Technical Report on Machine Learning Quality Evaluation and Improvement

4th English Edition

August 1, 2024

Technical Report DigiARC-TR-2024-02
Digital Architecture Research Center
National Institute of Advanced Industrial Science and Technology (AIST)

Technical Report CPSEC-TR-2024002
Cyber Physical Security Research Center
National Institute of Advanced Industrial Science and Technology (AIST)

Technical Report
Artificial Intelligence Research Center
National Institute of Advanced Industrial Science and Technology (AIST)

Foreword

In the project "Research and Development on the Quality Assessment Reference and Testbed of Machine-Learning /Artificial Intelligence Systems" (JPNP20006) commissioned by the New Energy and Industrial Technology Development Organization (NEDO), we are developing Machine Learning Quality Management (MLQM) Guideline [1] to explain the quality of machine learning. While developing the guidelines, we have also been researching and developing techniques for evaluating and improving the quality of machine learning. This report presents the results of the research and development for the 5 years (FY 2019~2023) to technically support the quality evaluation described in the MLQM Guideline.

Table of Contents

1 Introduction

Machine Learning Quality Management (MLQM) Guideline has been developed to clearly explain the quality of various industrial products including statistical machine learning (4th Edition [1]). The fourth edition of the MLQM guideline describes the 14 internal quality characteristics (e.g., Stability of the trained model, Reliability of underlying software system, etc.) for machine learning systems, but techniques for evaluating and improving these internal quality characteristics have not been sufficiently established yet. This report presents the results on survey, research, and development of techniques for evaluating and improving the internal quality characteristics, which have been conducted for supporting the development of the MLQM guideline.

1.1 Overview of this research and development

Figure 1.1 shows the relationship between the machine learning quality evaluation and improvement techniques (the center yellow boxes in Figure 1.1, where the number in each box shows the chapter number explained in this report) that were researched and developed for the 5 years (FY 2019~2023). The relations to the phases of the machine learning model lifecycle and the 14 internal quality characteristics are also shown. Each technique is briefly introduced below, and the details are explained in Chapters 2 ~ 9.

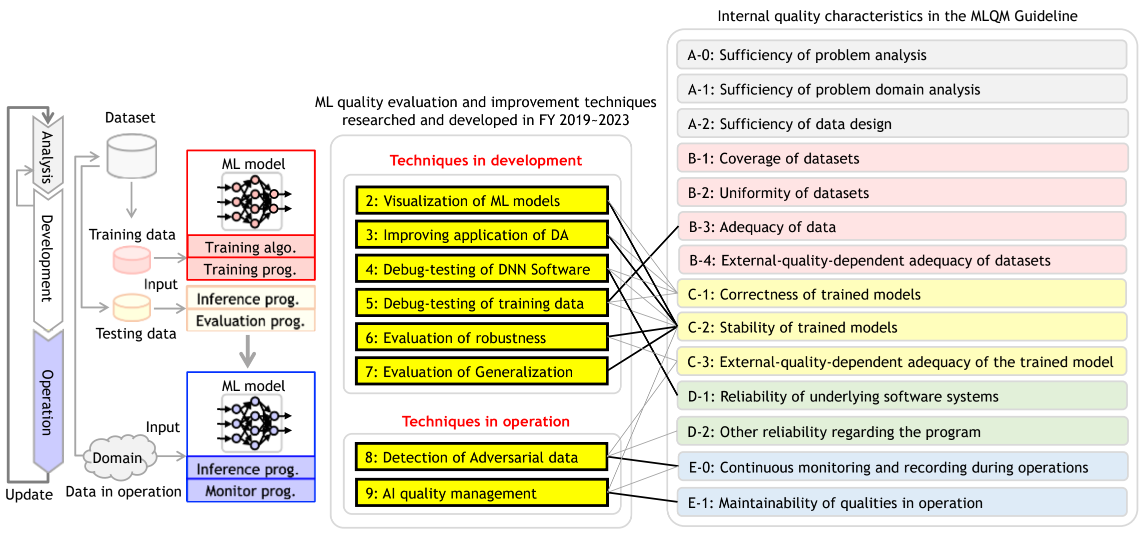

Internal quality characteristics in the MLQM Guideline
A-0: Sufficiency of problem analysis
A-1: Sufficiency of problem domain analysis
A-2: Sufficiency of data design
B-1: Coverage of datasets
B-2: Uniformity of datasets
B-3: Adequacy of data
B-4: External-quality-dependent adequacy of datasets
C-1: Correctness of trained models
C-2: Stability of trained models
C-3: External-quality-dependent adequacy of the trained model
D-1: Reliability of underlying software systems
D-2: Other reliability regarding the program
E-0: Continuous monitoring and recording during operations
E-1: Maintainability of qualities in operation
ML quality evaluation and improvement techniques researched and developed in FY 2019-2023
Techniques in development
2: Visualization of ML models
3: Improving application of DA
4: Debug-testing of DNN Software
5: Debug-testing of training data
6: Evaluation of robustness
7: Evaluation of Generalization
Techniques in operation
8: Detection of Adversarial data
9: AI quality management
Analysis
Development
Operation
Update
Dataset
Training data
Input
Testing data
ML model
Training algo.
Training prog.
Inference prog.
Evaluation prog.
ML model
Input
Domain
Data in operation
Inference prog.
Monitor prog.

Figure 1.1 Machine learning quality evaluation and improvement techniques in this report

– Visualization of Machine Learning Models (in Chapter 2):
  To support the quality evaluation work of machine learning models, we attempted to visualize the difference and comparison results between multiple models and the sensitivity of the workers (annotators and model designers) reflected in each model.

We proceeded with the implementation of a tool to visualize the work procedures of the workers involved in creating the models and their influence on the models with multiple views[2][3]. The main functions visualize the difference between the three items used training data, model structure, and optimization algorithm, as well as the intention of the adjustment by the operator and his/her impression and evaluation of the model. Using this tool, we created visualization results of multiple machine learning model adjustment work histories.

- Improved Quality through Better Application of Data Augmentation (in Chapter 3):
  To improve the data-diversity obtained by data augmentation and increase accuracy and stability in deep learning, we devised new two data augmentation methods, FC-mixup [16] and Latent DA, with simple algorithms, and report the results of their impact on generalization performance in experiments [4]. In addition, for the Latent DA method, we have been developing AdaLASE, for dynamically selecting appropriate layers for the data augmentation. For the FC-mixup method in CNN, there are two kinds for mixing: FC-channel which mixes at the channel level of CNNs, and FC-pixel which mixes at the pixel level. We compare them and their hybrid with the existing method Manifold Mixup, and then confirmed that the proposed method demonstrates higher accuracy than the existing methods. In addition, for efficiently exploring suitable data augmentation policies in early training epochs, we proposed a new metrics based on Affinity and Diversity, and then demonstrated the effectiveness of this approach.

- Debug-Testing of DNN Software (in Chapter 4):
  The failures of DNN (Deep Neural Network) models can be considered from two viewpoints of causes. One of them is the direct cause during inference (by prediction and inference programs) and the other one is the root cause during training (by training and learning programs, training models, and training data). We proposed an indicator and an analysis method for evaluating the presence of bugs in training programs by the internal information (e.g., neuron coverage) of DNN models, and then confirmed that the indicator is useful by experiments [5].

- Debugging and Testing of Training Data (in Chapter 5):
  For the case that failures in DNN (Deep Neural Network) models are caused by training data bias, we researched methods for detecting such bias from two quality viewpoints: model accuracy and model robustness. Then, we obtained the experiment results that the robustness of the DNN models increases without decreasing the correctness by retraining the DNN models by removing data such that causes low or high neuron coverage from the training dataset. It means that the internal states such as neuron coverage is useful for debugging training datasets. Such debag approach is thought to be a combination of a statistical view with software engineering methods.

- Evaluation and Improvement of Robustness (in Chapter 6):
  To evaluate and improve robustness of machine learned models, we report on the results of a survey on methods to measure the maximum safe radius (the maximum value of noise that can be guaranteed not to cause misclassifications) as a measure of robustness for input noise including adversarial examples, and methods to increase the safe radius.

- Estimation of Generalization Error Bounds (in Chapter 7):
  To evaluate the stability of trained models described in the MLQM Guideline, we focused on randomly/worst weight-perturbed generalization error bounds of neural classifiers and demonstrated that they are useful for evaluating the stability by experiments. Here, the weight-perturbed generalization error of a classifier represents the expected value of the misclassification-rate for any input including unseen input when perturbations are added to weights (i.e., training parameters) in the classifier, and random perturbations are randomly selected from uniform distribution with specified range, while worst perturbations are selected towards misclassification within the range.

- Adversarial Example Detection (in Chapter 8):
  To establish a practical method for detecting adversarial examples, we report on the results of a survey on the state-of-the-art adversarial example detection methods and classifies them into four main categories, and then present the results of follow-up experiments on representative methods. Consequently, we confirmed that NIC method shows the highest detection rate. Then, we constructed the NIC framework for detecting adversarial examples based on the NIC method and evaluated it by the Kullback-Leibler divergence for explaining the reason why the NIC method is effective.

- AI Quality Management in Operation (in Chapter 9):
  To maintain quality of machine learning models even for unseen data and/or changing trends during operation, we report on the results of a survey on detection and adaptation methods for changes in input-data distribution over time (e.g., concept drift), and also a survey on the latest unsupervised domain adaptation methods (e.g., label-shift). The surveys include not only supervised methods but also unsupervised/ semi-supervised methods that are promising approaches from the viewpoints of operational costs and practical adaptability.

1.2 Author list

The authors of each chapter are as follows:

- Chapter 1: Yoshinao Isobe (AIST CPSEC)
- Chapter 2: Yuri Miyagi (AIST AIRC)
- Chapter 3: Tomoumi Takase (AIST AIRC)
- Chapter 4: Shin Nakajima (NII)
- Chapter 5: Shin Nakajima (NII)
- Chapter 6: Yoshinao Isobe (AIST CPSEC)
- Chapter 7: Yoshinao Isobe (AIST CPSEC)
- Chapter 8: Yusei Nakashima and Keiichi Nishida (Techmatrix)
- Chapter 9: Yoshihiro Okawa and Kenichi Kobayashi (Fujitsu)

1.3 Acknowledgements

This report is based on results obtained from the project "Research and Development on the Quality Assessment Reference and Testbed of Machine-Learning /Artificial Intelligence Systems" (JPNP20006), commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

2 Visualization of Machine Learning Models

Information visualization is becoming a popular method to support the analysis of the structure and behavior of machine learning models, which are known as black boxes. We have started research on a new method for visualizing machine learning models with the following two objectives:

- Visualization of differences and comparison results between multiple models
  - ➢ Implementation of visualization based on expressions that are easy for humans to interpret and understand
- Visualization of the sensitivity of workers (annotators of training data, designers of model structures) reflected in the model
  - ➢ Proposal of new factors that can be used for quality assessment

In this chapter, we first describe the results of a survey of recent machine learning model visualization techniques. Then, we introduce the results of a prototype visualization tool for observing model and worker information, developed in 2020-2023, and our future implementation policy.

2.1 Survey on methods to support using machine learning

The basic purpose of visualization methods for machine learning is to improve the interpretability of models, and this is closely related to XAI (Explainable AI), which has attracted attention in recent years. There are no definitive definitions or evaluation methods for XAI, however, many papers about the classification of XAI are published, and we can devise visualization objectives and methods along these lines. In [6], the approaches to increase interpretability are classified into four categories:

(1) Total explanation (Approximation of a complex model structure by a simple model)
(2) Partial explanation (Explaining the rationale for decisions about model output results)
(3) Design of explainable models (Creation of readable models at the design stage)
(4) Explanation of the deep learning model (e.g., Highlighting the parts of the image data that the model recognizes)

Especially (2) and (4) have much room for contribution by visualization. These machine learning visualization methods are continuously being studied, and the number of survey papers is increasing due to the diversity of applications and target cases. For example, Hohman et al. [7] described and classified deep learning visualization methods according to the 5W1H elements. It also presents several overall directions and issues in the field of deep learning visualization. Especially "improving interactions for model evaluation" and "improving interpretability through active human involvement in models” are closely related to our research, which aims to develop visualization methods for quality evaluation.

As research on machine learning visualization progresses and becomes popular in the real world, there is a growing tendency for complex analysis to be performed in a single visualization view. In the past, visualization methods basically focused on detailed analyses of single models specialized in either data ([8]) or model structure ([9]). However, in recent years, research has been conducted on combined visualization methods for data and model structure, as well as methods that aim to compare multiple models. The number of elements that make up a machine learning model is enormous, and it takes a lot of time and effort to create visualization results for the number of models and compare them side by side. Besides, the differences in structure and accuracy between the models to be compared are often small and features of the models may be overlooked. Therefore, there is a high need for a visualization method that uses expressions that emphasize the differences so that the differences can be found efficiently within a limited screen. (For example, in [10], the pipeline from data input to output, hyperparameter values, etc. for more than 10 models can be compared on a single screen.)

