



**MACAU UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**School of Computer Science and Engineering**

**Faculty of Innovation Engineering**

**Senior Thesis for the Degree of Bachelor of Science**

Title: A Dynamic Detection Approach for Oscillating Loss  
Problem in DNN Based on AUTOTRAINER

Student Name : LIU HE YUAN

Student No. : 1909853L-I011-0062

Supervisor : HUANG RU BING

May, 2023



澳門科技大學

創新工程學院

計算機科學與工程學院

理學學士學位畢業論文

論文題目: 基於 AUTOTRAINER 的一種針對 DNN 中損失震盪問題  
的動態檢測方法

姓 名 : 劉和源

學 號 : 1909853L-I011-0062

指導老師 : 黃如兵

2023 年 5 月

# Abstract

Deep neural networks (DNNs) are widely used in both research and industry. However, bugs such as Oscillating Loss can have a significant impact on the time and computation power required for training. While systems like AUTOTRAINER have offered an effective way to automatically detect and repair these bugs, there is still room for improvement in their detection methods. This paper focuses on enhancing the detection of Oscillating Loss in DNNs. The current approach taken by AUTOTRAINER involves periodically checking parameters during the training procedure, which can be time-consuming and inefficient. Through an analysis of a considerable amount of training data, including 50 epochs, 100 epochs, 125 epochs and 150 epochs. The author discovered that certain regions have a higher likelihood of experiencing bugs. To improve the detection of Oscillating Loss, the author introduces a software engineering approach by transforming the original constant value used for bug detection into a variable value. This allows for more frequent checks in high-risk areas and fewer checks in other areas, resulting in a more efficient and effective bug detection system. The proposed approach not only improves the software engineering principles of the system, but also enhances its system design and statistical capabilities. This research has the potential to benefit a wide range of industries that rely on DNNs for their operations.

**Keywords:** Deep Neural Networks; Oscillating Loss; Software Engineering; DNN bugs;

## 摘要

深度神經網絡（DNN）廣泛用於研究和工業領域。然而，像振盪損失（Oscillating Loss）這樣的錯誤對於訓練所需的時間和計算能力有巨大影響。雖然像 AUTOTRAINER 這樣的系統已經提供了一種有效的自動檢測和修復錯誤的方法，但它們的檢測方法仍有改進的空間。本文專注於增強 DNN 中振盪損失的檢測。AUTOTRAINER 目前採用的方法是在訓練過程中定期檢查參數，這可能是耗時且低效的。通過對大量訓練數據（包括 50、100、125 和 150 個疊代）的分析，作者發現某些區域更容易出現錯誤。為了改進振盪損失的檢測，作者引入了一種軟件工程方法，即將用於錯誤檢測的頻率的原始常量值轉換為可變值，並設定根據當前疊代次數變化而變化的函數，並設定根據當前疊代次數變化而變化的函數。這樣可以在高風險區域進行更頻繁的檢查，在其他區域進行較少的檢查，從而實現更高效和有效的錯誤檢測系統。提出的方法不僅改善了系統的軟件工程原則，還增強了其系統設計和統計能力。這項研究有潛力造福廣泛依賴 DNN 進行運營的各行業。

**關鍵詞：**深度神經網絡；損失震盪；軟體工程；深度神經網路缺陷；

# Table of Contents

Abstract.....	I
摘要 .....	II
Table of Contents.....	III
List of Figures.....	V
List of Tables .....	VI
Chapter 1 Introduction.....	1
1.1 Background and motivation .....	1
1.2 Thesis organization .....	3
Chapter 2 Data Preparation for Validation.....	5
2.1 Assumption.....	5
2.2 Data Preparation .....	5
2.2.1 <i>Environments</i> .....	6
2.3 CPU and GPU .....	7
Chapter 3 Data Analysis for Validation .....	9
3.1 Experiments.....	9
3.1.1 <i>Data Types</i> .....	9
3.1.2 <i>Data Loading</i> .....	9
3.1.3 <i>How to determine the condition of oscillating loss</i> .....	10
3.1.4 <i>Data Cleaning</i> .....	10
3.1.5 <i>Data Analysis</i> .....	10
50-epoch version .....	14
100-epoch version .....	16
125-epoch version .....	18

<i>150-epoch version</i> .....	21
Chapter 4    Dynamic Approach on the Checking Frequency .....	25
4.1    Assumption.....	25
4.2    Key Value.....	27
Chapter 5    Experiments on Dynamic Checking Frequency .....	28
5.1    Experiments.....	28
5.2    Overhead and Validation.....	31
Chapter 6    Conclusion .....	34
6.1    Conclusion.....	34
6.2    Future work .....	34
References.....	35
Appendix I Documents .....	36
Appendix II Project details .....	37
Resume.....	38
Acknowledgements.....	39

## List of Figures

Figure 1-1 The system design of AUTOTRAINER [2] .....	3
Figure 3-1 line graph on values for 150-epoch case .....	24
Figure 3-2 line graph on accuracy for 150-epoch case .....	24
Figure 4-1 relationships between $1/q$ and runtime overhead [1] .....	25
Figure 4-2 Oscillating Loss's detection delay [1] .....	26
Figure 4-3 Oscillating Loss's Occurrence and time for Dying ReLU [1] .....	27
Figure 5-1 Code Structure in Original AUTOTRAINER [2] .....	29
Figure 5-2 The procedure of the "getdynamicCF()" .....	31
Figure 5-3 time consumption before the detection of OL .....	32

## **List of Tables**

Table 3-1	Detail in each case .....	11
Table 3-2	Training Results and number of trainings. ....	11
Table 3-3	The distribution of buggy model and bug-free model with 50 epochs .....	15
Table 3-4	The distribution of the moment that the bug detected.....	16
Table 3-5	The distribution of buggy model and bug-free model with 100 epochs .....	17
Table 3-6	The distribution of the moment that the bug detected.....	18
Table 3-7	The distribution of buggy model and bug-free model with 125 epochs .....	20
Table 3-8	The distribution of the moment that the bug detected.....	21
Table 3-9	The distribution of buggy model and bug-free model with 100 epochs .....	22
Table 3-10	The distribution of the moment that the bug detected.....	23
Table 5-1	Parameters for the equipment.....	33



# Chapter 1 Introduction

## 1.1 Background and motivation

Various types of bugs such as Dying ReLU, Oscillating Loss, Exploding Gradient, Vanishing Gradient, and Slow Convergence are commonly encountered during the training of Deep Neural Networks (DNNs). [1] These bugs significantly reduce the efficiency of DNN training. Typically, researchers and engineers can only fix these bugs after the entire training process is completed. However, systems such as AUTOTRAINER [2] and DeepDiagnosis [3] help to detect and automatically fix these bugs. In this paper, the focus is on Oscillating Loss, and the author proposes an improvement to AUTOTRAINER that saves time and computing power. AUTOTRAINER periodically analyzes the parameters in the training model with a constant checking frequency. In this paper, a dynamic checking frequency approach is used to make the checking more efficient. After conducting experiments and validation, it has been confirmed that the hypothesis proposed by the author is feasible and can be utilized to predict the probability of encountering bugs at a specific stage during the DNN training process.

**Deep Neural Networks (DNNs):** Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone a light on sequential data such as text and speech. [1]

**Oscillating Loss:** It is inevitable for the loss value to go up and down during the training procedure. But if there are large changes without decreasing trend, the training may not converge in a very long time which should be enough for training the model. We refer to such a problem as oscillating loss (OL). [2]

**MNIST:** In this paper, the author mainly uses the MNIST for training and testing. The MNIST [4] database of handwritten digits [5], available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

**AUTOTRAINER:** AutoTrainer, a DNN training monitoring and automatic repairing tool which supports detecting and auto repairing five commonly seen training problems. During training, it periodically checks the training status and detects potential problems. Once a problem is found, Auto Tr a i n e r tries to fix it by using built-in state-of-the-art solutions. It supports various model structures and input data types, such as Convolutional Neural Networks (CNNs) for image and Recurrent Neural Networks (RNNs) for texts. Our evaluation on 6 datasets, 495 models show that AUTOTRAINER can effectively detect all potential problems with 100% detection rate and no false positives. Among all models with problems, it can fix 97.33% of them, increasing the accuracy by 47.08% on average. The [Figure 1-1](#) illustrates the structure of the AUTOTRAINER [2]. Which consists of the problem detection module (left) and the automatic repair module (right). The whole system starts by training a model with an initial training configuration and using the problem recognizer to monitor the training. When a problem is detected, the system will launch the automatic repair module trying to retrain the model with new settings until the training can finish without any problem (repaired) or a detected problem cannot be solved (failed). Notice that if there exist several problems, our system will attempt to solve the detected problems one by one in the order of exposure.

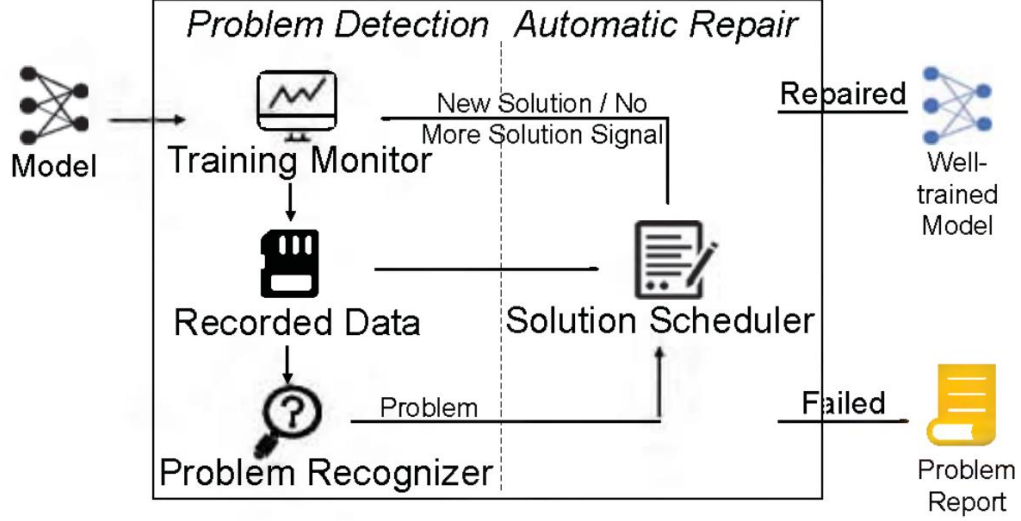


Figure 1-1 The system design of AUTOTRAINER [2]

In this paper, the author focuses on the modification on the Training Monitor in the AUTOTRAINER System, by making the checking frequency of the system dynamic, to decrease the time consumption on the periodical check in the training procedure corresponding to the Oscillating Loss issue.

