective minimum current pulse width (T) for an operating silicon chip. In *2007 International Power Engineering Conference (IPEC 2007)*, pages 922–927, 12 2007.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 1 2011.

[5] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1, 8 2011.

[6] Kun-Chih Jimmy Chen, Yen-Po Lin, Kai-Yu Chiang, and Yu-Hsien Chen. Correlation-graph-based temperature sensor allocation for thermal-aware network-on-chip systems. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 210–213. IEEE, 10 2016.

[7] Ryan Cochran and Sherief Reda. Consistent runtime thermal prediction and control through workload phase detection. In *Proceedings of the 47th Design Automation Conference on - DAC '10*, page 62, New York, New York, USA, 2010. ACM Press.

[8] David M. Corey, William P. Dunlap, and Michael J. Burke. Averaging Correlations: Expected Values and Bias in Combined Pearson r s and Fisher's z Transformations. *The Journal of General Psychology*, 125(3):245–261, 7 1998.

[9] Mohamed Elgamel and Magdy Bayoumi. Noise Analysis and Design in Deep Submicron Technology. *The Electrical Engineering Handbook*, pages 299–310, 1 2005.

[10] Yoav Goldberg. A Primer on Neural Network Models for Natural Language Processing. *Journal of Artificial Intelligence Research*, 57:345–420, 11 2016.

[11] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with Deep Bidirectional LSTM. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 273–278. IEEE, 12 2013.

[12] Meeta Sharma Gupta, Krishna K. Rangan, Michael D. Smith, Gu-Yeon Wei, and David Brooks. Towards a software approach to mitigate voltage emergencies. In *Proceedings of the 2007 international symposium on Low power electronics and design - ISLPED '07*, pages 123–128, New York, New York, USA, 2007. ACM Press.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. 12 2015.

[14] Michiel Hermans and Benjamin Schrauwen. Training and Analysing Deep Recurrent Neural Networks. In C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 190–198. Curran Associates, Inc., 2013.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[16] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks. 6 2017.

[17] Kyeong-Jae Lee, K. Skadron, and W. Huang. Analytical model for sensor placement on microprocessors. In *2005 International Conference on Computer Design*, pages 24–27. IEEE Comput. Soc.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[19] Yann LeCun, John S Denker, and Sara A Solla. Optimal Brain Damage. In D S Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[20] Adam Lewis, Soumik Ghosh, and N.-F. Tzeng. Run-time Energy Consumption Estimation Based on Workload in Server Systems. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, page 4, Berkeley, CA, USA, 2008. USENIX Association.

[21] S Li, J H Ahn, R D Strong, J B Brockman, D M Tullsen, and N P Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, 12 2009.

[22] Jinglan Liu, Yukun Ding, Jianlei Yang, Ulf Schlichtmann, and Yiyu Shi. Generative adversarial network based scalable on-chip noise sensor placement. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 239–242. IEEE, 9 2017.

[23] Xiaochen Liu, Shupeng Sun, Xin Li, Haifeng Qian, and Pingqiang Zhou. Machine Learning for Noise Sensor Placement and Full-Chip Voltage Emergency Detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):421–434, 2017.

[24] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 3 1982.

[25] Lukas Meier, Sara Van De Geer, and Peter Bühlmann. The group lasso for logistic regression. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(1):53–71, 1 2008.

[26] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. 11 2016.

[27] Vasilis F. Pavlidis, Ioannis Savidis, and Eby G. Frian. *Three-dimensional integrated circuit design.*

[28] Dabal Pedamonti. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. 4 2018.

[29] Juri Ranieri, Alessandro Vincenzi, Amina Chebira, David Atienza, and Martin Vetterli. EigenMaps. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 636, New York, New York, USA, 2012. ACM Press.

[30] Vijay Janapa Reddi, Meeta S. Gupta, Glenn Holloway, Gu-Yeon Wei, Michael D. Smith, and David Brooks. Voltage emergency prediction: Using signatures to reduce operating margins. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 18–29. IEEE, 2 2009.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 10 1986.

[32] Martin Saint-Laurent and Madhavan Swaminathan. Impact of power-supply noise on timing in high-frequency microprocessors. In *IEEE Topical Meeting on Electrical Performance of Electronic Packaging*, volume 2002-Janua, pages 261–264. IEEE, 2002.

[33] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[34] Robert Tibshirani. Regression shrinkage and selection via the lasso: a retrospective. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(3):273–282, 6 2011.

[35] Robert Tibshirani and Robert Tibshirani. Regression Shrinkage and Selection Via the Lasso. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 58:267–288, 1994.

[36] Varghese George, Sanjeev Jahagirdar, Chao Tong, Ken Smits, Satish Damaraju, Scott Siers, Ves Naydenov, Tanveer Khondker, Sanjib Sarkar, and Puneet Singh. Penryn: 45-nm next generation Intel® core™ 2 processor. In *2007 IEEE Asian Solid-State Circuits Conference*, pages 14–17. IEEE, 11 2007.

[37] Tao Wang, Chun Zhang, Jinjun Xiong, and Yiyu Shi. Eagle-Eye: A near-optimal statistical framework for noise sensor placement. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 437–443. IEEE, 11 2013.

[38] B Wojciechowski and J Biernat. Temperature prediction for multi-core microprocessors with application to Dynamic Thermal Management. In *18th International Workshop on THERMal INvestigation of ICs and Systems*, pages 1–6, 2012.

[39] Muhammad Nur Yanhaona. Architecture Implications of Pads as a Scarce Resource: Extended Results. 2014.

[40] Qixiang Zhang, Liangzhen Lai, Mark Gottscho, and Puneet Gupta. Multi-Story Power Distribution Networks for GPUs. pages 451–456, 2016.

[41] Runjie Zhang, Ke Wang, Brett H. Meyer, Mircea R. Stan, Kevin Skadron, Runjie Zhang, Ke Wang, Brett H. Meyer, Mircea R. Stan, and Kevin Skadron. Architecture implications of pads as a scarce resource. *ACM SIGARCH Computer Architecture News*, 42(3):373–384, 10 2014.

Index

KEY TO ABBREVIATIONS

CDMA: code-division multiple access 69

CTAN: comprehensive TEX archive network 69

IEEE: Institute of Electrical and Electronics Engineers, Inc. 69