So far, we have introduced trends and examples of visualization methods regarding the properties and accuracy of the models themselves. In parallel with this, we also investigated how the workers (annotators, designers, and end users) involved in the creation and evaluation of the model interact with the models. In fields such as image recognition, models with accuracy beyond human recognition capabilities have been developed, but there is a persistent suggestion, regardless of the field, that active human intervention is desirable to improve the accuracy of models. There are many papers that discuss the following items regarding the relationship between AI and humans and effective intervention methods in the modeling process:

– Introduction of operations (adjustment and evaluation) to improve the accuracy of the model in the learning process
– Designing an interface that is easy to use and can maintain the motivation of the operator
– Collaboration with related fields such as cognitive science and psychology

As an example, Amershi et al. examined the psychological state of workers who were assigned feedback to evaluate and improve several models [11]. The authors found that the workers preferred to be able to directly tell the correct processing steps to models. They also said that workers get more motivated to give more active feedback when they find their actions are improving the accuracy of the model. Although there seem to be few examples of visualization of such information about the workers themselves and the impact of each worker on the model, it can be adopted as a ground for quality assurance as follows:

– Show that their knowledge is sufficiently reflected in the model's behavior when domain or machine learning experts participated in the creation of the model.
– Indicate which workers' behavior is strongly reflected in the model and use this as a clue to identify elements (training data, parameters, etc.) that should be adjusted.

2.2 Visualization of model structure and worker information

Based on the above research results, we place particular importance on "comparative visualization of multiple models" and "visualization of worker's sensitivity" among machine learning model visualization methods. We proceeded to design a visualization tool with both properties. Figure 2.1 shows an overview of the proposed method. In this study, workers are classified into three types: annotators, model designers, and end users, with a particular focus on model designers.

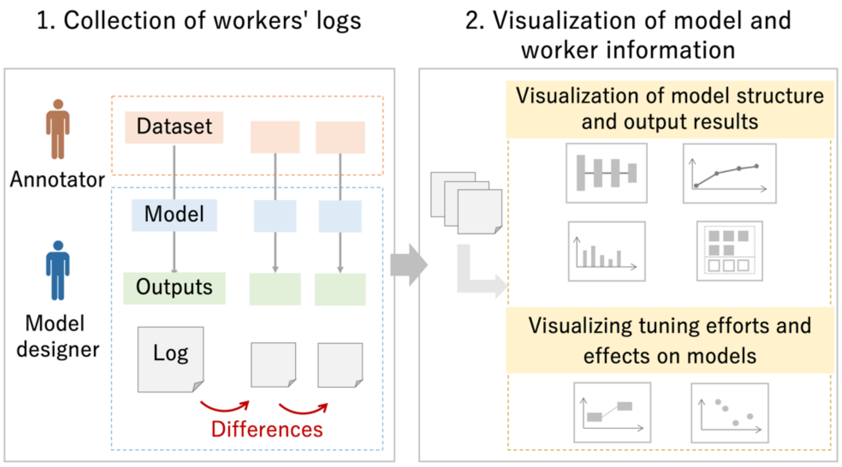

1. Collection of workers' logs
2. Visualization of model and worker information
Annotator
Model designer
Dataset
Model
Outputs
Log
Differences
Visualization of model structure and output results
Visualizing tuning efforts and effects on models

Figure 2.1 Overview of machine learning models and worker information visualization methods

2.2.1 Logging of differences between models

The first step is collecting logs of the structure of the model to be visualized, the process of adjusting the model, and the test results. The current implementation assumes image classification or regression analysis cases. The user extracts from the records by Comet.ml, a machine learning experiment management tool, or from articles submitted to machine learning competitions (codes and results used, and their explanations), the process of parameter adjustment by the model designer, and the results of tests. They get saved as text files. For the annotators, we do not directly collect work logs, but indirectly evaluate their work based on how the model designers selected data and applied preprocessing.

From these logs, we calculate differences between models (the amount of change from the model used immediately before). Differences between models are classified into three categories: training data, model structure, and optimization algorithm, and are calculated for each. The difference in training data is calculated by adding up the data used, the number of classes, and the difference in parameters used for preprocessing. The difference in model structure is obtained by creating pairs of layers that comprise the two models and summing the dissimilarities (differences in layer types and parameters) of each pair (Figure 2.2). For the difference in optimization algorithms, a constant is assigned if the algorithm types are different.

If they are the same, the difference is calculated from the difference in parameters. After obtaining the three types of differences, we obtain the overall change in the model by summing these values. In addition, the difference calculation for each category is obtained by adding the difference value calculated from basic information such as the number of training data and the number of layers in the model to the difference value calculated from factors unique to each case (e.g., processing related to data expansion in the case of image classification). Although this method makes it difficult to directly compare difference values in different cases because the formula for calculating differences changes from case to case, it does allow for comparison of the development process of a particular model within the same case, and of models created by multiple workers.

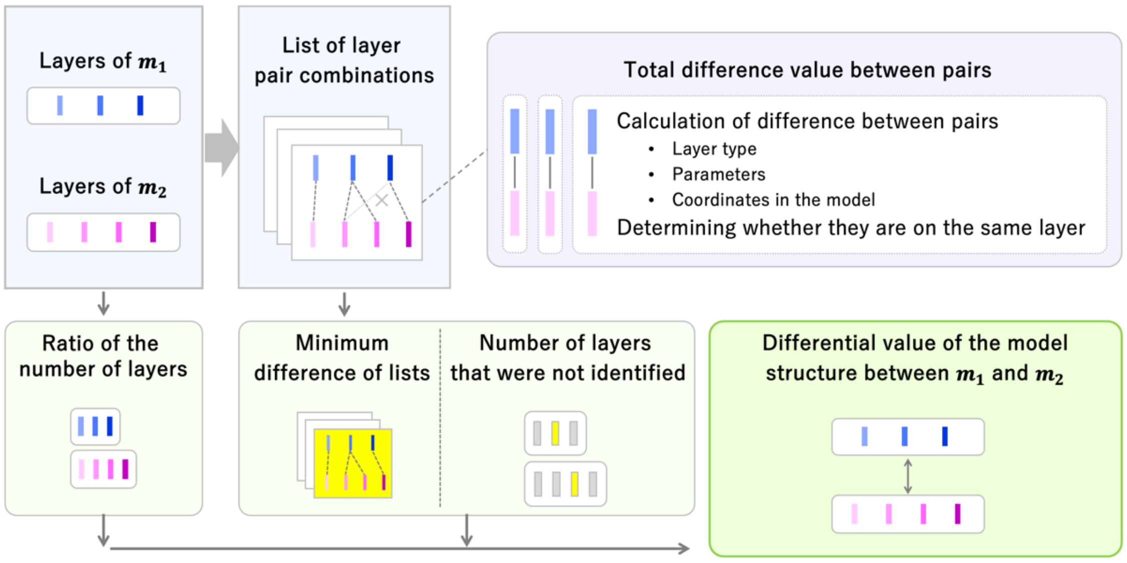

Layers of m_1
Layers of m_2
List of layer pair combinations
Total difference value between pairs
Calculation of difference between pairs
Layer type
Parameters
Coordinates in the model
Determining whether they are on the same layer
Ratio of the number of layers
Minimum difference of lists
Number of layers that were not identified
Differential value of the model structure between m_1 and m_2

Figure 2.2 Model structure difference calculation flow

2.2.2 Visualization of model structure and worker information

We use the collected logs to visualize the structure of the model and information about the workers. The main users of this tool are assumed to be model designers. Considering the possibility that users unfamiliar with visualization may be included, we proceeded with the implementation by combining basic visualization methods (line graphs, bar graphs, etc.) and actively linking them (highlighting related parts, etc.). In FY2020, we implemented views on basic information such as model structure and output results, and in FY2021-23, created to visualize the progress of model adjustments and testing by workers. Figure 2.3 shows the overview of prototype visualization views, which visualize the results of MNIST for two simple models developed in FY2020.

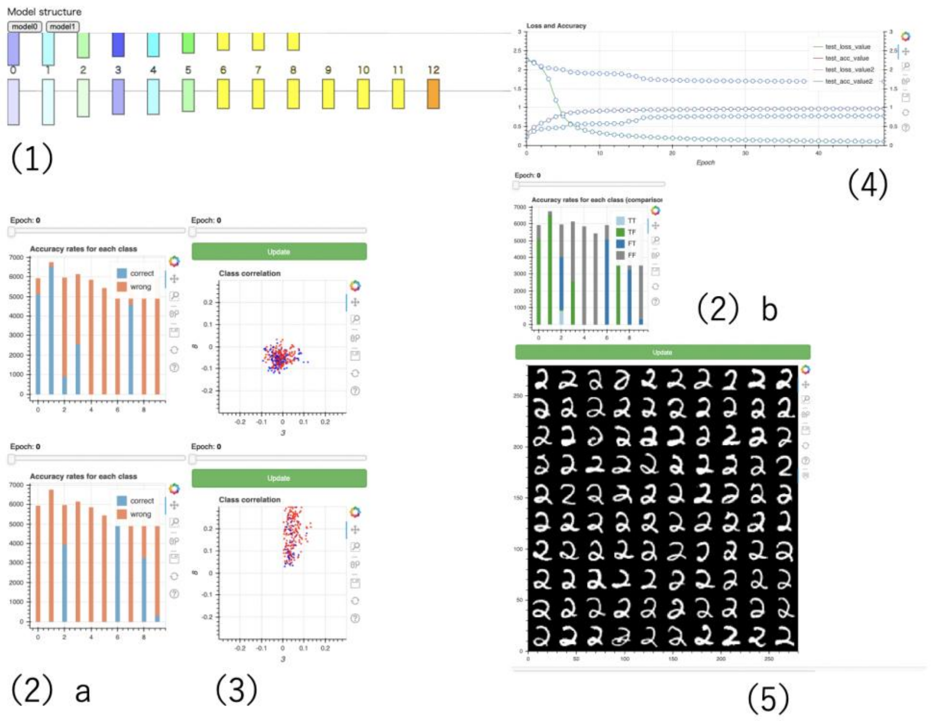


Figure 2.3 Visualization views on model structure and output results

We created the tool on JupyterLab, mainly using the machine learning library PyTorch and the visualization library Bokeh, so that we could compare the features of the two models:

(1) Network of each model structure
(2) Bar graph of output results for each class
  (a) Visualization for each model
  (b) Visualization of the difference between two models
(3) Scatter plot of output result correlation between two selected classes for each model
(4) Line graph of accuracy
(5) Thumbnail list of data classified with particularly high (low) confidence

Figure 2.4 shows an example of the results of classifying the output to MNIST for two models. The horizontal axis represents the class from 0 to 9, and the vertical axis represents the amount of data. The color-coding of each bar represents the combination of correct (T) and incorrect (F) answers for the two models, for example, where TF (FT) means that only model 1 (2) correctly classified. Immediately after the start of learning (Figure 2.4, left), model 1 had a high percentage of correct answers in classes 0, 1, and 7, and model 2 had a high percentage of correct answers in classes 2, 6, and 8, indicating that each model had different strengths. At the advanced stage of learning (Figure 2.4, right), both models had high percentages of correct answers in many classes. Besides, model 1 has a high percentage of correct answers, including classes 3, 4, 5, and 7, which model 2 is not good at, indicating that model 1 is more advanced in learning than model 2 at this stage.

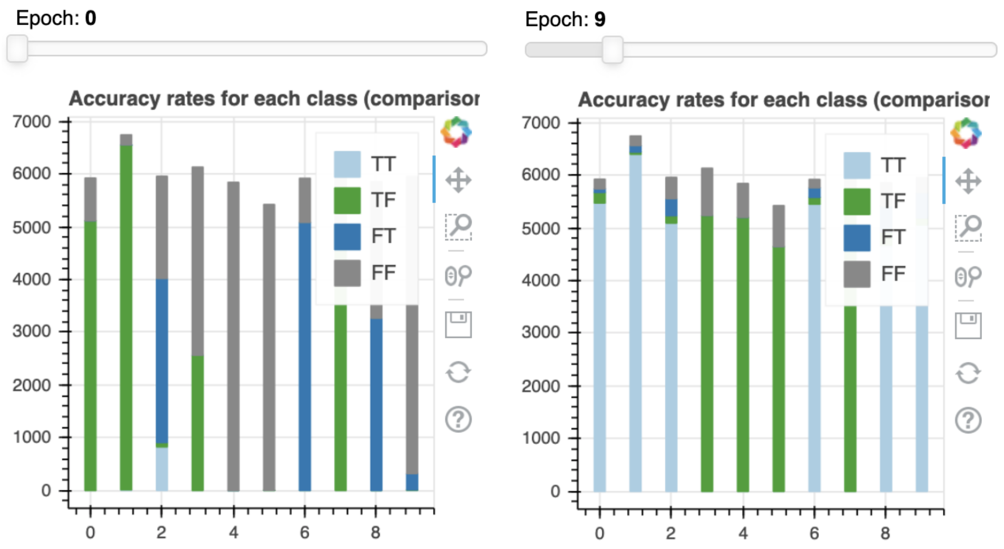


Figure 2.4 Examples of comparing the output results of two models

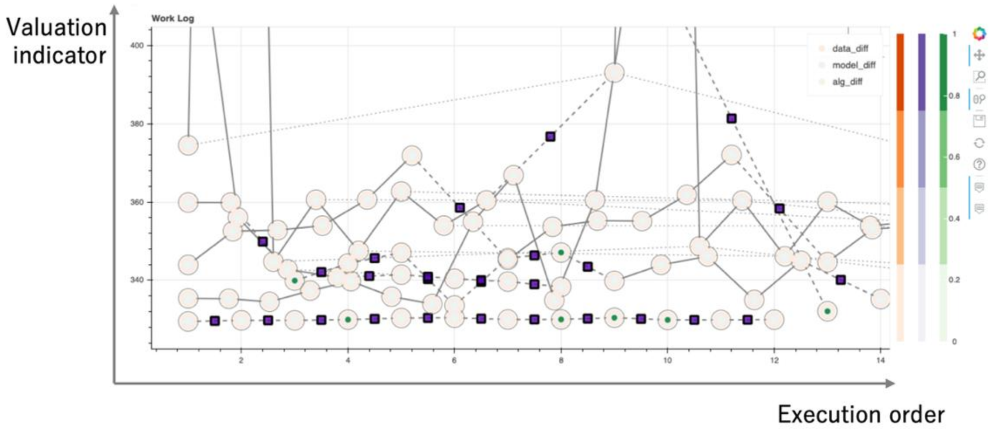


Figure 2.5 Overall view of the model adjustment work history visualization

Figure 2.5 shows the structure of the time-series visualization view of the model adjustment work history implemented from FY2021 to FY2023. The logs about the model structure and test results are used as input for visualization by network graphs.