## 1.2 Thesis organization

Chapter 2, Data Preparation for Validation, prepares the data which is used to valid the feasibility. The dataset is MNIST, select several different models to eliminate the influence from diversity of models and configure multiple versions of different number of epochs to Avoiding chance from experiments with a specific number of iterations.

Chapter 3, Data Analysis for Validation, analyze the data and valid the strategy works. Based on the data generated, conclude a general theory, and prove that the distribution of Oscillating Loss occurrence is predictable.

Chapter 4, Dynamic Approach on the Checking Frequency, describe the methodology on improvement, which includes the variables to control the change of checking frequency, and the changing strategy for checking frequency.

Chapter 5, Experiments on Dynamic Checking Frequency, describe the principles on dynamic monitoring and analyze the performance of the modification on Checking frequency.

Chapter 6, Conclusion, conclude the thesis and describe the further research in this particular field.

## Chapter 2 Data Preparation for Validation

Prepares the data which is used to valid the feasibility. The dataset is MNIST, select several different models to eliminate the influence from diversity of models and configure multiple versions of different number of epochs to Avoiding chance from experiments with a specific number of epochs.

### 2.1 Assumption

It is hypothesized that certain probability distributions exist for the occurrence of oscillating loss issues, the time at which the oscillating loss occurs is regular, and prior experiments have indicated that the occurrence and non-occurrence of the oscillation are random, but the distribution itself is not random. If this assumption is right, the dynamic checking frequency will save considerable time on useless periodical check of the AUTOTRAINER.

### 2.2 Data Preparation

The experiment utilized the classic MNIST dataset for machine learning, which contains a large set of images of handwritten digits for training and testing. To perform the experiment, a pre-existing model named “minist\_relu\_layer=20.h5” was used, which had been previously used to investigate the randomness of oscillating loss bugs in the AUTOTRAINER source code. The model was applied to the MNIST dataset to generate training data, which was then stored in a series of CSV files.

The model used in the most of case has 20 layers of rectified linear units (ReLU), “minist\_relu\_layer=20.h5”, which are commonly used in deep learning models for their simplicity and effectiveness. The other model named” minist\_relu\_layer=34.h5”, which is a 34-layer DNN model with the ReLU as the activation function. And the later one is used to eliminate the possibility of the difference caused by the difference of the model selection. The Adam [\[6\]](#) is used as the optimizer in the training. The experiment was run for a total of 100 training procedures, with each procedure generating a set of data that was stored in a separate CSV file. Each CSV file recorded the accuracy, loss value, validation accuracy, and validation loss value for that particular training procedure. The

details of those two model files description can be found in the Appendix I document, and those documents come from the AUTOTRAINER source code.

This approach provides a detailed record of the training data and allows for further analysis of the generated data to investigate the occurrence of oscillating loss bugs in the MNIST dataset. By analyzing the data in these CSV files, researchers can explore the characteristics of the oscillating loss bugs and potentially identify any patterns or underlying regularities in their occurrence. Overall, this experimental approach provides a valuable tool for studying the occurrence and behavior of oscillating loss bugs in machine learning models.

### *2.2.1 Environments*

The whole system and validation are deployed on the platform AutoDL, <https://www.autodl.com/>

Such a platform provides multiple choices on the GPU servers. During the data validation process, the AutoDL was used to run the python file and generate specific number of csv files which consist of the data about the accuracy, loss value, validation accuracy and validation value on each epoch. Based on those data collected, the jupyter notebook was used to run the ipynb file, which is a powerful tool to perform the data analysis.

For the lab and running environments, programming language, Python libraries, and tools are utilized throughout the project. Some key libraries include Keras [7], TensorFlow [8], Pandas, NumPy, Matplotlib, psutil, scikit-learn [9], and SciPy, with specific versions for each. And they are listed following.

```
tensorflow-gpu==2.3.0
```

```
keras==2.4.3
```

```
matplotlib==3.3.0
```

```
numpy==1.18.5
```

```
pandas==1.0.5
```

```
psutil==5.7.2
```

```
scikit-learn==0.23.1
```

```
scipy==1.4.1
```

To set up the environment on AutoDL, I uploaded the requirements.txt to Jupyter Lab and used Miniconda to construct a virtual environment. I created a new environment with Python (version 3.7) and used the command "pip install -r requirements.txt" to install the necessary libraries. After that, I executed "conda install cudatoolkit=10.1 cudnn=7.6.5" [10] in the terminal to equip the environment with CUDA, a parallel computing platform and application programming interface. This operation allows the training procedure to be performed with a GPU; otherwise, only the CPU is used. As a result, the CPU and GPU version is much faster than the CPU-only version. For example, for the prototype training model "minist\_relu\_layer=20.h5," the CPU-only version takes 8 seconds for a single epoch, while the CPU and GPU version only takes 5 seconds for an individual epoch.

## 2.3 CPU and GPU

Deep Neural Networks (DNNs) are used in various applications, such as image recognition, speech recognition, and natural language processing. DNNs are computationally intensive, which makes training them time-consuming. To speed up the training process, Graphics Processing Units (GPUs) are used in addition to the Central Processing Units (CPUs). CPUs are designed to handle a wide range of tasks and are optimized for single-threaded performance. They have a few powerful cores that can execute a small number of tasks at a time. In contrast, GPUs have many small, efficient cores that can execute multiple tasks in parallel, making them well-suited for tasks that require a lot of parallel processing, such as DNN training. When training a DNN, the data is fed to the network in batches, and each batch requires a set of operations to be performed on the input data. These operations can be performed in parallel on a GPU, which can significantly speed up the training process. GPUs are also designed to handle large matrices, which are commonly used in DNNs. CPUs are still necessary for DNN training as they handle the I/O operations, such as loading data into memory and saving the trained model to disk. They also handle tasks that are not suitable for parallel processing, such as control flow operations. When performing the training process to validate the assumption using different computational resources, differences were

observed between using only a CPU and using both a CPU and GPU. The initial task was performed using only a CPU, which took around 8 seconds per epoch. However, when both CPU and GPU were used, the time consumption per epoch reduced to around 5 seconds, resulting in an increase in training performance by approximately 67.5%. At the beginning, the GPU is not work since the dashboard offered by AutoDL shows that the percentage of use in GPU is 0%, thanks to the article from CSDN [https://blog.csdn.net/m0\\_51440939/article/details/125695072](https://blog.csdn.net/m0_51440939/article/details/125695072) [10], by creating the virtual environment in conda, and reset the cuda toolkit, after that, the percentage of use for GPU reaches 45% in average and the time consumption for each epoch in MNIST training get improved from 8s each epoch to 5s each epoch.



## Chapter 3 Data Analysis for Validation

Analyze the data and valid the strategy works. Based on the data generated, conclude a general theory, and prove that the distribution of Oscillating Loss occurrence is predictable.

### 3.1 Experiments

#### 3.1.1 *Data Types*

Once data has been collected in CSV files, it can be analyzed using Jupyter Notebook. The ipynb files are comprised of multiple cells, each of which can be executed independently, making data analysis a more efficient and flexible process.

To analyze the training results, each batch is considered separately, and depending on the number of training iterations in each batch, the results are analyzed one by one. The goal is to determine how many models within the batch are affected by oscillating loss, and at which stage this loss occurs. By identifying this information, it becomes possible to better understand the behavior of the models during training, and make informed decisions about how to adjust the training process in order to achieve better results.

#### 3.1.2 *Data Loading*

To load and read the CSV files, the “pandas” library is commonly used. Specifically, the built-in function “`read_csv()`” is used to import the data. After performing some basic operations on the imported tables, the “`ol_judge()`” function is called to determine whether the model is experiencing oscillating loss. This function returns the number of epochs at which the oscillating loss bug first appears.

By utilizing these functions and libraries, data scientists can more effectively analyze and troubleshoot issues in their models during training. The ability to pinpoint exactly when oscillating loss occurs can help identify potential problems and provide insights into how to improve the training process.

### 3.1.3 *How to determine the condition of oscillating loss*

The “`ol_judge( )`” function works by analyzing a list of accelerometer values and determining if there is a pattern of oscillating loss present. It does this by first identifying the maximum and minimum values in the data stream that meet certain criteria. Maximum values are identified where the value at index  $i$  is greater than both the value at index  $i-1$  and the value at index  $i+1$ . Similarly, minimum values are identified where the value at index  $i$  is less than both the value at index  $i-1$  and the value at index  $i+1$ . The function then calculates the difference between the maximum and minimum values, and counts the number of differences that are greater than or equal to a specified threshold parameter. Finally, the function compares the number of differences to a specified rate parameter, which is the proportion of maximum and minimum values that must have a difference greater than the threshold in order to identify oscillating loss. If the number of differences is greater than or equal to the rate times the length of the accelerometer data stream, the function returns a value of 1 to indicate that oscillating loss has been detected. Otherwise, the function returns a value of 0 to indicate that no oscillating loss has been detected.

### 3.1.4 *Data Cleaning*

There is no need to perform the data cleaning, since there is no NaN value or other weird values (accuracy beyond 1, accuracy lower than 0).

### 3.1.5 *Data Analysis*

After performing simple data operations on the CSV files, we have generated a chart that displays the data for each batch of training sets. The chart is useful for understanding the behavior of the models during the training process. Two key parameters, “`unstable_threshold`” and “`unstable_rate`,” are used to determine whether the oscillating loss bug occurs. These correspond to the threshold and rate input parameters in the “`ol_judge()`” function.

By default, the values for these two parameters are set to 0.02 and 0.20, respectively. Specifically, the unstable threshold parameter indicates that a model will only be considered to have oscillating loss if the difference between the maximum and minimum values in a given interval exceeds the threshold. The unstable rate parameter indicates

that the length of violation must be greater than 1/5 of the total length of the training process for the model to be considered as having oscillating loss.

These parameters provide a standardized set of criteria for determining when a model is experiencing oscillating loss, which can help to identify and troubleshoot issues during training. By carefully analyzing the data and adjusting these parameters as necessary, data analysts can improve the accuracy and effectiveness of their models.

The detail of each case is listed in the [Table 3-1](#), which contains the time when the bug detected and the percentage of the buggy cases etc. And the list of the time when the bug detected stored in the [Table 3-2](#) separately for each individual case.

Table 3-1 Detail in each case

Case Name	Epochs	Detected percentage	model
Case_50epoch1	50	60%	minist_relu_layer=20.h5
Case_50epoch_anoterModel	50	35%	minist_relu_layer=34.h5
Case_50epoch2_CPUversion	50	64%	minist_relu_layer=20.h5
Case_100epoch1	100	69	minist_relu_layer=20.h5
Case_100epoch2	100	67	minist_relu_layer=20.h5
Case_125epoch1	125	70	minist_relu_layer=20.h5
Case_125epoch2	125	48	minist_relu_layer=20.h5
Case_150epoch1	150	38	minist_relu_layer=20.h5
Case_150epoch2	150	34	minist_relu_layer=20.h5

Table 3-2 Training Results and number of trainings.

Case name	The time when the bug detected (counted as epochs)	Training numbers

Case_50epoch1	[14, 14, 12, 15, 16, 14, 18, 12, 10, 14, 13, 15, 14, 15, 15, 12, 13, 13, 12, 15, 16, 12, 14, 14, 15, 15, 14, 11, 18, 12, 17, 14, 11, 13, 12, 15, 15, 13, 13, 15, 16, 12, 14, 13, 14, 15, 15, 11, 15, 14, 16, 15, 15, 15, 16, 15, 14, 12, 12, 12]	100
Case_50epoch_anoterModel	[16, 12, 12, 15, 13, 12, 13, 12, 17, 14, 13, 12, 14, 12, 14, 13, 12, 11, 15, 17, 16, 13, 12, 14, 13, 15, 15, 17, 13, 10, 12, 13, 13, 12, 15]	100
Case_50epoch2_CPUversion	[12, 14, 17, 12, 14, 13, 18, 13, 13, 12, 15, 15, 14, 15, 14, 13, 16, 14, 14, 14, 15, 13, 14, 12, 14, 16, 16, 14, 16, 13, 15, 13, 14, 15, 14, 14, 13, 10, 14, 15, 13, 16, 14, 13, 15, 13, 14, 12, 15, 15, 14, 14, 13, 14, 12, 13, 14, 15, 14, 14, 16, 13, 16, 13]	100
Case_100epoch1	[32, 32, 25, 29, 28, 28, 31, 27, 28, 30, 28, 31, 32, 32, 27, 31, 32, 27, 30, 27, 28, 28, 27, 33, 31, 29, 30, 35, 27, 27, 27, 28, 31, 33, 31, 27, 31, 35, 28, 27, 25, 28, 33, 30, 31, 34, 30, 30, 31,	100

	29, 25, 29, 32, 26, 29, 26, 27, 28, 29, 36, 29, 32, 29, 30, 33, 26, 30, 28, 28]	
Case_100epoch2	[32, 24, 31, 31, 35, 28, 32, 29, 28, 28, 28, 27, 34, 27, 28, 33, 33, 30, 28, 28, 25, 25, 29, 31, 27, 27, 26, 27, 28, 30, 28, 29, 30, 27, 29, 29, 29, 26, 29, 30, 26, 25, 30, 25, 27, 30, 28, 32, 34, 28, 31, 30, 31, 30, 30, 27, 28, 29, 30, 27, 31, 31, 31, 31, 27, 30, 29]	100
Case_125epoch1	[52, 35, 35, 33, 35, 39, 35, 36, 37, 36, 33, 45, 48, 41, 48, 36, 36, 36, 39, 36, 33, 34, 41, 37, 38, 37, 38, 36, 36, 35, 35, 38, 36, 40, 42]	50
Case_125epoch2	[35, 37, 35, 32, 53, 41, 35, 36, 35, 39, 40, 34, 34, 37, 34, 37, 38, 39, 38, 41, 43, 40, 34, 47]	50
Case_150epoch1	[54, 46, 41, 43, 48, 44, 41, 50, 49, 41, 47, 47, 48, 43, 44, 44, 56, 53, 44]	50
Case_150epoch2	[47, 48, 51, 47, 52, 47, 44, 47, 55, 40, 47, 49, 42, 49, 44, 45, 39]	50

### *50-epoch version*

The distribution of buggy cases in CPU: [12, 14, 17, 12, 14, 13, 18, 13, 13, 12, 15, 15, 14, 15, 14, 13, 16, 14, 14, 14, 15, 13, 14, 12, 14, 16, 16, 14, 16, 13, 15, 13, 14, 15, 14, 14, 13, 10, 14, 15, 13, 16, 14, 13, 15, 13, 14, 12, 15, 15, 14, 14, 13, 14, 12, 13, 14, 15, 14, 14, 16, 13, 16, 13]

The distribution of buggy cases in case\_50epoch\_1: [14, 14, 12, 15, 16, 14, 18, 12, 10, 14, 13, 15, 14, 15, 15, 12, 13, 13, 12, 15, 16, 12, 14, 14, 15, 15, 14, 11, 18, 12, 17, 14, 11, 13, 12, 15, 15, 13, 13, 15, 16, 12, 14, 13, 14, 15, 15, 11, 15, 14, 16, 15, 15, 15, 16, 15, 14, 12, 12, 12]

The distribution of buggy cases in case\_50epoch\_anotherModel: [16, 12, 12, 15, 13, 12, 13, 12, 17, 14, 13, 12, 14, 12, 14, 13, 12, 11, 15, 17, 16, 13, 12, 14, 13, 15, 15, 17, 13, 10, 12, 13, 13, 12, 15]

The numbers in the 3 lists represent the moment when the bugs get detected, in this stage, 3 result sets are chosen with the same 50 epochs. Each result sets have 100 cases.

Due to the different model choice, this model with 34 layers illustrates a different distribution on the buggy models and no bug detected models. In this particular case, only 35% of all the cases(35 cases) are detected with Oscillating Loss bug, and there is no symptom that they have Oscillating Loss bug. However, the distribution of the stage (epoch number) when the bugs detected is similar to the CPU version and GPU version of the selected model. [Table 3-2](#) shows the difference between “anotherModel” and the other two cases on the distribution of buggy and bug-free model comes from the different model selection. The “anotherModel” uses the 34-layer DNN model (“minist\_relu\_layer=34.h5”) to perform the experiment, well the other two cases select another one, the 24-layer one(“minist\_relu\_layer=20.h5”). The 24-layer model has a more possibility to generate the buggy model, which means that it has a better performance on the generating samples to analyze. The details of these two different training model are placed in the Appendix I Documents.

[Table 3-3](#) shows that the model with 20 layers is able to generate almost twice buggy models as the 34-layer one, which means that the 20-layer model can save half of the time consumption on buggy samples collection.

Table 3-3 The distribution of buggy model and bug-free model with 50 epochs

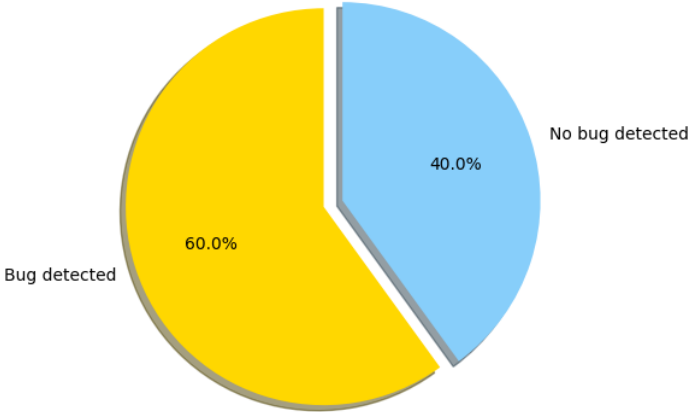
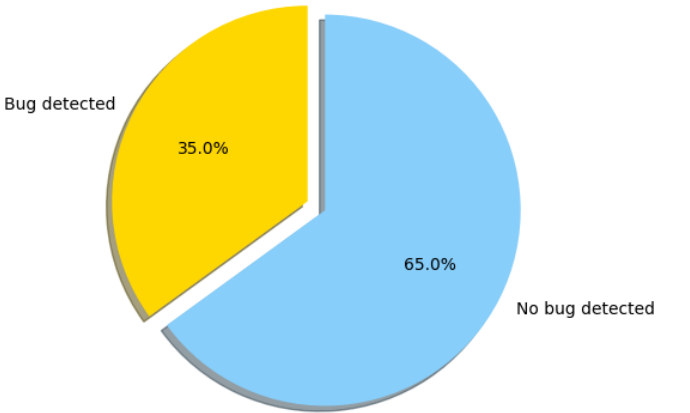
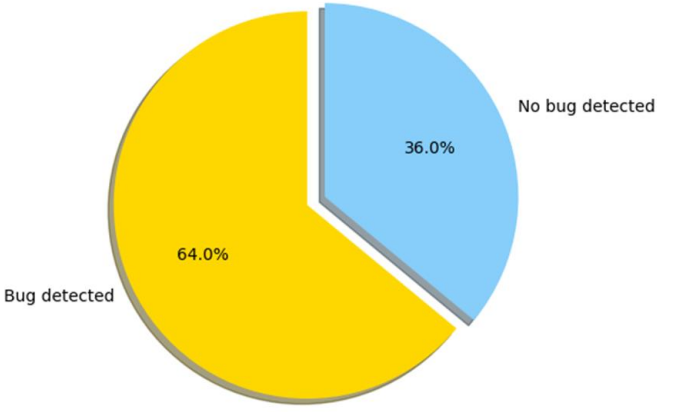
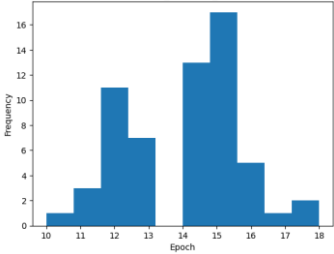
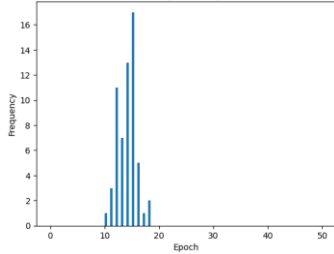
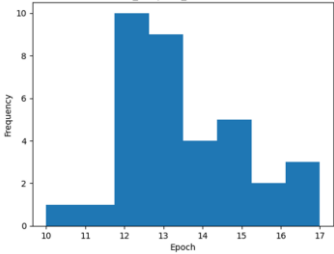
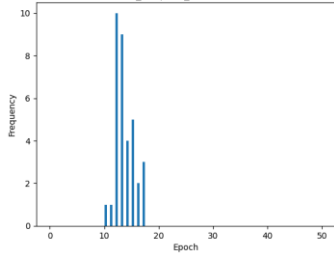
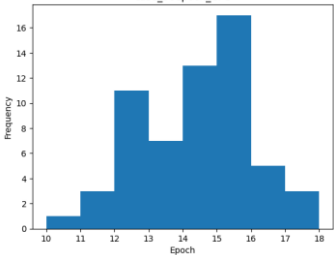
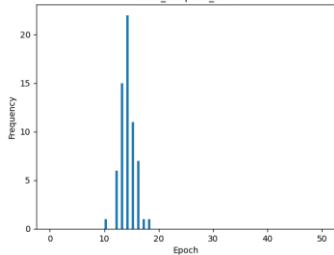
The comparison on the distribution of buggy model and bug-free model with 50 epochs							
Training with GPU, the model is the 20-layer model.	<p>case_50epoch_1</p>  <table><thead><tr><th>Category</th><th>Percentage</th></tr></thead><tbody><tr><td>Bug detected</td><td>60.0%</td></tr><tr><td>No bug detected</td><td>40.0%</td></tr></tbody></table>	Category	Percentage	Bug detected	60.0%	No bug detected	40.0%
Category	Percentage						
Bug detected	60.0%						
No bug detected	40.0%						
Training with GPU, the model is the 34-layer model.	<p>case_50epoch_anthoerModel</p>  <table><thead><tr><th>Category</th><th>Percentage</th></tr></thead><tbody><tr><td>Bug detected</td><td>35.0%</td></tr><tr><td>No bug detected</td><td>65.0%</td></tr></tbody></table>	Category	Percentage	Bug detected	35.0%	No bug detected	65.0%
Category	Percentage						
Bug detected	35.0%						
No bug detected	65.0%						
Training with CPU, the model is the 20-layer model.	<p>case_50epoch_2_CPUversion</p>  <table><thead><tr><th>Category</th><th>Percentage</th></tr></thead><tbody><tr><td>Bug detected</td><td>64.0%</td></tr><tr><td>No bug detected</td><td>36.0%</td></tr></tbody></table>	Category	Percentage	Bug detected	64.0%	No bug detected	36.0%
Category	Percentage						
Bug detected	64.0%						
No bug detected	36.0%						