The graph is arranged with the vertical axis as the evaluation index of the model and the horizontal axis as the time axis. The graph is composed of alternating nodes of different types and shapes as shown in Figure 2.6, which are defined as "model nodes" and "intention nodes". The model nodes represent the adjustments and test results for one of the models used. The interior of the model node is divided into three colored rings. The color of the ring represents the difference from one of the previously used models, which from the outside means the difference in training data (orange), the difference in model structure (purple), and the difference in optimization algorithm (green). Higher saturation indicates a larger difference value from the compared model, and the correspondence between color and difference value can be checked with the color bar placed at the right end of the view. Usually, model nodes are placed at equal intervals along the horizontal axis to indicate the order in which the models were used. However, in cases where multiple settings were prepared for a particular parameter and comparative experiments were conducted simultaneously, multiple models used in parallel can be grouped and placed in close proximity. In this case, the past models to be used in the difference calculation are selected as shown in Figure 2.7. In this example, one model is used in Stage 1, three models are set up in Stage 2, and two models are created and used in parallel in Stage 3. If multiple models are not used in parallel in the previous phase, as in phase 1 to phase 2, they are compared to the models in the immediately preceding phase (blue arrows). When multiple models are candidates for comparison, as in steps 2 through 3, priority is given to comparisons with models with similar structures (red arrows) or with models that were more accurate (green arrows). These settings allow for observation of "the process of development of a model of a particular structure through detailed parameter adjustments" and for comparison and evaluation by focusing on models with parameter settings that are likely to be employed continuously with good accuracy.

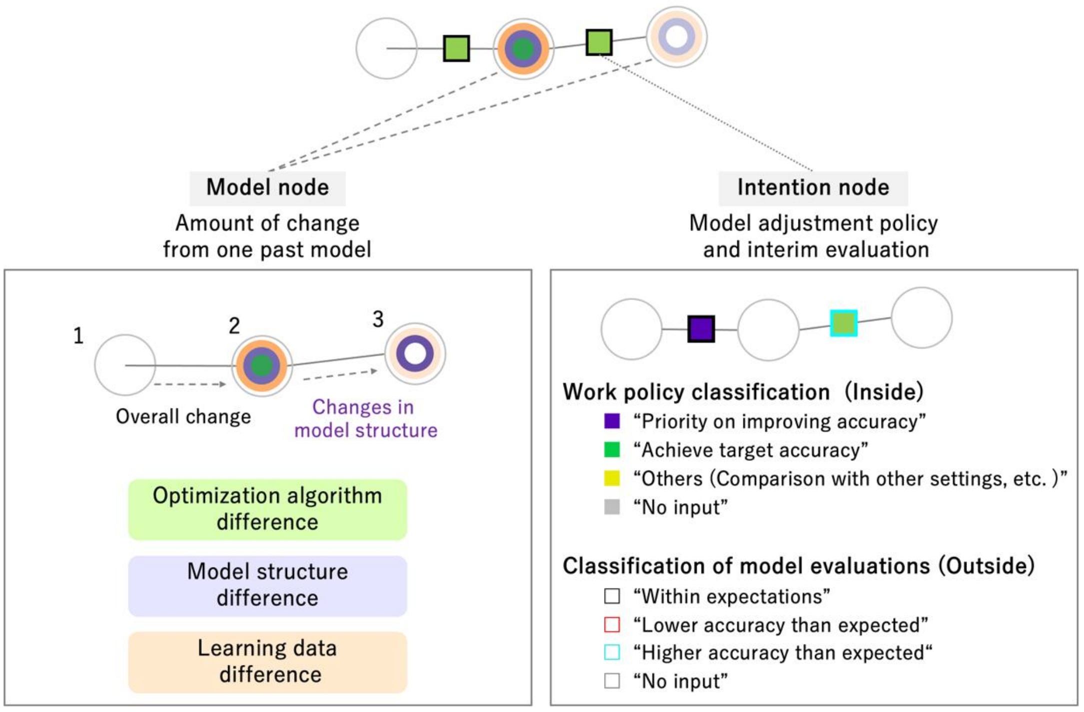


Figure 2.6 Composition of the graph of model adjustment work history

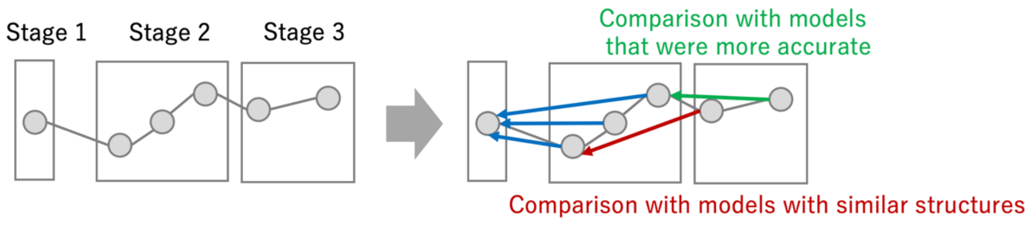


Figure 2.7 Image of selection of model nodes to be compared

The intention node is placed between the two model nodes and is assigned two different colors: inner and outer. The inner color represents the intention or policy of the worker in creating the model immediately following that intention node. Information on intentions and policies can be selected from a list of pre-defined options during the logging phase, and the color of the node changes accordingly (Figure 2.6). The outer part of the intention node, on the other hand, visualizes the worker's impression and evaluation of the immediate model. The frame gets colored red (blue) when "the accuracy of the created model has fallen (risen) below the expected level" by adding a note to the log. This means the tool can highlight areas of unexpected results to indicate points where work policies should be analyzed and modified.

These nodes are connected by three types of edges (solid, thick dotted, and thin dotted lines) to indicate the order in which the model was used and the flow of development (Figure 2.8). Solid edges connecting multiple model nodes mean that they are a group of models tested in parallel. The thick dotted line corresponds to the timing when the parallel experiment was completed, meaning that the next model was used after checking the results of the previous model and applying changes to the settings. Therefore, the intention node is placed in the middle of the thick dotted edge. Thin dotted edges represent model derivation relationships, such as when subsequent models take over parameter settings. This does not necessarily connect models that have been used in succession, but it does confirm the long-term developmental relationships when models of a particular structure are fine-tuned and used.

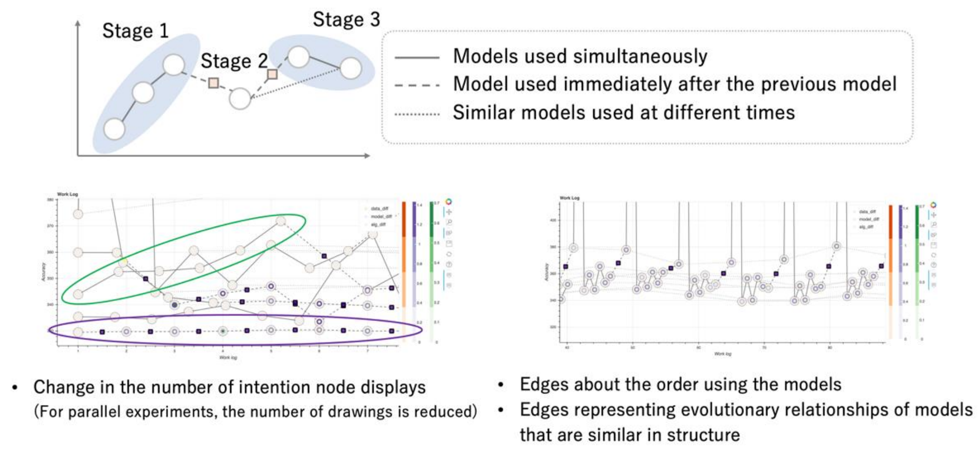

Stage 1
Stage 2
Stage 3
Models used simultaneously
Model used immediately after the previous model
Similar models used at different times
Change in the number of intention node displays
(For parallel experiments, the number of drawings is reduced)
Edges about the order using the models
Edges representing evolutionary relationships of models that are similar in structure

Figure 2.8 Edge types and display examples

Using these functions, we created visualization results for the work history data of the three cases.

(1) Model tuning history during user testing in the study of machine learning model visualization tools
(2) Model tuning history for one participant in the Kaggle competition Digit Recognizer
(3) Model tuning history for 6 participants in the Kaggle competition Prediction of Wild Blueberry Yield

Using the data in (1), we visualized the accuracy and the amount of change in model structure when multiple ResNet models trained on CIFAR-100 were tested in sequence (Figure 2.9).

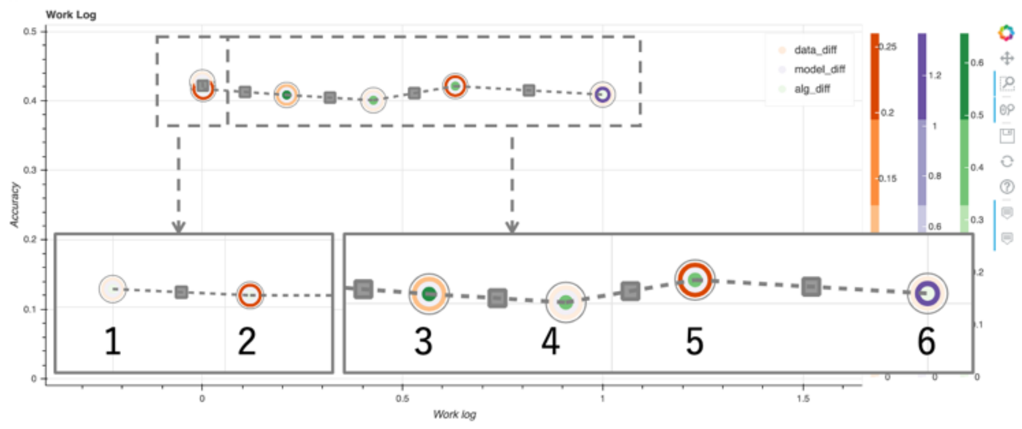


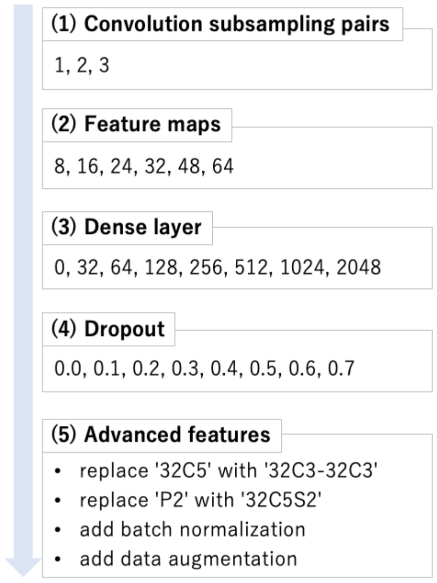
Figure 2.9 An example of visualization of a worker's model adjustment history

Table 2.1 Parameters of the model used for work history visualization

Index	l	m	p	a	d
1	0.1005	0.9	0	0.4	18
2	0.1005	0.9	0.45	0.4	18
3	0.06	0.5495	0.409	0.55	18
4	0.06	0.919	0.409	0.55	18
5	0.0335	0.919	0.2955	0.287	18
6	0.0335	0.919	0.2955	0.287	34

Referring to the hyperparameter tuning scenario conducted in [12], the model was trained and tested 6 times and logged while changing the parameters as shown in Table 2.1. l is the learning rate, m is the momentum value. p and a are the erasing probability and max erasing area when random erasing was applied to the training data. d is the depth of the ResNet model used. Note that in this case, all the intent nodes are grayed out because no information on the intent nodes was entered. The numbers in Figure 2.9 correspond to the Indexes in Table 2.1. In 2 and 5, the outermost rings in the model node that represent changes in the training data are highlighted in red. In going from model 1 to 2 and from 4 to 5, this reflects a significant change in p and a, the parameters related to the training data. Similarly, in 3, 4, and 5, the nodes are highlighted in green, indicating changes related to the optimization algorithm. Specifically, this reflects changes in l and m. In addition, only in 6, the node is colored purple, indicating that a change in the model structure difference has occurred.