Table 3-4 The distribution of the moment that the bug detected

	In the high possibility interval	In the whole training procedure
Training with GPU, the model is the 20-layer model.		
Training with GPU, the model is the 34-layer model.		
Training with CPU, the model is the 20-layer model.		

*100-epoch version*

The distribution of buggy cases in case\_100epoch\_1: [32, 32, 25, 29, 28, 28, 31, 27, 28, 30, 28, 31, 32, 32, 27, 31, 32, 27, 30, 27, 28, 28, 27, 33, 31, 29, 30, 35, 27, 27, 27, 28, 31, 33, 31, 27, 31, 35, 28, 27, 25, 28, 33, 30, 31, 34, 30, 30, 31, 29, 25, 29, 32, 26, 29, 26, 27, 28, 29, 36, 29, 32, 29, 30, 33, 26, 30, 28, 28]

The distribution of buggy cases in case\_100epoch\_2: [32, 24, 31, 31, 35, 28, 32, 29, 28, 28, 28, 27, 34, 27, 28, 33, 33, 30, 28, 28, 25, 25, 29, 31, 27, 27, 26, 27, 28, 30, 28, 29, 30, 27, 29, 29, 29, 26, 29, 30, 26, 25, 30, 25, 27, 30, 28, 32, 34, 28, 31, 30, 31, 30, 30, 27, 28, 29, 30, 27, 31, 31, 31, 31, 27, 30, 29]

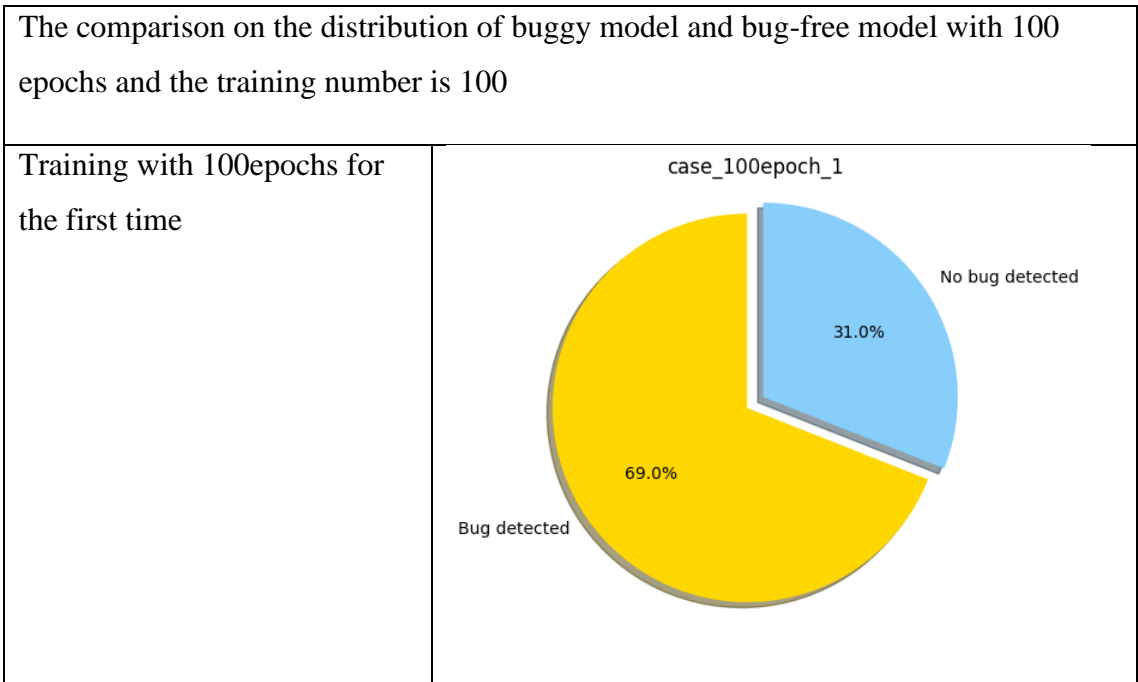


In the case of “case\_100epoch\_1” the given data represents the epoch numbers where bugs occurred during the training process of 100 models. Based on the provided data, we can see that the epoch numbers where bugs occurred range from 27 to 36, with the most common values being 27, 28, 29, 30, 31, and 32, occurring with the highest frequencies. This suggests that bugs mainly occurred in the early and middle stages of training in this dataset.

In the case of “case\_100epoch\_2”, the range of epochs in which OL bug occurs in these models is between 24-35, with 28 and 31 being the most common. The histogram shows that the epochs in which OL bug occurs in these models are mostly concentrated between 27-31 and the distribution is relatively even. In addition, the box plot shows that the median of epochs in which OL bug occurs in these models is 29, the lower quartile is 28, and the upper quartile is 31, with a few outliers. This data set shows the range and distribution of epochs in which OL bug occurs in 100 models and can be used to optimize and improve the training process of the models.

In conclusion, based on the [Table 3-6](#) there is the same distribution compares to the 50-epoch version in [Table 3-4](#). The high possibility interval in 100 epochs still ranges from 1/5 of the whole training procedure to 2/5 of the whole training procedure.

Table 3-5 The distribution of buggy model and bug-free model with 100 epochs



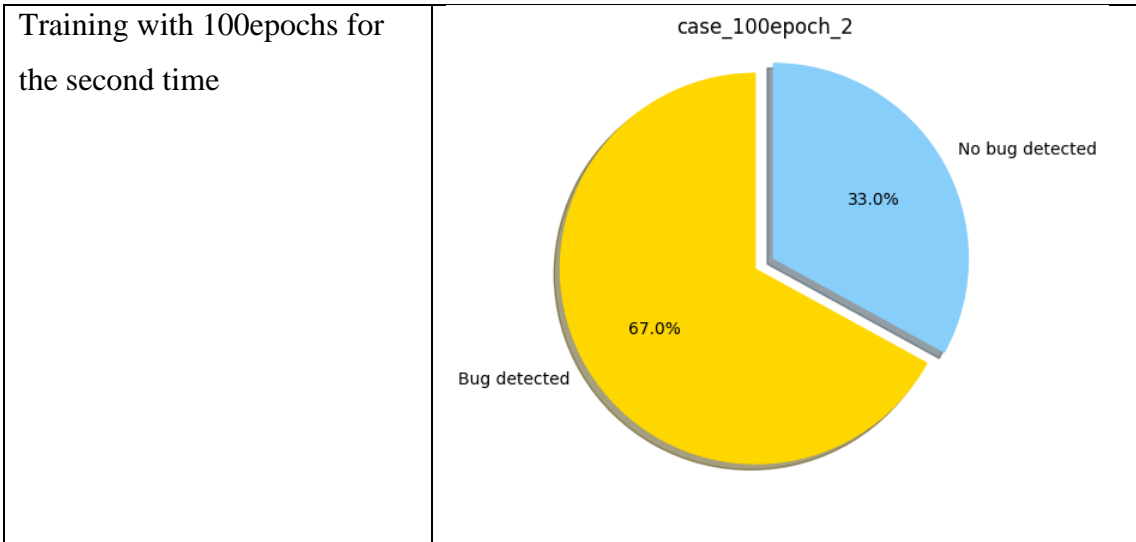
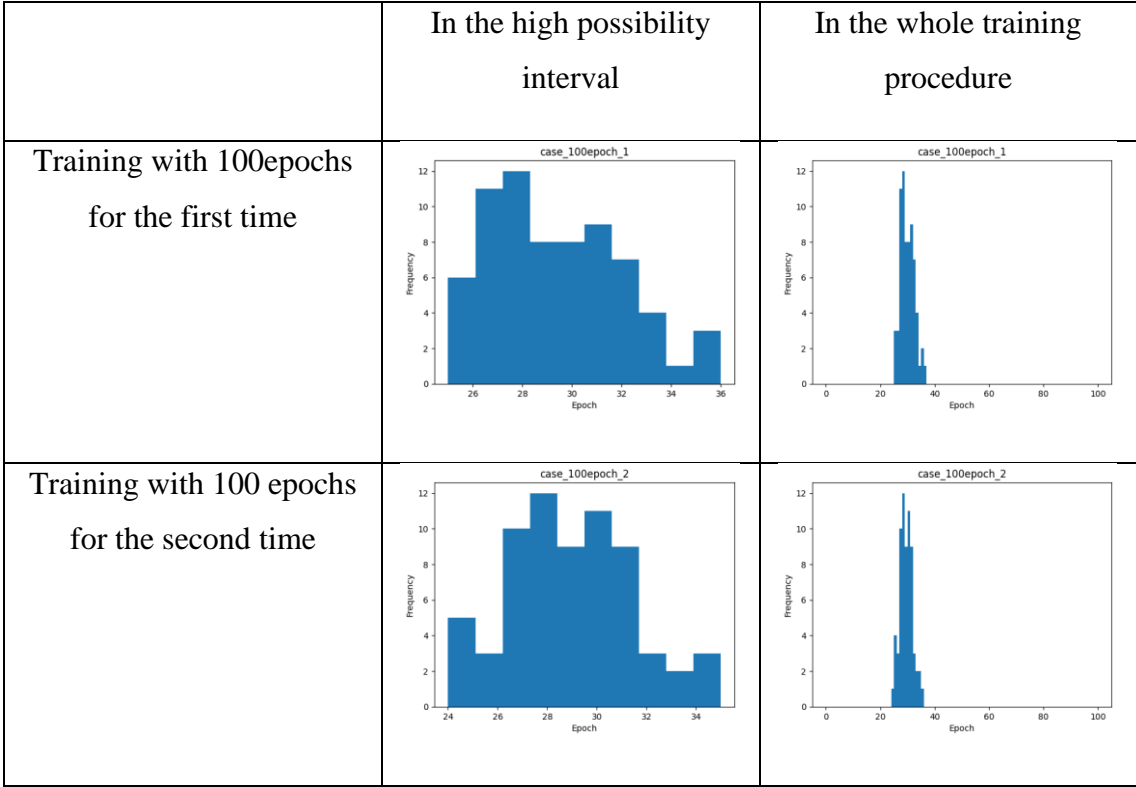