Next, we present an example of visualization for the participation records of the machine learning competition in (2): We selected one article of the code and its explanation of the parameter comparison experiment submitted to Digit Recognizer, one of the competitions held for beginners at Kaggle, and information on parameter settings and obtained accuracy was entered into the visualization tool. The experiment was divided into five stages as shown in Figure 2.10. Except for the fifth step, the process is iterative, focusing on a particular element (parameter or model structure) to find the optimal setting, and then taking over the setting to verify the next element. Figure 2.11 shows the visualization results. Since the adjustments mainly change the model structure and the Dropout rate, dark purple and green colors can be seen in the colors of some model nodes. The change in the color of the intent node indicates that the first half of the node is green, indicating that the parameters are being adjusted while considering the constraints of the computational resources, and the second half, which is purple, indicates that the setting that improves accuracy is being obtained in priority. If we look at the thin dotted edges that represent the derivation relationship of the models, we can see that the model that is the derivation source (the model node at the left end of the thin dotted edge) is not necessarily the one that achieved the optimal accuracy among the models experimented in parallel (the model nodes connected with solid lines). As with the intent node, this reflects the fact that, in consideration of the cost of computational resources, priority was given to selecting "a setting that achieves some high accuracy but does not make the cost too large".



(1) Convolution subsampling pairs
1, 2, 3

(2) Feature maps
8, 16, 24, 32, 48, 64

(3) Dense layer
0, 32, 64, 128, 256, 512, 1024, 2048

(4) Dropout
0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7

(5) Advanced features
- replace '32C5' with '32C3-32C3'
- replace 'P2' with '32C5S2'
- add batch normalization
- add data augmentation

Figure 2.10 Parameterization of the adjustment process for models submitted to Digit Recognizer

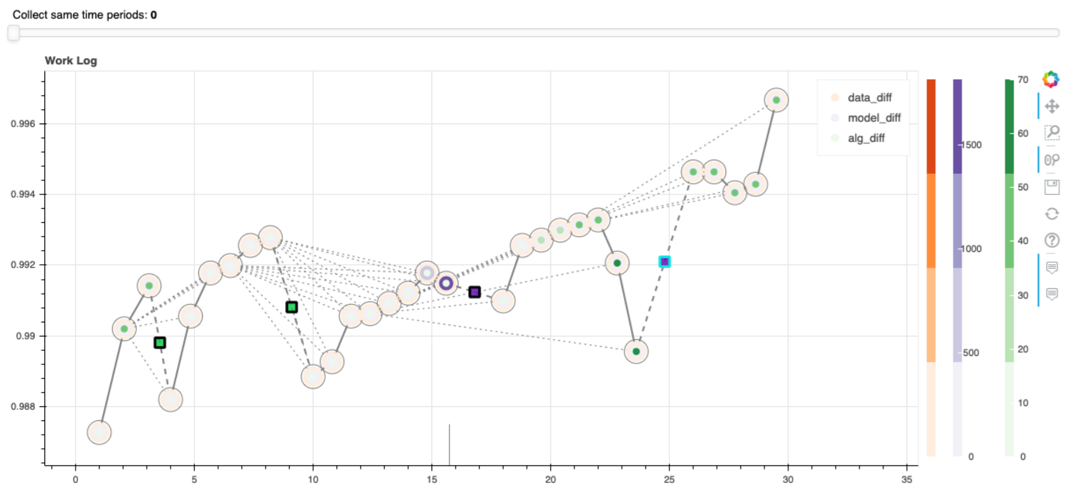


Figure 2.11 Visualization results of the adjustment process of models submitted to Digit Recognizer

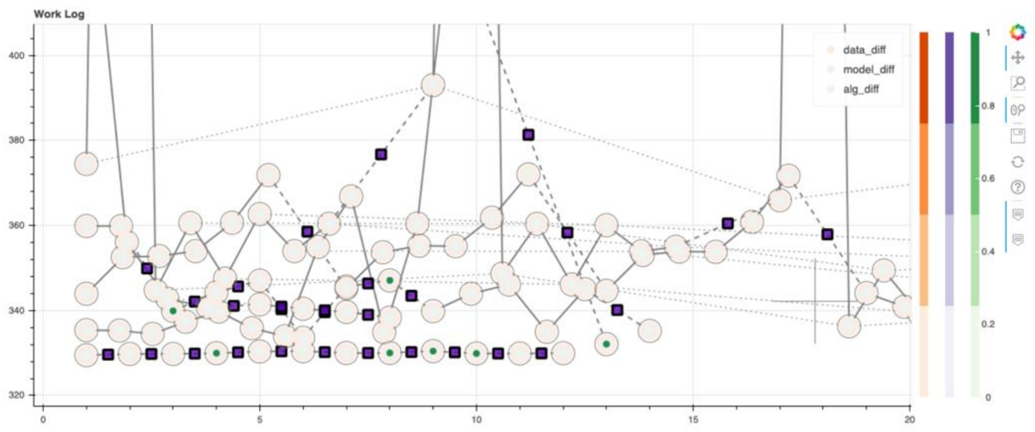


Figure 2.12 Example visualization of Prediction of Wild Blueberry Yield participant's work record (for 6 participants)

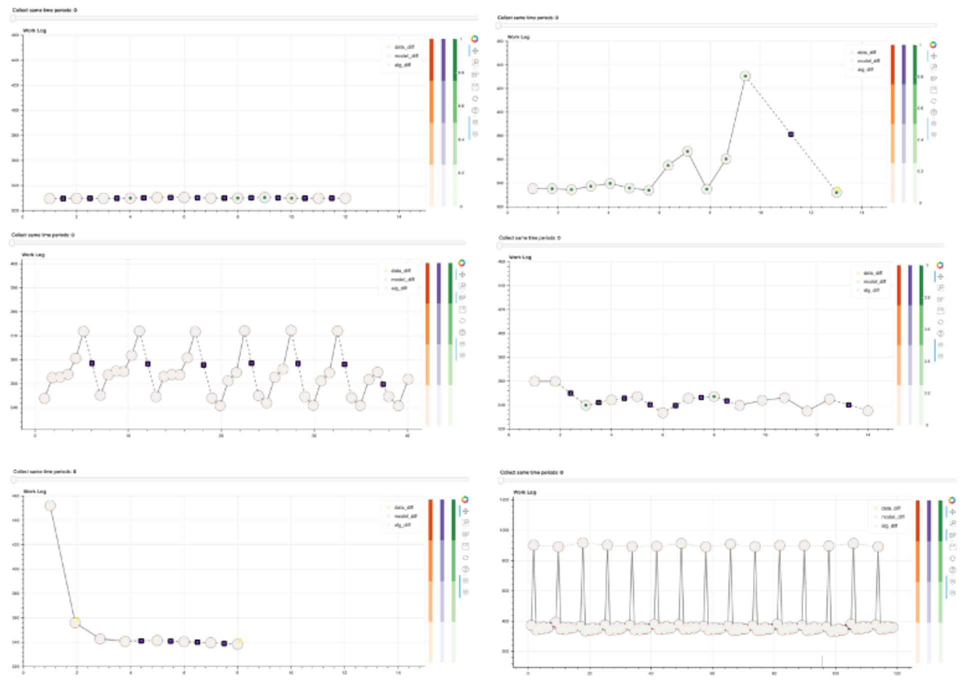


Figure 2.13 Example visualization of Prediction of Wild Blueberry Yield participant's work record (individual)

The data in (3) are work records for six participants in Kaggle's competition Prediction of Wild Blueberry Yield, and Figure 2.12 and Figure 2.13 show the visualization results. In this case, unlike (1) and (2), MAE is used as the evaluation index of the model, so smaller values on the vertical axis indicate better results. Figure 2.12 depicts the records of six people together, while Figure 2.13 depicts the records of one person at a time. Differences can be seen in the overall shape of the graphs created and in the number of nodes, indicating that the flow of MAE changes and the number of times the test was performed differed greatly from worker to worker. On the other hand, there are large differences in the length of the work history, and in Figure 2.12, as a result of aligning the beginning of the work history and arranging the nodes in order from the left, there are too many nodes concentrated at the extreme left end of the screen, making it difficult to compare the latter part of the work history. In addition, because the amount of change in some models is extremely large, only the corresponding model nodes are colored darkly, and the color of the nodes of other relatively little-changed models does not change much, making it difficult to observe detailed difference values.

2.3 Future work

In future work, we would like to expand the visualization function with the following policy. First, we will introduce more detailed difference value calculations for the data used in this study to enable comparison of various items. Then we would like to compare the work patterns of a large number of workers and visualize the similarity between models or workers, and the classification results in an overhead view. By observing these visualization results, we would like to be able to estimate the skill level of workers and classify the characteristics of work (work patterns). Furthermore, we aim to present the results of the recommendation of model improvement measures along with the calculation results of the difference values and the contents of the visualization results.

Although this method currently assumes manual model design, it would be useful to extend it to a wider range of cases by combining it with NAS (Neural architecture search) and other methods.

In order to evaluate this method, we would also like to conduct user tests on the functionality of the visualization in the form of observing the generated visualization result images and the visualized contents.

3 Improved Quality through Better Application of Data Augmentation

This chapter describes the results of developing a new method for applying data augmentation in neural network learning and evaluating its effect to learning quality through experiments.

3.1 Research purpose

Data augmentation is a technique to increase the number of samples by adding deformations to the data, and it is highly effective in deep learning, which has a tendency of performance degradation when the number of training samples is small. On the other hand, the effectiveness of data augmentation strongly depends on the data used, so the selection of data augmentation methods and the parameters of each method must be set appropriately. However, theoretical analysis of data augmentation is difficult, and general ways to use it have not yet been established. This leads to unintentional and inappropriate use, which in turn compromises the quality of learning. In fact, there are many cases that training performance is degraded by setting inappropriate values for the amount of deformation of each data augmentation method, such as mask size or rotation angle, or where the user is puzzled as to what data augmentation method to select for the actual data to be used.

Therefore, to move away from the empirical use of data augmentation, this study focused on data diversity. Increasing diversity is the essential goal of data augmentation, and it has been demonstrated in the work of [13] that increasing diversity has a significant impact on improving generalization performance. Recently, a technique called RandAugment [14], which dynamically applies randomly selected operations from multiple data augmentation operations during training, has attracted much attention, but while it greatly improves diversity, effectively using it is not easy because many parameters need to be adjusted. In this study, we proposed the following two new methods for applying data augmentation related to data diversity, and improved the algorithms and evaluated their performance.

- We apply data augmentation at various layers of the neural network, including hidden layers, and perform automatic optimization of the applied layer (Section 3.2).
- We improve the Mixup method [15], a promising data augmentation method, and propose a new way to mix samples (Section 3.3).

Another practical problem in using data augmentation is the computational cost. There are many types of data augmentation methods, and each method has its own hyperparameters, such as rotation angle, as mentioned above. In order to select the appropriate data augmentation, training must be performed many times, which requires high computational cost. Therefore, we proposed the following method to efficiently find appropriate data augmentation.

- A new data augmentation metric that takes into account Affinity and Diversity is used to successfully explore data augmentation policies in a short number of training steps

(Section 3.4).

3.2 Improved application layer for data augmentation

3.2.1 Data augmentation at hidden layers

Generally, data augmentation is considered to be applied to input data, but in neural networks, it is also possible to apply data augmentation to hidden layers. There are several previous studies on this subject, but most of them are not versatile methods, such as Manifold mixup [17], which limits the method to mixup [15], or other methods that require specific networks and datasets. In this study, we considered applying various data augmentation methods used for image data, such as affine transformation and mask processing, in the hidden layers. Since features are extracted hierarchically in CNNs, data augmentation can be applied in various layers randomly selected for each minibatch to generate a wide variety of samples. As with application to input images, data augmentation can be applied to the feature maps obtained at the intermediate layers, making implementation easy.

An example of actual application of mask processing and translation to an input image and feature map is shown in Figure 3.1. Here, a sample is input to the model in training, and the images are shown in the upper row, aligned in size, immediately after data augmentation was applied at different layers with the same parameters (mask position and translation amount). The feature maps in the final layer of the sample are shown in the lower row. They are different images depending on the layer where the data augmentation was applied. This result shows that data augmentation at various layers leads to an increase in the diversity of the generated data and results in learning different from when data augmentation is applied only to the input data.

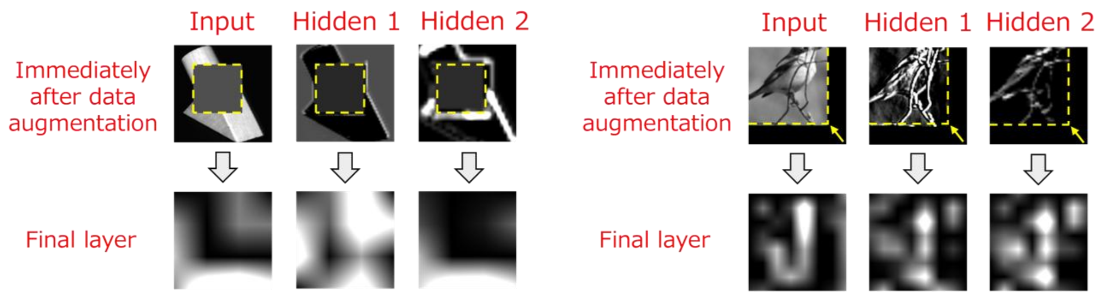

Input
Hidden 1
Hidden 2
Immediately after data augmentation
Final layer
Input
Hidden 1
Hidden 2
Immediately after data augmentation
Final layer

Figure 3.1 Example of applying data augmentation to input images and feature maps obtained at hidden layers

To compare the performance of data augmentations in the input layer and that in feature maps, we used various data augmentations and obtained test accuracies for models trained with supervision. Here, WideResNet28-10 was trained for 200 epochs using the CIFAR-10, Fashion-MNIST, and SVHN (without extra data) datasets. The results are shown in Figure 3.2. In each figure, the horizontal axis represents the accuracy [%] of the conventional method (Input DA) and the vertical axis represents the accuracy of the proposed method (Latent DA). As can be seen from these results, the proposed method tends to show higher accuracy than the conventional method, and the proposed method presented higher accuracy even in cases where the conventional method presented lower accuracy, such as the results using Crop. These results indicate that the diverse samples generated by the application of data augmentation to random layers are effective in improving performance.