Table 3-6 The distribution of the moment that the bug detected



*125-epoch version*

The distribution of buggy cases in case\_125epcoh\_1: [52, 35, 35, 33, 35, 39, 35, 36, 37, 36, 33, 45, 48, 41, 48, 36, 36, 36, 39, 36, 33, 34, 41, 37, 38, 37, 38, 36, 36, 35, 35, 38, 36, 40, 42]

The distribution of buggy cases in case\_125epcoh\_2: [35, 37, 35, 32, 53, 41, 35, 36, 35, 39, 40, 34, 34, 37, 34, 37, 38, 39, 38, 41, 43, 40, 34, 47]

Based on the results from the training procedures above, there are clear distributions in those different cases above. In the cases with 50 epochs, the bugs are always detected from 10th epoch to 18th epoch, only the case with another model has no bug detected later than 17th epoch. However, maybe because the number of training procedures is not large enough, so the interval is not precise enough. To improve the fault tolerance rate, in the cases with 50 epochs, the high possibility interval is defined from tenth epoch to twentieth epoch.

For the two cases with 100 epochs, there are clear distributions illustrates that all the bugs are detected in the interval whose epochs from 20 to 40. And such a result show that the high possibility interval has the same location in all epochs for 100 epochs and 50 epochs version where only take the relative location into consideration, which, means that the OL problem can be detected in the interval from 1/5 of the whole training procedure to 2/5 of the whole training procedure. There is an assumption that maybe the model's unique attributes lead to this phenomenon. However, when taking another model into consideration, the distribution is the same as the original model. In the following experiments, the "another +model" (with 34 layers network) will be not taken into consideration, since in that particular model, the buggy models can be produced is relatively much less than the original one (with 20 layers network). And such a fact leads to a double time consumption if the 34 layers network is chosen to generate buggy modes to perform the experiments for statistics. Since in the cases above, in particular 50 epochs version, the original model generates more than 60 buggy cases for analyzing, while the 34-layer version only generates 35 buggy cases, relatively little number of samples has a worse performance than the original one.

When taking the 125 epochs version into consideration, the proportion of the bug detected cases is relatively little in the second training set. It should be a special situation since the training number for one set is modified from 100 to 50, the samples are less than the 100 trainings version and such a small number makes the occurrence of buggy case more contingent. But the histograms of two cases with 125 epochs has the same shape, although the volume of y-axis is different, which means that although the number of buggy models is not the same, the distribution and proportion of the buggy epochs are the

same. Meanwhile, those two cases have the numbers range from 30 to 55, actually, most of the numbers ranges from 30 to 50, only two single numbers are 52 and 53 separately in case 1 and case 2, the ratio of strange cases in case 1 and case 2 is 2.857% and 4.167%. Such a fact illustrates that the general distribution is similar to the former version of epochs. [Table 3-8](#) shows that the highest possibility interval is from 6/25 to 2/5, which is almost the same as the former two version of cases.

Table 3-7 The distribution of buggy model and bug-free model with 125 epochs

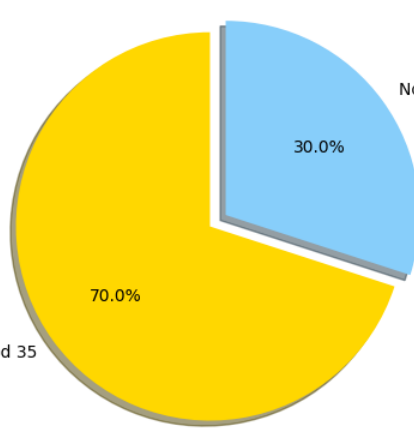
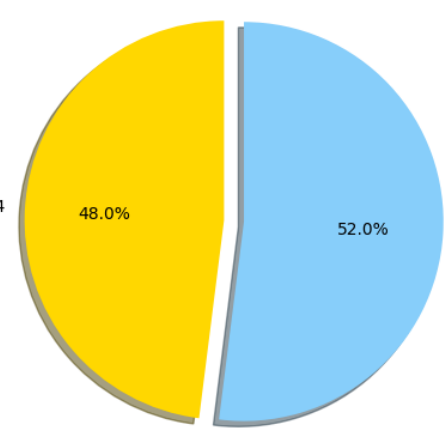
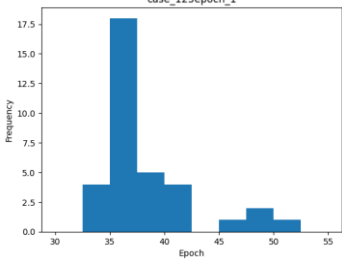
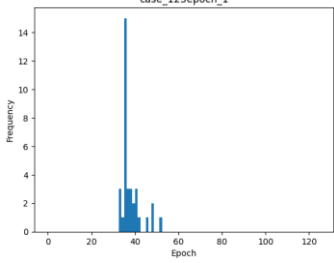
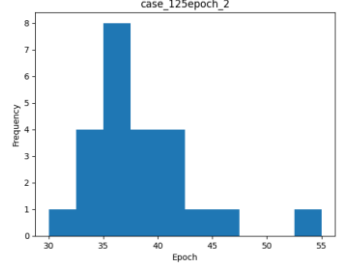
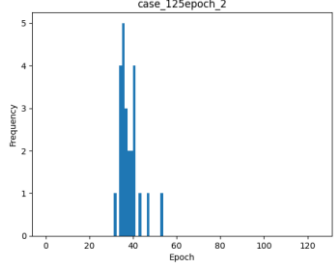
The comparison on the distribution of buggy model and bug-free model with 125 epochs and the training number is 50										
Training with 125 epochs for the first time	<div>case_125epoch_1</div>  <table><tr><th>Category</th><th>Count</th><th>Percentage</th></tr><tr><td>Bug detected</td><td>35</td><td>70.0%</td></tr><tr><td>No bug detected</td><td>15</td><td>30.0%</td></tr></table>	Category	Count	Percentage	Bug detected	35	70.0%	No bug detected	15	30.0%
Category	Count	Percentage								
Bug detected	35	70.0%								
No bug detected	15	30.0%								
Training with 125 epochs for the second time	<div>case_125epoch_2</div>  <table><tr><th>Category</th><th>Count</th><th>Percentage</th></tr><tr><td>Bug detected</td><td>24</td><td>48.0%</td></tr><tr><td>No bug detected</td><td>26</td><td>52.0%</td></tr></table>	Category	Count	Percentage	Bug detected	24	48.0%	No bug detected	26	52.0%
Category	Count	Percentage								
Bug detected	24	48.0%								
No bug detected	26	52.0%								

Table 3-8 The distribution of the moment that the bug detected

	In the high possibility interval	In the whole training procedure
Training with 125 epochs for the first time		
Training with 125 epochs for the second time		

### 150-epoch version

The distribution of buggy cases in case\_150epoch\_1: [54, 46, 41, 43, 48, 44, 41, 50, 49, 41, 47, 47, 48, 43, 44, 44, 56, 53, 44]

The distribution of buggy cases in case\_150epoch\_2: [47, 48, 51, 47, 52, 47, 44, 47, 55, 40, 47, 49, 42, 49, 44, 45, 39]

In the 150-epoch version, fewer models with the oscillating loss bug were detected compared to all the previous versions. This could be due to the longer length of each model. According to the methodology for judging the Oscillating Loss, one important factor is that the length of oscillating loss should be longer than a quarter of the total length of the training procedure in a model. Therefore, the unstable length is not long enough in these models. However, there is still a clear distribution in the 150-epoch version cases, as both of them are located between the 40th and 60th epoch, corresponding to the interval of 4/15 to 6/15 of the entire 150 epochs in each training procedure, which is shown in the [Table 3-10](#).

And due to the similar distribution for the occurrence of buggy models and bug-free models in [Table 3-3](#), [Table 3-5](#), and [Table 3-7](#), it can be conclude that the distribution on the buggy and bug-free is also predictable for specific model, however, the things turn out to be different in the second case of [Table 3-7](#) and [Table 3-9](#), since the bar charts shows that the distribution is different, so may be the more training epochs, the less Oscillating Loss problems, or the Oscillating always exists, but the oscillating lengthen would not be longer with the more training epochs. Might be more epochs should be a solution to the Oscillating Loss, but it will waste much more resources.