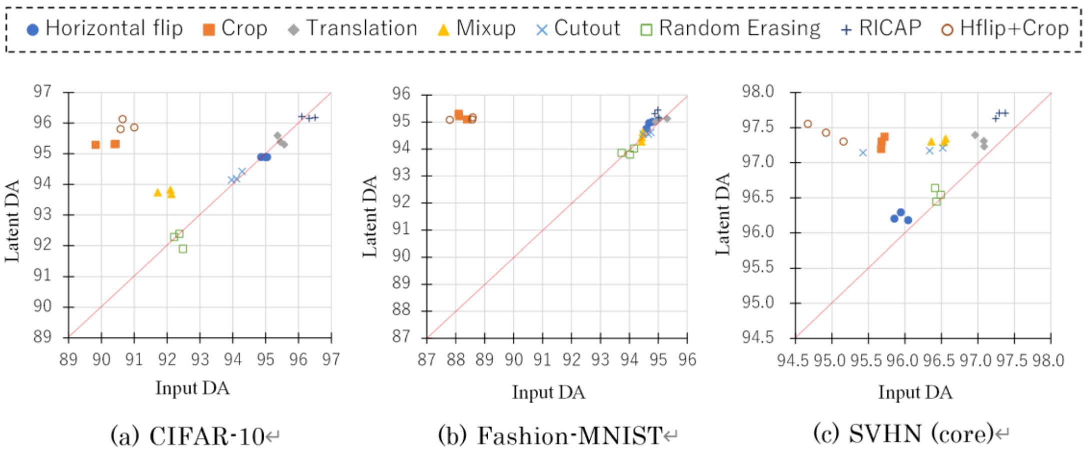


Figure 3.2 Comparison of test accuracy between input DA and latent DA

3.2.2 Selecting appropriate layers for data augmentation

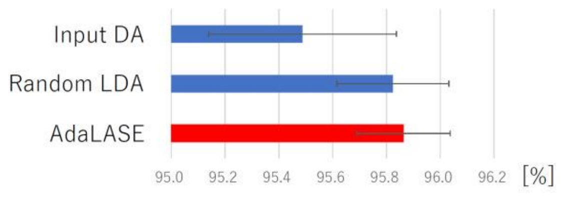
Although previous studies have shown that data augmentation at hidden layers is effective, the question arises as to which layer is optimal for data augmentation. Although it is possible to find the optimal layer by repeatedly training with different layers of data augmentation and comparing the values of validation accuracy, it is an inefficient and impractical method because it increases the overall training time. Therefore, in this study, we worked on developing a method to dynamically discover the optimal layer for data augmentation in a single training session.

The approach is to prepare a parameter called the acceptance ratio for each layer, update the acceptance ratio during training, and apply data augmentation in the layer selected probabilistically according to the acceptance ratio. The updating of the acceptance ratio is done using the gradient descent method as shown below.

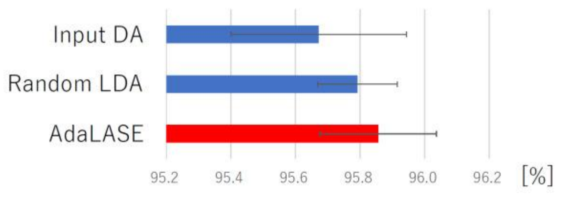
$$q_l \leftarrow q_l - \eta \frac{\partial L_{val}}{\partial q_l},$$

where q_l is the acceptance ratio of layer l, L_{val} is the value of the error when the validation data is input, and η is the step width of the update. In practice, the values of the validation data should not be included in the algorithm, so the update is performed by creating pseudo-validation data with the training data with data augmentation. In the initial state of training, all acceptance ratios are set to equal values so that the sum is 1, and the acceptance ratio is updated for each minibatch. This optimization is expected to improve the generalization performance by increasing the acceptance ratio of layers suitable for data augmentation and decreasing the acceptance ratio of layers unsuitable for data augmentation.

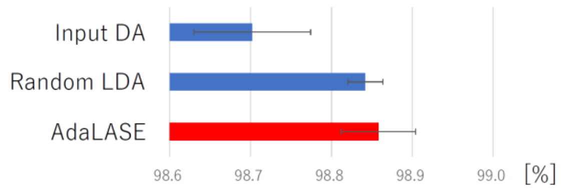
We named this method Adaptive Layer Selection (AdaLASE) and compared its test accuracy to that of conventional methods. The data was CIFAR-10 and MNIST, the model was ResNet18 and a multilayer perceptron (MLP) with one intermediate layer, data augmentation on input, data augmentation on random layers, and AdaLASE. Cutout and Mixup were used as augmentation methods. In the results in Figure 3.3, the mean and standard deviation of the accuracy for five different initial values are shown for each method. These results show that AdaLASE can perform as well as or better than conventional methods. Future plans include a detailed analysis of how layers are selected and whether AdaLASE is functioning properly by looking at the change in acceptance ratio during training.



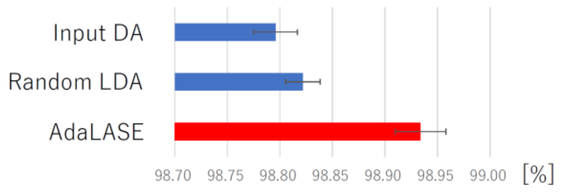
(a) CIFAR-10, Cutout, ResNet18



(b) CIFAR-10, Mixup, ResNet18



(c) MNIST, Cutout, MLP



(d) MNIST, Mixup, MLP

Figure 3.3 Comparison of test accuracy between AdaLASE and conventional methods

3.3 Proposal for a new mixing method by improving Mixup

3.3.1 Feature Combination Mxup

In actual training, data augmentation often involves the simultaneous use of multiple methods, such as cropping, rotating, and flipping. Therefore, we focus on the compatibility between methods when multiple methods are used in this way, and in particular, we consider discussing the compatibility from the viewpoint of data diversity. As a first step in this approach, we propose a new method that is a variant of an existing method and use it simultaneously with the original method to increase the diversity of the data generated and improve performance. The method for formulating the diversity is described in the work of [13]. In this study, we first compare only the accuracy and verify whether the proposed method improves the performance.

Here, we have improved Mixup [15], one of the data augmentation methods. This method generates a new sample by linear interpolation of two samples, and takes the same ratio of linear interpolation for each of the input values and labels, as shown in the following equation.

$$\begin{cases} \tilde{x} = \lambda x_i + (1-\lambda)x_j \\ \tilde{y} = \lambda y_i + (1-\lambda)y_j \end{cases},$$

where (x_i, y_i) and (x_j, y_j) represent the input values for the i-th and j-th samples, and λ is the mixing ratio sampled from the beta distribution. Mixup was chosen as the subject of this study because of its versatility and because it can be used for many numerical data, including not only images but also time series data, and therefore, the impact of improving the Mixup method would be significant.

An improved version of mixup so that it can also be performed in a hidden layer of a neural network is called manifold mixup [17], but both mixups generate samples only in a localized region of the data distribution, on a line segment between two points, and are inappropriate for data sets with distributions in which the properties of the points on that line segment vary nonlinearly.

The Feature Combination Mixup (FC-mixup) proposed in this study [16] is a method of mixing samples in a different way than conventional mixups, and is outlined in Figure 3.4. Suppose that two samples A and B in the same minibatch output the features Z_{A} and Z_{B} in a randomly selected layer. d is the total number of features in that layer, FC-mixup randomly extracts and combines $d\lambda$ features from Z_{A} and $d(1-\lambda)$ from Z_{B} and generates a new sample Z_{X}. Since the number of possible combinations is large for a single value of λ, different data can be generated depending on the random number, and thus samples can be generated over a wide range of the data distribution. FC-mixup is expressed as follows, so Z_{A} and Z_{B} are mixed so that this equation is satisfied.

$$|Z_{\mathrm{A}} \cap Z_{\mathrm{X}}| = \mathrm{d}\lambda$$

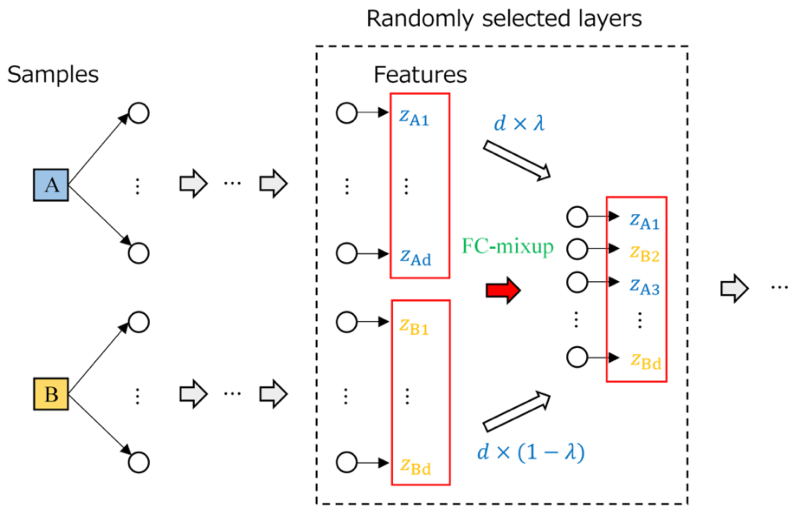


Figure 3.4 Overview of FC-mixup

This technique of generating new data by combining the parts of two data sets is also found in Puzzle Mix [18], but the target is limited to the input image. A similar technique is used in Adversarial mixup resynthesis [19], but it is limited to use in autoencoders, while FC-mixup is designed for more general use.

In our experiments, we used several multi-class classification datasets to compare the classification accuracy of the test data between the conventional method (no data augmentation, mixup at the input layer [15], Manifold Mixup [17]) and the proposed method (FC-mixup, Hybrid method). MNIST, CIFAR-10, CIFAR-100, and SVHN were used for the data. Models used were a multilayer perceptron (MLP) with one intermediate layer, a small convolutional neural network (CNN) and ResNet18. In addition to the full-size data, experiments were conducted on reduced data with 1,000 randomly selected samples. Means and standard deviations in five trials with different initial values were obtained and compared.

Table 3.1 shows the results, with boldface letters representing the highest accuracy. The results show that the proposed method FC-mixup (channel-wise) produces the highest accuracy in many cases; it outperforms existing methods such as Manifold mixup and CutMix [20], and the results demonstrate the high performance of FC-mixup. Since it is a versatile data augmentation, the use of FC-mixup in addition to Manifold mixup is considered to be practically useful.

Table 3.1 Comparison of test accuracy on multi-class classification data

	CIFAR-10 ResNet18	SVHN ResNet18	CIFAR-100 ResNet18
No mixup	93.75 ($\pm$0.42)	96.28 ($\pm$0.05)	74.61 ($\pm$0.29)
Input mixup	95.20 ($\pm$0.32)	96.72 ($\pm$0.13)	76.84 ($\pm$0.27)
Manifold mixup	94.92 ($\pm$0.45)	97.10 ($\pm$0.08)	**78.58 ($\pm$0.39)**
FC-mixup (channel-wise)	**95.23 ($\pm$0.16)**	**97.12 ($\pm$0.06)**	78.54 ($\pm$0.20)
CutMix	95.04 ($\pm$0.15)	96.96 ($\pm$0.14)	77.38 ($\pm$0.29)
	CIFAR-10 (1000) small CNN	SVHN (1000) small CNN	
No mixup	58.98 ($\pm$0.93)	72.45 ($\pm$1.13)	
Input mixup	61.09 ($\pm$0.15)	74.44 ($\pm$0.57)	
Manifold mixup	60.04 ($\pm$0.39)	74.18 ($\pm$0.52)	
FC-mixup (channel-wise)	**62.60 ($\pm$0.91)**	**78.05 ($\pm$0.40)**	
CutMix	58.29 ($\pm$0.59)	72.25 ($\pm$0.58)	

3.3.2 Feature Combination Mxup

The FC-mixup proposed in Section 3.3.1 was that samples can be mixed per unit for MLP and per channel for CNN. However, for CNNs, it is also possible to mix samples per pixel. Channel-wised FC-mixup is called FC-channel, and pixel-wised FC-mixup is called FC-pixel. Figure 3.5 shows the difference between them. In FC-channel, each sample channel is combined to make a new sample channel, whereas in FC-pixel, each sample pixel is combined to make a new sample pixel. The selection of pixels to be mixed in FC-pixel is the same for all channels.