Table 3-9 The distribution of buggy model and bug-free model with 100 epochs

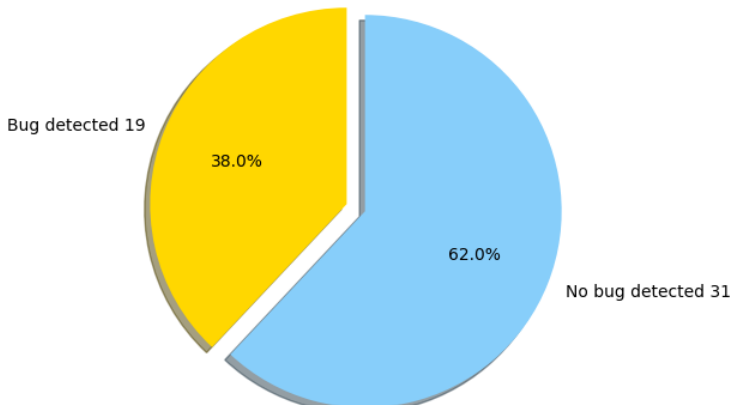
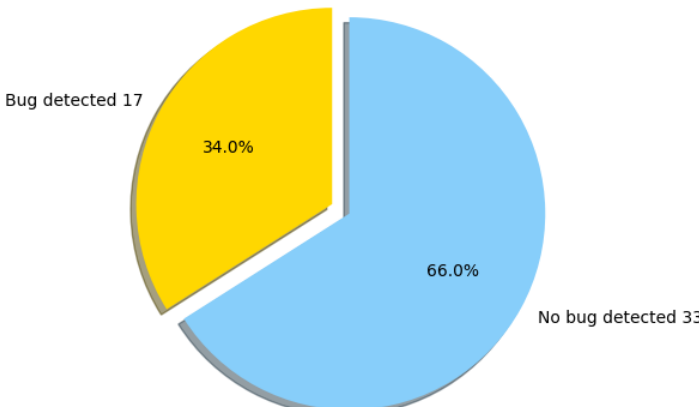
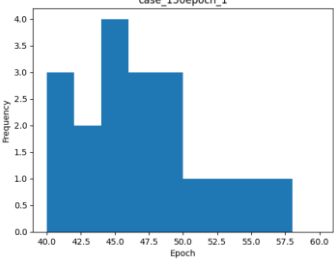
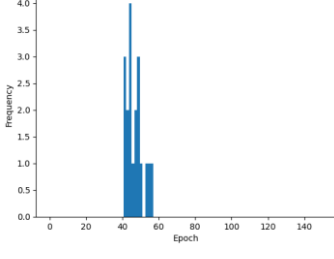
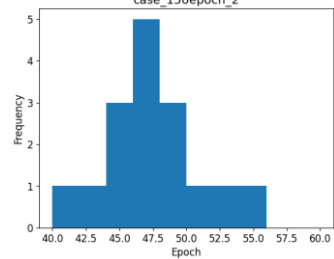
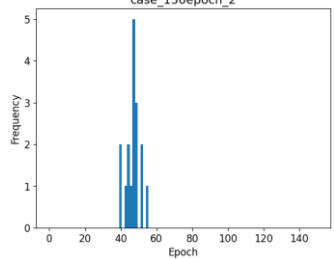
The comparison on the distribution of buggy model and bug-free model with 150 epochs and the training number is 50										
Training with 150 epochs for the first time	<div>case_150epoch_1</div>  <table><tr><th>Category</th><th>Count</th><th>Percentage</th></tr><tr><td>Bug detected</td><td>19</td><td>38.0%</td></tr><tr><td>No bug detected</td><td>31</td><td>62.0%</td></tr></table>	Category	Count	Percentage	Bug detected	19	38.0%	No bug detected	31	62.0%
Category	Count	Percentage								
Bug detected	19	38.0%								
No bug detected	31	62.0%								
Training with 150 epochs for the second time	<div>case_150epoch_2</div>  <table><tr><th>Category</th><th>Count</th><th>Percentage</th></tr><tr><td>Bug detected</td><td>17</td><td>34.0%</td></tr><tr><td>No bug detected</td><td>33</td><td>66.0%</td></tr></table>	Category	Count	Percentage	Bug detected	17	34.0%	No bug detected	33	66.0%
Category	Count	Percentage								
Bug detected	17	34.0%								
No bug detected	33	66.0%								

Table 3-10 The distribution of the moment that the bug detected

	In the high possibility interval	In the whole training procedure
Training with 150 epochs for the first time		
Training with 150 epochs for the second time		

Based on the results from the first case set in the version of 150 epochs, the buggy case number is:

[1, 3, 8, 10, 13, 18, 21, 22, 23, 24, 25, 32, 33, 35, 36, 39, 41, 42, 48]

And the numbers of epochs have been processed when the bug detected is:

[54, 46, 41, 43, 48, 44, 41, 50, 49, 41, 47, 47, 48, 43, 44, 44, 56, 53, 44]

Choose the eighth case in the example, its bug is generated in the forty-first epoch. The line graph for it is shown in the following. [Figure 3-1](#) shows the trend of 4 kinds of values in the csv file which stores the records for Oscillating Loss cases. The accuracy has the almost same shape with validation accuracy, so there is no overfitting. However, the trend of the accuracy is not significant in this graph, since the values ranges from 0.00 to 2.00, it is a pretty huge scale for accuracy. So, the next figure [Figure 3-2](#) only takes the accuracy into consideration, which shows a direct demonstration on the Oscillating Loss issue. There are dramatic fluctuations in the 0-20 epochs and 80-150 epochs stages.

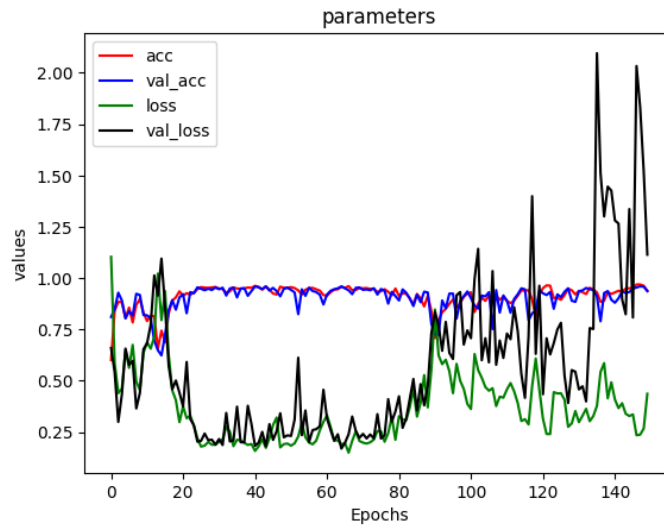


Figure 3-1 line graph on values for 150-epoch case

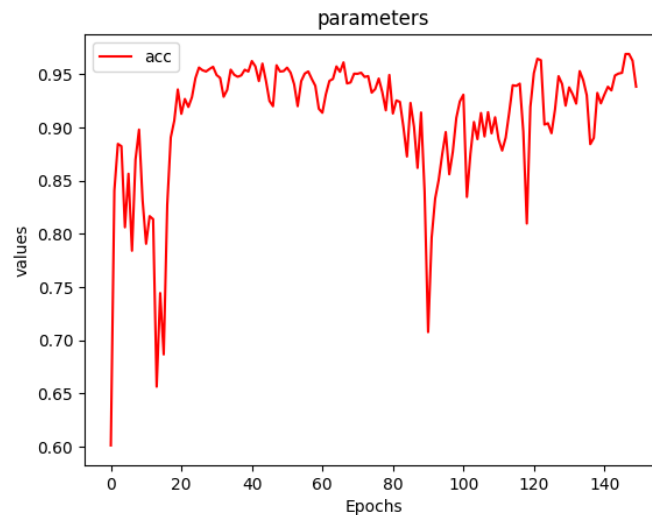


Figure 3-2 line graph on accuracy for 150-epoch case

Based on the experiments above, the high possibility zone could be the 1/5 to 2/5 in each case, and since there are still several strange values. The solution is that, reassign the rest of the whole procedure as the low possibility zone, with the checking frequency as 1 check per 10 epochs. Because in some cases such as the “39” in 150-epoch case 2, which is not in the high-possibility interval, the additional check in the zone out of high-possibility interval is necessary.



## Chapter 4 Dynamic Approach on the Checking Frequency

Describe the methodology on improvement, which includes the variables to control the change of checking frequency, and the changing strategy for checking frequency.

### 4.1 Assumption

More frequent problem checking causes higher runtime overhead. Suppose that one training iteration and one checking separately take  $t_1$  and  $t_2$  time, then the overhead of AUTOTRAINER is roughly equals to the product of  $q$  and  $t_2/t_1$ . (4,1)

$$overhead = q * \frac{t_2}{t_1} \quad (4,1)$$

where  $q$  is the checking frequency. [Figure 4-1] presents the correlations between checking frequency and runtime overhead on Circle and MNIST. The X-axis is the number of iterations between two checks ( $1/q$ ), and Y-axis is the runtime overhead. The solid line represents the collected data, and the dashed curve is the theoretical results (4,1). As we can see, shapes of experiment data conform to our theoretical analysis. By comparing the two figures, we observe the smaller dataset has the higher runtime overhead. By default, AUTOTRAINER checks the problem every 3 iterations, which causes less than 5 % overhead even for small datasets like Circle [2].

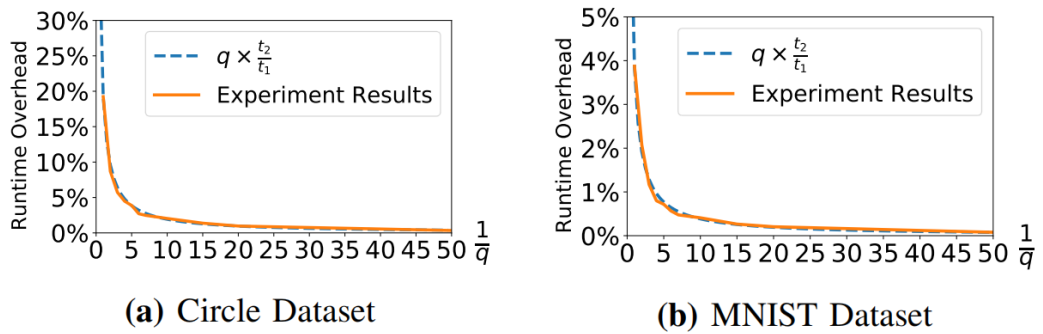


Figure 4-1 relationships between  $1/q$  and runtime overhead [1]

Problem	Detection Delay					
	$1/q=2$	$1/q=3$	$1/q=4$	$1/q=5$	$1/q=9$	$1/q=15$
VG	0.33	1.12	1.48	1.78	3.22	6.22
EG	0.38	1.38	2.38	3.38	7.38	13.38
DY	0.43	1.15	2.15	2.29	8.01	8.01
OL	0.40	1.60	1.60	2.40	3.40	6.40
SC	0.32	1.06	1.25	2.13	2.74	6.09

Figure 4-2 Oscillating Loss's detection delay [1]

According to the data from AUTOTRAINER, the reciprocal of checking frequency (parallel axis) affects a lot on the Runtime Overhead (vertical axis). And the two graphs (based on the circle and MINIST dataset) illustrate that Runtime Overhead decreases as  $1/q$  increases, indicating that Runtime Overhead is proportional to Checking Frequency. Particularly, the [Figure 4-1](#) shows that there is an inverse relationship between the runtime overhead and the value of  $1/q$ , where  $n$  is a constant. As the value of  $1/q$  increases (i.e., as  $q$  decreases), the runtime overhead decreases and vice versa. The rate at which the runtime overhead changes with respect to  $1/q$  is inversely proportional to the value of  $n$ , meaning that as  $n$  increases, the rate of change of the runtime overhead with respect to  $1/q$  decreases. Therefore, changes in the value of  $1/q$  have a significant impact on the runtime overhead when  $1/q$  is small (i.e., when  $q$  is large), but as  $1/q$  grows larger (i.e., as  $q$  becomes smaller), the impact becomes progressively smaller. To be more efficient, I reassign the value of the checking frequency, in the original version, the checking frequency is a constant value, however, such a method will waste the time on unnecessary checking on the low possibility interval, and makes the bug detected late in the high possibility interval. My assumption is that the checking frequency should be transformed from constant value to variable and based on the current epochs have been processed the checking frequency shifts.

[Figure 4-3](#) shows that Oscillating Loss problem occurs in the first 10 training epoch in 50% of the cases. In other cases, this problem happens in later epochs and 29% cases even don't perform this problem in the training.

The assumption on the particular distribution is validated, however, the original statistics on the Oscillating Loss about the occurrence timing is wrong, since the condition on Oscillating Loss is the rate greater than 0.2 which means the oscillating lengthen longer

than 1/5 of the whole single training procedure. Particularly, in a 50-epoch version of case, there should be no possibility that the bug detected in the first ten epochs. But the motivation in AUTOTRAINER introduces the distribution on Oscillating Loss with half of cases are detected with bugs in the first 10 epochs. Since the shortest lengthen should be passed in 50-epoch version is 10 epochs, and such a lengthen causes no bug will be detected in the first 10 epochs. That might be a mistake for the AUTOTRAINER.

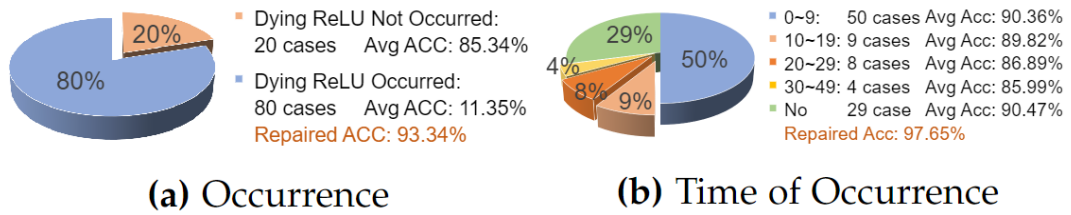


Figure 4-3 Oscillating Loss's Occurrence and time for Dying ReLU [1]

## 4.2 Key Value

The key value represents the checking frequency is the variable named "check\_type", the "check\_type" is default assigned as "epoch\_5", and that stands for that the checking frequency is 1 check per 5 epochs, which is a constant value. However, the dynamic checking frequency requires it a variable. The current number of epochs should be calculated and get the proper checking frequency and assign it to the "check\_type". It is hard to find out the current number of epochs have been processed. So, the author made a detour, the current number of epochs is impossible to get, but the history of the other values is not. To be more specific, assign the "len(history["loss"])" to the "currentEpochs". Since the number of the loss values is the epochs that the training procedure has done. And the history of loss values is stored in the system. Based on this methodology on redefining the checking frequency, it is transformed to be a dynamic one, which can change by the pace of the training procedure.

## Chapter 5 Experiments on Dynamic Checking Frequency

Describe the principles on dynamic monitoring and analyze the performance of the modification on Checking frequency.

### 5.1 Experiments

The Class LossHistory is built to store the accuracy and loss value by the API from Keras callbacks callback [\[7\]](#).

Keras callbacks are a set of methods that enable developers to check their model's intermediate features (e.g., weights, gradients). Also, callbacks enable the developers to inspect the model's behavior during training. Our callback approach is inspired by prior work. In particular, our callbacks allow capturing and recording the key values (i.e., weight, gradient, etc.) during feed-forward and backward propagation stages [\[11\]](#).

A callback function is a set of functions that are called during a specific phase of training. You can use the callback function to observe the state and statistics inside the network during training. By passing a list of callback functions to the model's ".fit()", you can call the functions in that set at a given training stage. The callback function takes a parameter to the dictionary logs, which contains a series of information related to the current batch or epoch [\[11\]](#).

The values are watched in the class file, "monitor". And the useful data generated in the training procedure like accuracy, loss value, accuracy for validation and accuracy for loss value. Meanwhile, there are also gradient issue scheduler and gradient calculator in the module to compute the gradient and other values for analysis on some particular symptoms such as the issues on convergence, exploding gradient, vanishing gradient.

For the Oscillating Loss issue, the gradient is not needed in both analysis and fixing, so there is no need to bring the gradient into consideration, which means that the gradient related functions would not be called in Oscillating Loss detection and repair. And the author's methodology has been validated in the former chapter, the particular distribution does exist in the issue of Oscillating Loss.

By modifying the “checkstyle” to different values, like modifying the number of the epochs to have an effect on the checking frequency and cut down the time cost. In the original AUTOTRAINER, the check frequency, represented in the code as “checktype” is default to 5 or 3, which is depended on the functions default settings. The goal of my personal improvement is to convert the static “checktype” to a dynamic one. Which means that when the system monitoring the model, the check frequency is changing by the progress of the training procedure. For some kinds of parts, the problem density is higher than other, so the checking frequency can be high in that particular period to detect the problem as soon as possible, the higher density of problems, the higher checking frequency. Because of the check need time, the dynamic checking procedure will save a lot of time. The original idea of AUTOTRAINER is to save the time for retraining, because we can only retrain the model after the original training procedure completed without AUTOTRAINER. With the improvement of the checking frequency, the system will save more time.

```
- AutoTrainer/
  - data/
  - demo_case/
    - Gradient_Vanish_Case/
    - Oscillating_Loss_Case/
    - Improper_Activation_case/
  - utils/
  - reproduce.py
  - README.md
- Motivation/
  - DyingReLU/
  - OscillatingLoss/
  - README.md
```

Figure 5-1 Code Structure in Original AUTOTRAINER [2]

The [Figure 1-1](#) shows the overall architecture of the AUTOTRAINER system.

The training monitor starts a training procedure and records data which is used to recognize symptoms and retrain the model when a problem is detected. It records the Model definition including layers and their configurations (kernel sizes in convolutional layers), the definition of optimizer and its parameters, training and validation accuracy and loss values, gradients for each neuron, Hyper-parameters such as the batch size and learning rate.

The problem recognizer regularly conducts analysis on the recorded data to recognize training problems. The first column lists the training problems, and the second column specifies the symptoms involving gradient and training accuracy. If the depicted condition is met, our system regards the corresponding symptom as observed. The last column presents the built-in solutions in AUTOTRAINER.

The main role of the solution scheduler is to pick one solution to fix the problem and restart the training procedure. For the same problem, it will try each possible solution one by one based on the default order if users do not specify preferred orders. If one solution can fix the problem, the scheduler will not be triggered by the same problem. otherwise, it will try a new solution. And if none of these solutions can fix it, AUTOTRAINER fails to resolve this problem and will report this to the user to determine what to do next [2].

As for the code structure shown in the [Figure 5-1](#), the components which need to be adjusted is not only the “monitor”, since the architecture of the system and the code is different. The original AUTOTRAINER has both high coupling and high cohesion, which makes the modification difficult. The modifications are performed in the utils, there are several components in it. And the “moudles.py”, the “monitor.py” and “utils.py” are taken into consideration. The author creates a new function in the “monitor.py” to modify the checking frequency according to the current number of epochs, and the current number of epochs come from the length of the history values. In addition, it should be able to call the history data in the “module.py”. Meanwhile, the main function named “model\_train( )”called in the demo case is defined in the “utils.py”, and some modification are made in that to access the history to compute the proper checking frequency based on the current number of epochs.

The check frequency should be calculated by the current epoch number., however, the current epoch number is not available even the epoch numbers can be seen in the running terminal. But the Keras [7] provides the “History” in the class “Callback”, which contains 4 types of values in each epoch in the whole training procedure, the accuracy, loss value, validation accuracy, validation loss value, and they are represented by “acc”, “loss”, “val\_acc”, “val\_loss”. The results for the experiments in Chapter III also depends on the “History” to generate the records for analysis and issue judgment. Although the current number is not available, even the user can see it. The number of the

records in “History” is equal to the epochs passed. So, the current epoch number can be replaced with the length of “History[“acc”]”. And then, the checking frequency can be calculated in “getdynamicCF()”. The procedure of “getdynamicCF()” is shown in [Figure 5-2](#).