The results of visualizing the samples generated by Manifold mixup and the two FC-mixups are shown in Figure 3.6. It shows the feature maps generated by mixing the two samples at some intermediate layer, and although each sample has multiple channels, three of them are taken up and shown. From these results, it can be seen that the feature maps generated by the Manifold mixup and the two FC-mixups are very different. When Manifold mixup is used, an image is generated by superimposing the two samples. When FC-channel is used, it can be seen that for each channel, a feature map from one of the original two samples is employed. In this example, the feature maps of Sample 2 for ch1, Sample 2 for ch2, and Sample 1 for ch3 were used to generate a new image. When FC-pixel is used, the image of each channel appears to be significantly deformed because a new feature map is generated by adopting pixels from either sample.

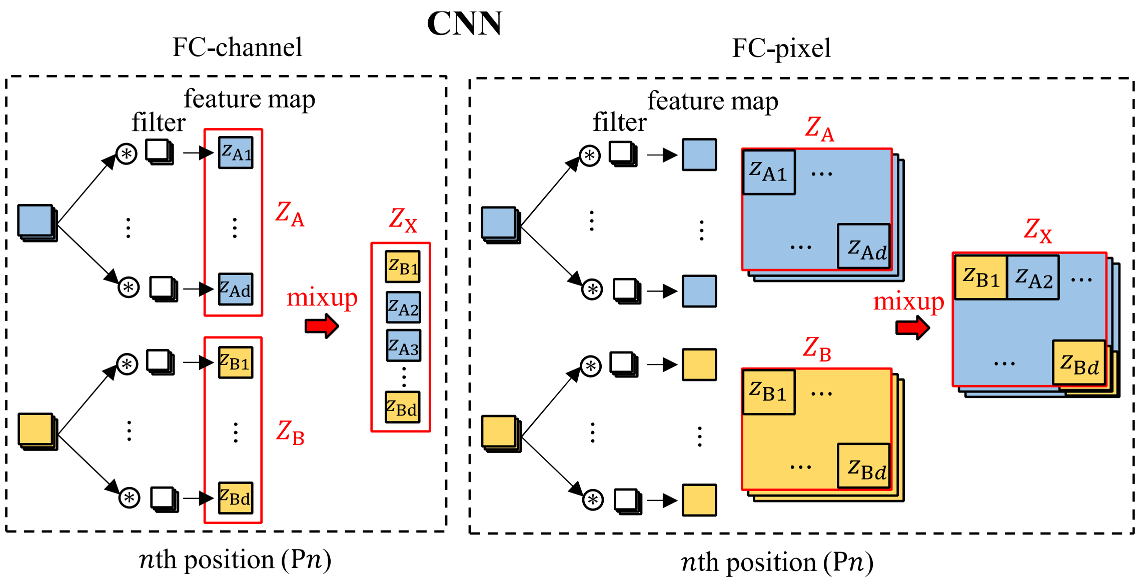


Figure 3.5 Two types of FC-mixups in CNN

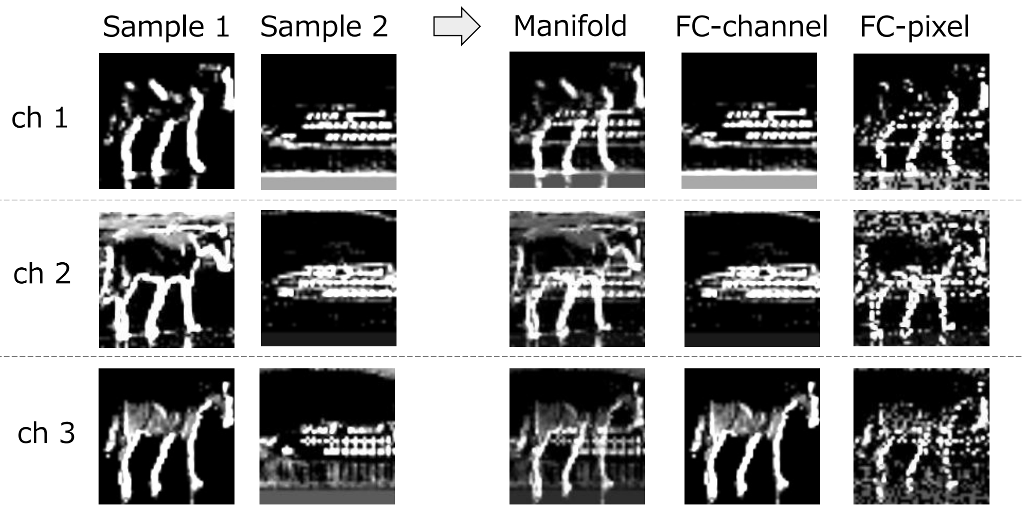


Figure 3.6 Visualization of feature maps generated by each mixup

3.3.3 **Hybrid Use of Mixup**

To increase the diversity of the generated data, we consider the Hybrid method, which combines FC-mixup and Manifold mixup [17]. Here, two hybrid methods are possible, as shown in Figure 3.7. Hybrid 1 is a method in which Manifold mixup and FC-mixup are applied separately to the same two samples, the images obtained are multiplied by 0.5, and then added together. The Hybrid 2 method first applies FC-mixup to generate images, and then mixes two of the generated images by using Manifold mixup.

FC-channel and FC-mixup by themselves and in two different Hybrid methods were compared with the highest accuracy of the existing methods. The results in Table 3.2 show that the proposed method has higher accuracy than the existing methods; the Hybrid method did not always have the highest accuracy, and which FC-mixup was optimal depended on the dataset, model, and other conditions.

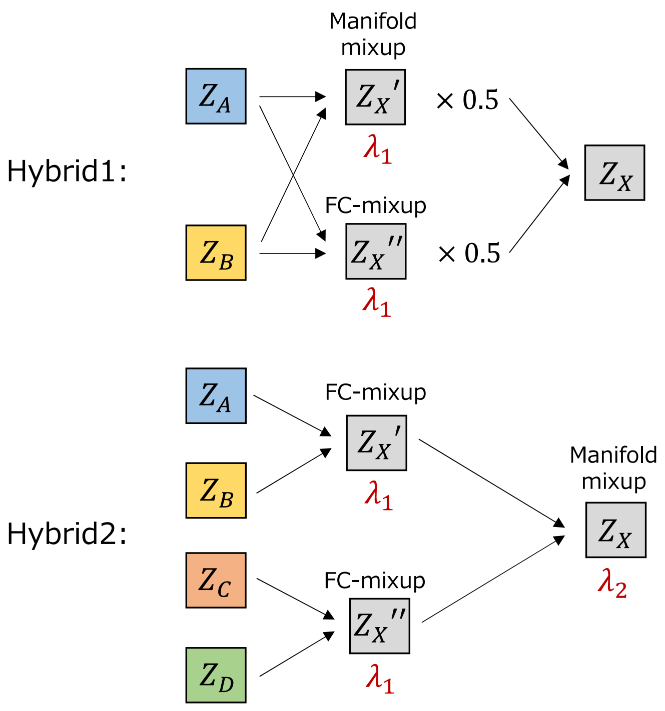


Figure 3.7 Proposal of two Hybrid methods

Table 3.2 Comparison of test accuracy between multiple FC-mixup methods and existing methods

	CIFAR-10 ResNet18	SVHN ResNet18	CIFAR-100 ResNet18
Conventional method	95.20 ($\pm$0.32)	97.10 ($\pm$0.08)	78.58 ($\pm$0.39)
FC-mixup (channel-wise)	**95.23 ($\pm$0.16)**	97.12 ($\pm$0.06)	78.54 ($\pm$0.20)
FC-mixup (pixel-wise)	95.08 ($\pm$0.16)	96.95 ($\pm$0.10)	**78.74 ($\pm$0.11)**
Hybrid 1 (channel-wise)	95.06 ($\pm$0.18)	96.93 ($\pm$0.07)	78.68 ($\pm$0.37)
Hybrid 1 (pixel-wise)	94.88 ($\pm$0.20)	96.97 ($\pm$0.09)	78.72 ($\pm$0.25)
Hybrid 2 (channel-wise)	95.00 ($\pm$0.33)	**97.15 ($\pm$0.11)**	78.72 ($\pm$0.23)
Hybrid 2 (pixel-wise)	95.00 ($\pm$0.24)	97.01 ($\pm$0.10)	**78.74 ($\pm$0.10)**
	CIFAR-10 (1000) small CNN	SVHN (1000) small CNN	MNIST (1000) small CNN
Conventional method	61.09 ($\pm$0.15)	74.44 ($\pm$0.57)	96.54 ($\pm$0.10)
FC-mixup (channel-wise)	**62.60 ($\pm$0.91)**	78.05 ($\pm$0.40)	96.74 ($\pm$0.07)
FC-mixup (pixel-wise)	59.61 ($\pm$0.53)	75.41 ($\pm$0.57)	96.83 ($\pm$0.23)
Hybrid 1 (channel-wise)	61.08 ($\pm$0.77)	76.09 ($\pm$0.64)	96.52 ($\pm$0.15)
Hybrid 1 (pixel-wise)	59.89 ($\pm$0.70)	74.63 ($\pm$0.62)	96.61 ($\pm$0.20)
Hybrid 2 (channel-wise)	61.54 ($\pm$0.92)	**78.37 ($\pm$0.74)**	96.79 ($\pm$0.12)
Hybrid 2 (pixel-wise)	59.60 ($\pm$0.64)	75.81 ($\pm$0.53)	**96.94 ($\pm$0.21)**

3.4 Efficient data augmentation policy search using Affinity and Diversity

3.4.1 Affinity, Diversity

There are numerous methods for data augmentation, and each method has its own hyperparameters. Therefore, determining an appropriate augmentation policy requires a lot of training, which is computationally expensive. In this study, we have developed a new method to reduce the computational cost of augmentation policy search.

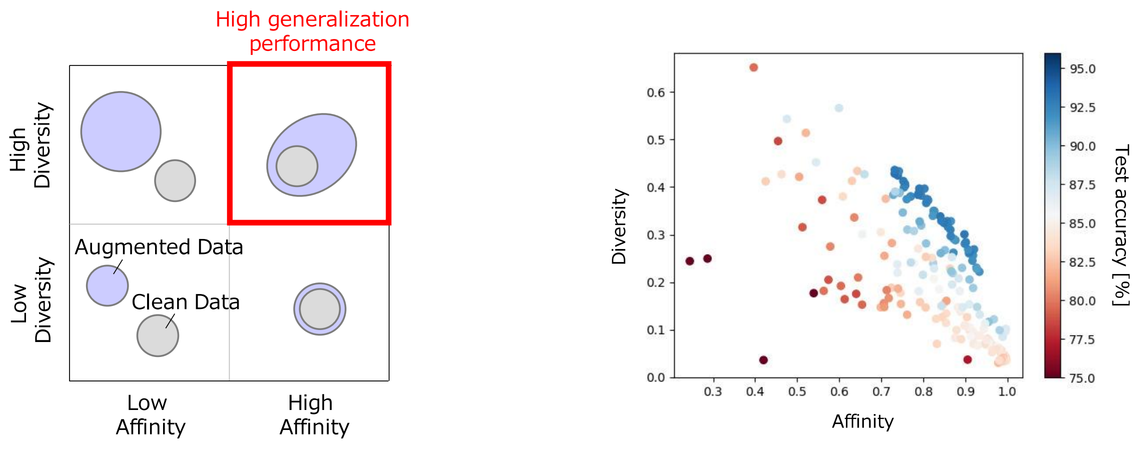
In general, the search for appropriate augmentation policies is performed using validation accuracy, but in this study, we focused on Affinity and Diversity, indicators proposed for data augmentation [13]. As shown in Figure 3.8, Affinity represents the degree of overlap between the original and extended data distributions and is calculated by the following equation:

$$Aff \coloneqq A(m, D'_{val}) \;/\; A(m, D_{val})$$

where $A(m, D'_{val})$ represents the accuracy of validation data with data augmentation in a model trained with clean data, and $A(m, D_{val})$ represents the accuracy of clean validation data in the same model. Diversity represents the spread of the data distribution after the augmentation and is calculated by the following equation:

$$Div \coloneqq E_{D'_{train}}[L_{train}] \;/\; E_{D_{train}}[L_{train}]$$

where $E_{D'_{train}}[L_{train}]$ represents the value of the error function of the model trained using training data with data augmentation, and $E_{D_{train}}[L_{train}]$ represents the value of the error function of the model trained using clean training data.



(a) Difference in distribution according to the size of the metric

(b) Relationship to test accuracy

Figure 3.8 Characteristics of affinity and diversity

It is known that test accuracy is higher when training with a data augmentation policy that increases both Affinity and Diversity values, as shown in the distribution framed in red in Figure 3.8(a). This can be confirmed in the results shown in Figure 3.8(b). In this figure, each point shows the results of trainings using data augmentation with different methods and hyperparameters. The higher accuracy in the upper right portion of the figure can be seen, indicating that both affinity and diversity are important.

3.4.2 Shortening the search phase with a metric considering Affinity and Diversity

To reduce the computational cost of searching for data augmentation policies, this study proposes shortening the search phase, i.e., reducing the number of training steps for search. This can easily reduce the computational cost, but when using validation accuracy, the problem arises that the test accuracy cannot be well estimated with a short number of training steps. Therefore, we propose a metric $aff \times div^{\alpha} \times val$ that takes into account affinity and diversity. This is a metric that becomes large when affinity, diversity, and validation accuracy all take large values. Since diversity is an unstable metric in the early stages of training, as described below, α, which takes values between 0 and 1, is used to reduce the influence of diversity.

The training method using this proposed metric is shown in Figure 3.9. With the conventional method, the search phase is carried out to the final step, and the test accuracy is estimated using the validation accuracy. On the other hand, in this method, after a short search phase, the proposed metric is evaluated by calculating affinity, diversity, and validation accuracy, and the data augmentation policy with the largest value is selected. That augmentation policy is used for training until the final step. This approach reduces the overall computational cost by shortening the search phase, in which multiple data augmentations are used to learn individually. For example, if the overall learning is 200 epochs and the search phase in this method is 5 epochs, roughly speaking, the computational cost could be reduced by a factor of 0.025.

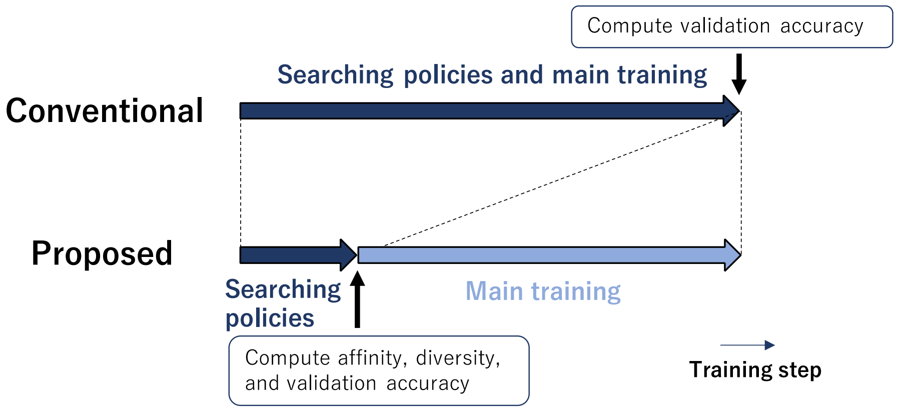

Compute validation accuracy
Searching policies and main training
Conventional
Proposed
Searching policies
Main training
Compute affinity, diversity, and validation accuracy
Training step

Figure 3.9 Summary of the proposed method

Experiments were conducted to test the effectiveness of the proposed method using multiple datasets. The model used was ResNet50 for ImageNet and ResNet18 for other datasets. Nine data augmentation methods, PatchGaussian [21], FlipLR, FlipUD, Crop, Cutout, Crop+FlipLR+

Cutout, Rotate, ShearX, and ShearY, each with multiple hyperparameters, were used, with a total number of 100 data augmentations for ImageNet and 216 for the other datasets. After a search phase of 5 epochs, a data augmentation policy is selected using different metrics. The data augmentation was applied to the training until the final epoch (100 epochs for ImageNet and 200 epochs for the other datasets) and the test accuracies were compared. The experimental results are shown in Table 3.3. The results show that the proposed metric (Proposed) is able to select an augmentation policy that yields higher test accuracy compared to the case where only validation accuracy is used (Val acc). Ground truth is the most accurate result among all the data augmentations trained until the last epoch. By taking affinity and diversity into account, the proposed method was able to estimate the test accuracy with good accuracy even in a short search phase.

Table 3.3 Comparison of test accuracy of trainings with selected augmentation policies for each metric

Dataset	Metric	Selected data augmentation	Selected hyperparameter	Test acc [%]
CIFAR-10	Val acc	ShearX	6, 1.0, -, variable	85.7
	Affinity	FlipLR	0.25, -, -, fixed	86.5
	Diversity	PatchGaussian	32, 2.0, -, fixed	80.4
	Proposed	Crop + FlipLR + Cutout	0.5, 0.25, 0.25, fixed	92.4
	Ground truth	Crop + FlipLR + Cutout	1.0, 0.5, 1.0, fixed	93.7
CIFAR-100	Val acc	PatchGaussian	4, 1.5, -, fixed	59.6
	Affinity	FlipLR	0.25, -, -, fixed	65.3
	Diversity	Rotate	45, 0.75, -, fixed	59.8
	Proposed	Crop + FlipLR + Cutout	0.25, 0.5, 0.25, fixed	70.9
	Ground truth	Crop + FlipLR + Cutout	1.0, 0.75, 0.25, fixed	72.5
SVHN	Val acc	PatchGaussian	16, 2.0, -, fixed	95.7
	Affinity	ShearY	6, 0.25, -, variable	95.8
	Diversity	PatchGaussian	28, 0.3, -, fixed	94.6
	Proposed	Rotate	20, 0.75, -, variable	96.2
	Ground truth	Crop	1.0, -, -, fixed	96.8
ImageNet	Val acc	ShearY	4, 1.0, -, fixed	70.8
	Affinity	ShearY	5, 1.0, -, fixed	70.9
	Diversity	Crop + FlipLR + Cutout	0.25, 0.5, 0.5, fixed	71.2
	Proposed	Crop + FlipLR + Cutout	0.5, 0.5, 0.25, fixed	71.8
	Ground truth	Rotate	60, 0.5, -, fixed	72.2

The correlations between epoch 5 and epoch 200 for each metric using CIFAR-10 are shown in Figure 3.10. Each point represents the result of training with different data augmentations. These results show that Affinity is highly positively correlated. This is thought to be because the calculation of affinity requires a model trained with clean data, so that in the early stages of training, the model is not unstable, with large changes in accuracy due to data augmentation. Conversely, the calculation of diversity requires learning with data augmentation, and the values of the error function in the early stages of training do not necessarily reflect the final test accuracy values well, resulting in smaller correlations. This is also the reason why the influence of diversity is reduced in the proposed metric.

In summary, this study addressed the problem of high computational cost in data augmentation policy search. By significantly shortening the training step in the search phase and using an metric that takes into account affinity and diversity in addition to validation accuracy, we were able to reduce the overall computational cost while performing augmentation policy search with high accuracy.

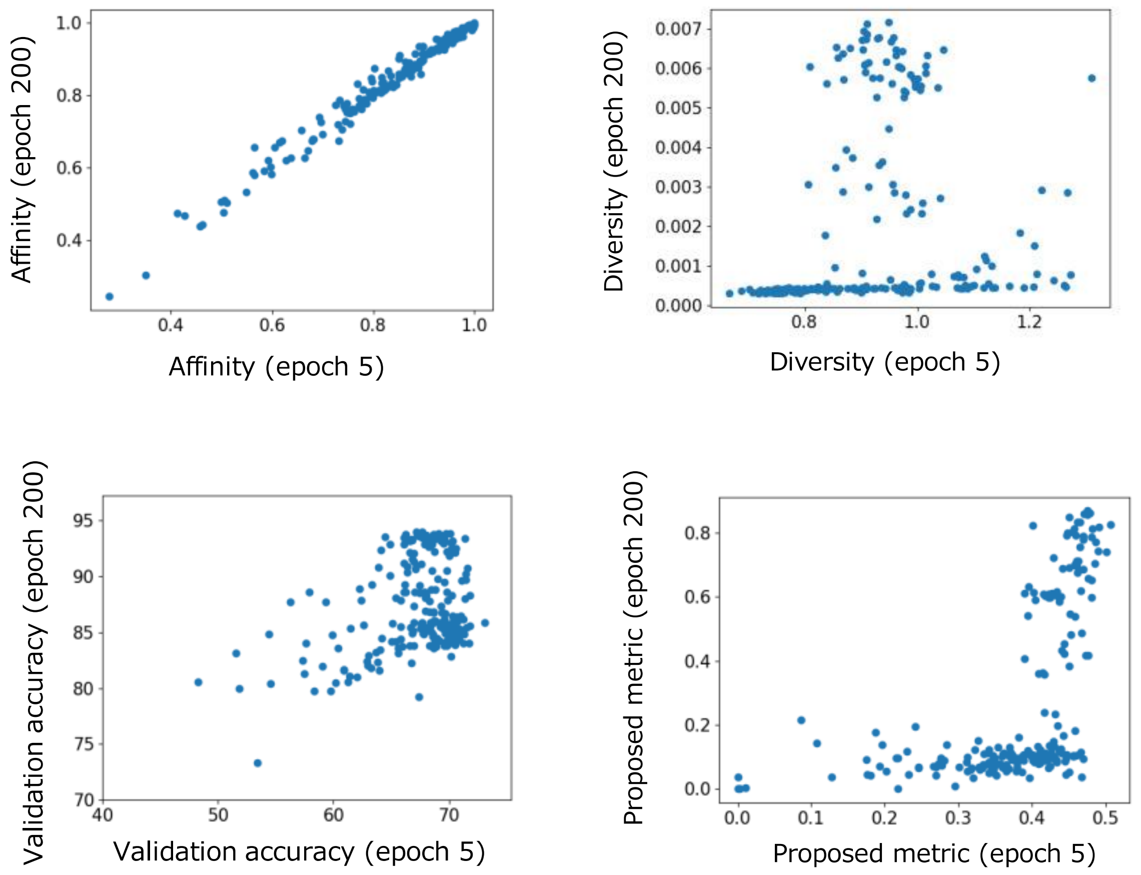


Figure 3.10 Correlations between epoch 5 and epoch 200 for each metric in the study using CIFAR-10

4 Debug-Testing of DNN Software

In the initial development stage of Deep Neural Network software (DNN software), we ensure that the required functions and prediction performance are achieved through iterative trial-and-error processes, in which three viewpoints (elaborating and refining requirements, preparing datasets for training, and selecting appropriate machine learning models) are considered. This trial-and-error process corresponds to debugging in conventional program development. In the case of DNN software, the debugging activities involve generating datasets for debug-testing, monitoring the training and learning status, and identifying and removing root causes that hinder the fulfilment of requirements. In the following, we will report on a debug-testing method investigated in FY2020, discuss the experimental results obtained, and summarize our future plans.

4.1 Direct cause of failure

A standard method of supervised DNN learning involves two types of programs: training (or learning), and prediction (or inference). When training data is given and a learning task to achieve is made clear, a learning model for the target DNN software is selected, and some design decisions on the method used in the training and learning process is fixed. If we use available open-source machine learning frameworks, we may set up several parameters of the framework. The next step is to construct training dataset. Then, we run the training/learning program (possibly provided by the machine learning framework) with the training model and training dataset as input, and derive a trained DNN model as a computation result. More precisely, the training/learning program searches for a set of weight parameter values that define the trained DNN model uniquely. This trained DNN model defines behavior of the prediction/inference program.

From a user's point of view, a prediction/inference program is the entity to use. In the case of a classification learning task, for example, the program calculates certainty levels of probabilities of classification results for an input data. By examining the output results, we can determine whether the DNN software works as intended. When the program does not produce results as expected, we localize possible fault locations and remove them. In other words, we conduct debugging.

A failure may be occurred due to a flaw somewhere in the information used in the execution process of the training/learning program, either in the training dataset, the training model, the learning mechanism, or their combinations. However, direct causes of failure in prediction/ inference results are attributed to the trained DNN model or set of obtained weight parameter values. While a root cause of failure is somewhere and often not known, the failure is attributed to a defect in the weight parameter values or the trained DNN model. Thus, from users' point of view, a certain distortion of the trained DNN model seems a direct cause of the failure [22]. A method to measure such distortion degrees is needed regardless of the root causes.

In this chapter, we investigate whether we can detect faults in DNN software with an internal metric to measure such distortion degrees of trained DNN models. The weight parameter values in the DNN models are the output of the training/learning program, but there is no direct way to check its validity, because those expected weight parameter values cannot be known in advance. If such expected parameter values were known, training/learning could be skipped. We can just use those known values, as embedded in a trained DNN model, to implement a prediction/ inference program.

4.2 Internal indices

This section first introduces the notion of neuron coverage (NC). We consider a learning model as a network of neurons. Given a threshold, neurons whose output values exceed the threshold are said to be activated. When the number of neurons constituting the learning model is N and the number of activated neurons is A, the neuron coverage is defined as the ratio of active neurons is (NC = A/N). In [23], NC is assumed to be criteria for test coverages of trained DNN models; the research work investigates how the choice of input data for evaluation affects NC values.

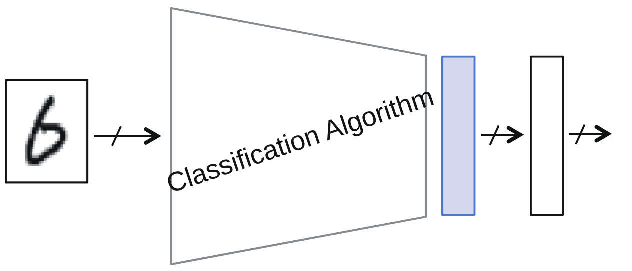


Figure 4.1 Trained DNN model.

In this chapter, NC is assumed to be used as an internal index [24] to represent distortion degrees by appropriately choosing the target neurons to be considered. Figure 4.1 shows a schematic diagram of the trained DNN model. NCs are defined for the neurons in the final stage of the middle layer (or the penultimate layer as shaded gray), but not for all the neurons in the trained DNN model as in [23].