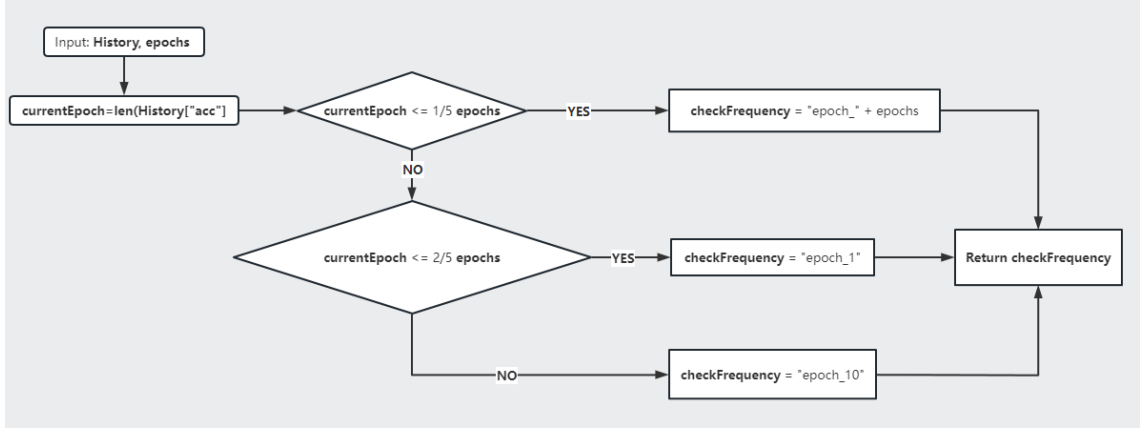


Figure 5-2 The procedure of the “getdynamicCF()”

In the [Figure 5-2](#), the high possibility interval is selected as the 1/5 to 2/5 of the whole training procedure, when the epoch passed less than 1/5 of the whole length, the check frequency will be set extremely slowly, to avoid useless check in the first stage(first 1/5 of all epochs). And when the training gets into the second stage (1/5 to 2/5 of all epochs). Although most of buggy cases happened in the second stage, as known as the high possibility interval, to avoid the strange cases shown in the 125-epoch experiments, the rest of the training is set as 1 check per 10 epochs. In the AUTOTRAINER system, the variable for check frequency is the “checktype” and it is default set as “epoch\_5” which means 1 check per 5 epochs. That is the reason why the return value in this function is a string but not a integer.

## 5.2 Overhead and Validation

It is hard to evaluate the difference between the original version and the improved version, because the checking has a lower time occupation on the whole training procedure, and the occurrence of oscillating loss issue is random. In AUTOTRAINER original time evaluation, for normal training, the runtime overhead is purely from problem checker, which is about 1% [1]. And the possibility in occurrence of bugs is random, if the original version has 1 more formal case than the improved one, the time difference is

too huge to cover the time saved by dynamic check for hundreds of cases. More terribly, the timing of bug occurrence is random, too. Which means that we cannot control the consistency on epochs when the bug is detected. It is hard to perform a large-scale experiment to evaluate the improved system.

Generally, in original AUTOTRAINER for the example case with 50 epochs and 700 samples, there is 546  $\mu\text{s}$  in average for 1 epoch without check and 607  $\mu\text{s}$  in average for 1 epoch with check. The basic parameters for the equipment are listed in the [Table 5-1](#). The difference is around 61  $\mu\text{s}$  per 5 epochs. For the first 10 epoch, there are 2 checks in original AUTOTRAINER, and the number is 0 in the improved version, which save around 122  $\mu\text{s}$  in the first 10 epochs. Meanwhile, for the second 10 epochs, the improved one may detect the bug 5 epochs at most in advance compared to the original one, which saves the time for 5 epochs. In some cases, with relatively little epochs, dynamic checking frequency takes a huge occupation. For example, the case with 50 epochs is able to save 10% in time consumption at most.

So, in the specific case with 50 epochs and 700 samples, the results showed in [Figure 5-3](#) indicates the improvement on time consumption should be around 2800  $\mu\text{s}$  at most, which is 1/10 of the whole normal training procedure. It is clear that the improvement on AUTOTRAINER have a considerable time saving. The maximum can be got when the bug comes out at eleventh epoch with the whole lengthen is 50 epochs, in this particular case, the original AUTOTRIANER cannot stop the training until fifteenth epoch.

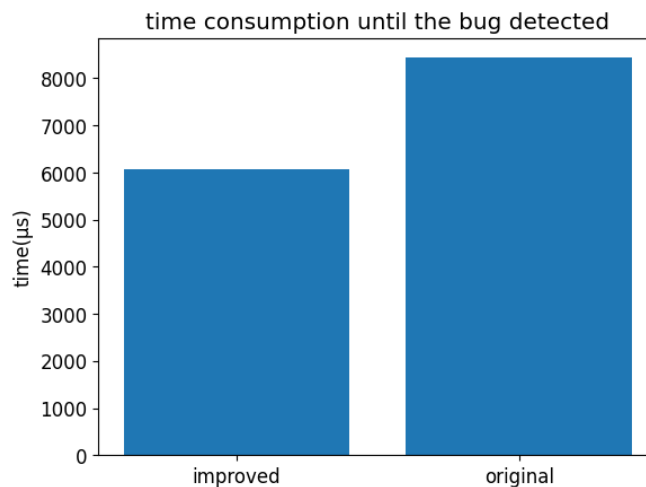


Figure 5-3 time consumption before the detection of OL



Table 5-1 Parameters for the equipment

CPU	Intel i7-8750H
CPU occupation	37%
GPU	No GPU used in the experiments

## Chapter 6 Conclusion

### 6.1 Conclusion

The DNN provides huge improvement on both research and industry, but the Oscillating Loss issue struggles the users a lot. The AUTOTRAINER has offered an available solution on that to save the time on waiting for the complete of the buggy training procedure. The improvement on the AUTOTRAINER described in this paper, provides a dynamic methodology to check the parameters to judge the Oscillating Loss issue. Corresponding to the original version, the improved one has a around 25% time saving on Oscillating Loss detection. This represents a significant improvement in the efficiency and effectiveness of deep learning models trained using the AUTOTRAINER. The datasets, training results, analysis and codes can be found at [https://github.com/MiSFiT5/FYP\\_dynamicAUTOTRAINER/tree/master](https://github.com/MiSFiT5/FYP_dynamicAUTOTRAINER/tree/master).

### 6.2 Future work

In the future, I will continue this work in my Master days. To be more specific, the properties of the other 4 types of the bugs in DNN (Dying ReLU, Oscillating Loss, Exploding Gradient, Vanishing Gradient, and Slow Convergence) are going to be explored. Mainly focus on the regularities in bug generation. Besides, I will contribute myself on the prevention of the bugs occurred in DNN, try my best to make the training more efficient, although the improved AUTOTRAINER introduced in this paper is powerful, the time is wasted when the system is detecting the bug. If the bugs can be predicted, there is no need to retrain the models, which will save a lot of time.

## References

- [1] LeCun, Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature* (London), 521(7553), 436–444. <https://doi.org/10.1038/nature14539>
- [2] Zhang, Zhai, J., Ma, S., & Shen, C. (2021). AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 359–371. <https://doi.org/10.1109/ICSE43902.2021.00043>
- [3] Wardat, Cruz, B. D., Le, W., & Rajan, H. (2022). DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 561-572. <https://doi.org/10.1145/3510003.3510071>
- [4] [LeCun et al., 1998a] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.
- [5] Y. LeCun, "The mnist database of handwritten digits," <http://yann.lecun.com/exdb/mnist/>, 1998.
- [6] D. P. Kingma and J. Ba, "Adam : A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
- [7] "Keras: The python deep learning library," 2020, <https://keras.io3>
- [8] M. Abadi, P Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in 12th {USENIX} symposium on operating systems design and implementation ({OSD1} 16), 2016, pp. 265-283.
- [9] "scikit-learn, machine learning in python," 2020, <https://scikit-learn.org/stable/>
- [10] [https://blog.csdn.net/m0\\_51440939/article/details/125695072](https://blog.csdn.net/m0_51440939/article/details/125695072)
- [11] <https://keras-cn.readthedocs.io/en/latest/other/callbacks/>

## Appendix I Documents

minist\_relu\_layer=20.h5: This is the name of a .h5 file which is a DNN model on the dataset of MNIST and the activation function is ReLU. The number of layers for this model is 20. The source of this file is from the github website of AUTOTRAINER, the address is: <https://github.com/shiningrain/AUTOTRAINER/tree/origin/Test-Master/Motivation>, which is used to validate the stochasticity of whether the bug occurs. And it can generate more Oscillating Loss bug in the training procedure, so the usage of it can cut off the time consumption on the Oscillating Loss buggy samples generation.

minist\_relu\_layer=34.h5: This is the name of a .h5 file which is a DNN model on the dataset of MNIST and the activation function is ReLU. The number of layers for this model is 34. The source of this file is from the github website of AUTOTRAINER, the address is: <https://github.com/shiningrain/AUTOTRAINER/tree/origin/Test-Master/Motivation>, which is used to validate the stochasticity of whether the bug occurs. And it can generate Oscillating Loss bug in the training procedure less than the 20-layer one, but the usage of it can prove that the particular distribution of the time on Oscillating Loss occurrence is not because of the type of model. That distribution is a general principle. And this model is as known as “anotherModel”.

## Appendix II Project details

The details including codes, datasets, training results, analysis can be found at [https://github.com/MiSFiT5/FYP\\_dynamicAUTOTRAINER/tree/master](https://github.com/MiSFiT5/FYP_dynamicAUTOTRAINER/tree/master), which is my personal website in GitHub. There is no files in the branch “main”, so you have to access the branch “master” to see the project.

The raw data set is the original dataset from original AUTOTRAINER, the author share them in the Readme file and the files are stored in Google drive, and the address is <https://drive.google.com/file/d/1AnzEwQZtKXAXA6jo4xGdhRLuAjnUFMLd/view>

## Resume

### Resume

Name: LIU HE YUAN

Gender: Male

Email: [liuheyuan05@gmail.com](mailto:liuheyuan05@gmail.com)

Education:

2016 ~ 2019, High School, 83. Shenyang Middle School.

*Shenyang, Liaoning*

2019 ~ 2023, Bachelor of Science, Macau University of Science and Technology.

*Macao SAR*

Awards:

2022 National College Students E-commerce "Innovation, Creativity and Entrepreneurship" Challenge Competition in Macao,

Provincial second prize,

Provincial Innovation Award,

Provincial Creative Award

Work experience:

2022 06 ~ 2022 08 Volkswagen-Mobvoi Information Technology Co.,Ltd.

*Beijing*

## Acknowledgements

It is an honor to be a student of Professor Rubing Huang and a student of Macau University of Science and Technology.

With the encouragement from my families, friends, teachers, I could be a person with bravery to do the things that I never had done before.

This world was once vast and mysterious, but now it seems to be shrinking and becoming more familiar. We are standing at the threshold where legends have become reality, where the information world is at our fingertips and instantaneous communication networks span the globe. With just a touch of our fingers, we can reach the other side of the Earth. Computer science and information technology have brought unprecedented productivity and prosperity to the entire human civilization. I am grateful to be living in this era and learning in this cutting-edge field, where I might have my own one small step like Armstrong.,