In general, in machine learning techniques, this penultimate layer is often considered to hold meaningful information. For example, in the case of an image classification task, the early stages of the model is responsible for the correlation analysis (analysis of patterns of pixel values), which plays a specific role in algorithms such as image recognition, and their calculation results are summarized in the penultimate layer. In this chapter, we assume that direct causes of defects are manifested in this internal layer. Furthermore, various statistical indices can be derived based on NC values of this layer. We will investigate, through experiments, what derived index is appropriate depending on test objectives to be investigated.

4.3 Experiments: method and results

We present the results of several experiments and discuss the usefulness of the internal or derived indices mentioned in the previous section. First, we show the results of comparative experiments when a training/learning program (or a learning framework) has faults in it. In the following, BI is the training/learning program which is a bug-injected version of a probably correct program PC.

Figure 4.2 depicts the accuracy (the percentage of reconstructed correct answers) for a test dataset. In the experiments, a classical fully-connected network is chosen as the learning model, and different number of neurons are placed in the middle layer, which implies that each model is of different structural capacity. When we have a sufficient number of neurons (50 on the horizontal axis), there is no significant difference in the accuracy between PC and BI. Thus, it is difficult to distinguish between the PC and BI solely by examining their accuracy values, and thus the presence or absence of a defect cannot be identified. In addition to this finding (Figure 4.2), the results of an experiment to systematically investigate the situation further (Figure 4.3) are presented below.

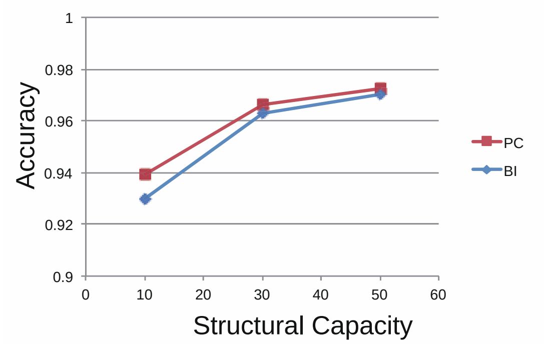


Figure 4.2 Learning models of different capacities.

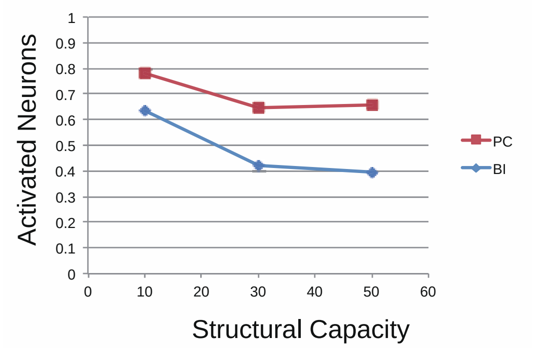


Figure 4.3 Relationship with internal indices

Figure 4.3 plots values of the internal index (activated neurons or neuron coverage) on the vertical axis. Their absolute values, for example, of 10 for BI and 30 for PC are both around 0.7, making it impossible to distinguish between BI and PC if we do not take into account the structural capacity. The indices are not usable to examine the activated states of neurons.

Therefore, we will study if there is an appropriate indicator to be derived from the internal index of NC. As a set of data (a sample), in the test dataset, leads to a collection of neuron coverages, we can obtain some statistics from the sample such as the mean μ and variance σ^2, and calculate σ / μ. Figure 4.4 shows the case where this derived index σ / μ is used on the horizontal axis. From the values on the vertical axis, we can find out which leaning model has which value by referring to Figure 4.3.

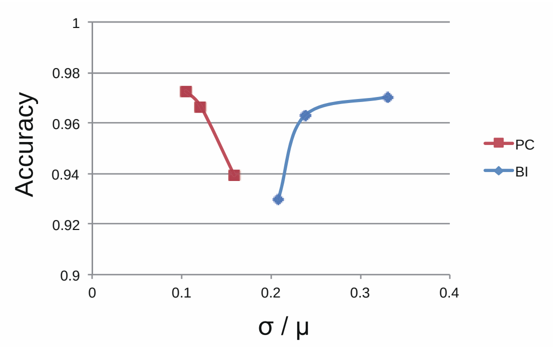


Figure 4.4 Derived index

Figure 4.4 shows that we can distinguish between the PC and BI. Although the internal index cannot distinguish between the PC and BI with different capacities (Figure 4.3), a derived index of σ / μ can discriminate between the PC and BI. We can see that the neuron coverage basically contains a piece of useful information.

Next, Figure 4.5 is a scatter plot of classification probability using corrupted data for the evaluation; the horizontal axis refers to the classification output by the BI and the vertical one by the PC.

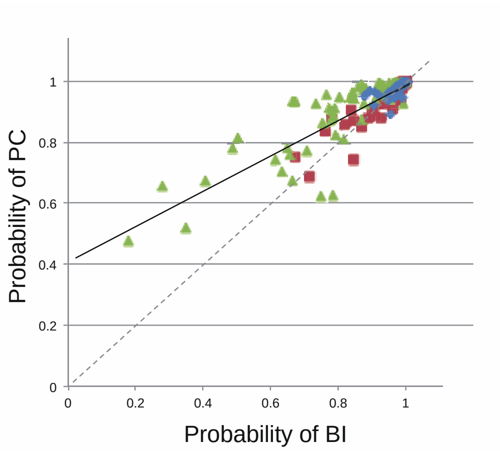


Figure 4.5 Classification certainty for corrupted data.

In Figure 4.5, a △ represents an output value for corrupted data, which is supposed to be distributed on the dotted line passing through the origin, if we assume that the PC and BI output the same value for the same data. In fact, it can be seen that □ selected from the test dataset (without any corruption) mostly arranged on the dotted line. On the other hand, corrupted data (△) are distributed along the solid line, indicating that the PC is a better classification certainty than the BI. It implies that the BI, containing bugs in it, is less robust, although the accuracy remains the same as that of PC (Figure 4.2).

The following experiment confirms that differences in robustness can be detected by using an internal index (Figure 4.6).

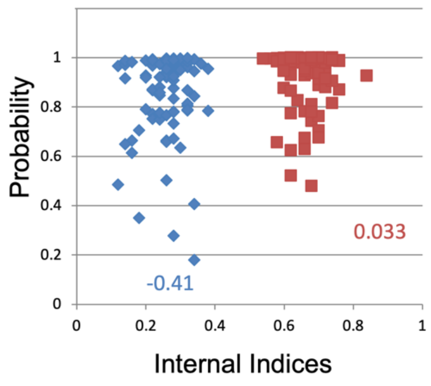


Figure 4.6 Differences in internal indices

The corrupted data described above were input, and the internal index for each input was plotted on the horizontal axis. The □ distributed in a group on the right side shows the results of PC, and the ◇ distributed in a group on the left side shows the results of BI. The scatter plot shows that (1) the value of the internal index of PC is large, and (2) the correlation between the internal index and prediction probability (certainty of classification) is negligible (0.033). Next, we calculate σ / μ, which is 0.0876 for PC and 0.2183 for BI. Figure 4.6 shows results that corrupted data affect the robustness, and that the value of σ / μ is considered to have correlations with the robustness.

Next, we conducted experiments to investigate how distorted training data affect the trained DNN model. We plotted the accuracy for a test dataset common to all the cases. Thus, differences in the vertical axis indicate a certain difference (distortion degree) in the training dataset used for obtaining the trained DNN model (Figure 4.7).

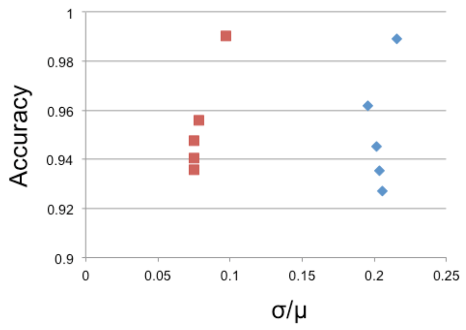


Figure 4.7 Differences in training datasets.

Figure 4.7 shows the two independent series for the PC (□) and BI (◇). From top to bottom in a series of each measured points (from better to worse accuracy), a training dataset with a larger distortion is used. Because the test dataset is common, the data shift of the test data is relatively larger as the distortion degrees in the training data is larger. Furthermore, the accuracy decreases as the shift becomes large. Figure 4.7 also shows that the value of the horizontal axis (σ / μ) is clearly different between the PC (□) and BI (◇). It can be confirmed that the accuracy and the robustness suggested by the σ / μ values are two independent perspectives.

From the above (Figure 4.7), the distortion in training dataset can be examined by the method based on the accuracy. As is done in practice, the method based on the accuracy is useful when checking the training dataset quality. On the other hand, if there is a possibility that other factors such as faults in a training/learning program are involved (multiple defects are assumed), it is desirable to examine the values of the internal and derived indices (σ / μ) at the same time.

4.4 Related work

Neuron coverage (NC) is a simple quantitative measure introduced in DeepXplore [23] as a test coverage metric. In conventional software testing, test coverage is defined in terms of the basic block of program codes, which is the statements executed by a given test input data. A program is represented as a Control Flow Graph (CFG) whose nodes refer to executable statements. In the simplest case, the criterion is whether or not a node in the CFG is contained in an execution path induced by an input test, i.e., whether or not the statements are executed. As a DNN model is represented as a network, a kind of graphs, metrics similar to those for CFG can be introduced. The neuron coverage concerns whether neurons located at nodes are activated (output values of these neurons exceed a specified threshold), which is comparable to the C0 criterion defined on the CFG. DeepXplore assumes that high NC values refer to the situations where high percentage of neurons are exercised by input data, and discusses how to generate new test input data to increase the NC values.

Neuron coverage would be a straightforward idea analogous to the conventional test coverage criteria. Later, satisfying the criteria, to achieve 100% in terms of NC, is found empirically not difficult. New metrics are proposed to take into account correlations among multiple neurons or those in different layers [25], which may be comparable to more elaborated coverage metrics, such as C1 or the others, in conventional software testing.

The original NC is simple and easy to use as a metric to guide or control automated test generation processes. Usually, a classical data augmentation method picks up a seed data, from which a series of new data is to be generated by pre-defined data transformation algorithms. New test data are successively generated until the accumulated NC values is saturated. When reached the situation where no increase in the NC is seen, the generation method switches a seed data to new one and continue the process [26]. The classical data augmentation method can be replaced by other approaches such as test input generation based on GAN [27]. Test generation method using GAN with a help of NC is reported in [28]. Although it is a simple metric, NC is now considered as a practical criterion to control the automated test generation process (coverage-guided test generations).

Some of early works on testing pre-trained DNN models adapt application-specific properties as software test oracles; the DNN models for regression tasks in the auto-pilot car application [26][27] use the calculated steering angle as the oracle. There is also a research work [29] to investigate whether test inputs to increase the NC values are useful for detecting faults. The usefulness of NC is dependent on what are considered failures. The work [29] also indicates that the correlation between NC and external indices such as the accuracy is weak. In this chapter, based on this observation that the correlation between the two is weak, an internal index based on the NC is used for the test, which is not contradictory to the discussion in [29], but rather in the same direction. Note that the test coverage is a criterion for terminating testing, while detecting faults depends on whether the test input data executes corner cases. These two notions, the test coverage and corner cases, refer to different aspects. In fact, it has been reported that the enhancement of coverage does not necessarily leads to the improvement of the efficiency of fault detection in conventional software testing. The same findings would be applicable to cases of DNN testing.

In this chapter, we use the NC value as a simple test index, from which a sort of distortion degrees in trained DNN model is derived [22][24]. Our approach is based on a view that faults in DNN models appear as inappropriate NC values, whereas existing works use NC as a criterion for the test coverage. In our experiments, we were able to examine the reliability of the training and learning programs and the robustness of the trained DNN models. These are two primary concerns in debug-testing.

4.5 Conclusion

In this chapter, we used an internal index based on the neuron coverage (NC) defined on the penultimate layer for representing a sort of distortion degrees in trained DNN model. The NC is a scalar and easy to measure, and thus can be used as a light-weight test index. It, however, discards the information about the individual activated neurons, and thus lacks useful information. In fact, Kim et al. [30] proposes a method to estimate the distribution of activated neuron and to discuss the usefulness of input data for testing. Distribution on such neuron values may be considered to have rich information. In future, we will study how to debug training dataset by making use of such distribution information.

5 Debugging and Testing of Training Data

5.1 Three Problem Settings

In early stages of software development, in which programs are constructed to employ Deep Neural Networks (DNNs) [31], debugging and testing is performed to ensure that the core DNN components behave as expected. This is the process of feeding appropriate data to the DNN components and checking whether the predicted output is exactly what is expected. If the output is faulty in some ways, the DNN component under test contains a defect. The purpose of debugging is to identify and remove such unknown defects.

Defects in DNN components are the direct cause, but not the root cause, of failures. In the standard method for building DNN components [32], three distinctive components are basically involved: (a) the machine learning infrastructure, (b) the training model (a template of the DNN model), and (c) the training data. The root cause is one of them or their certain combinations leading to the failure that the DNN component exhibits. The problem setting of the inspection differs depending on where the root cause is assumed [33].

The basis of DNN component construction is to make use of a training dataset consisting of a huge number of training data and derive the information inherent in those data by means of statistical methods so as to obtain a DNN model (a nonlinear function) inductively. In a naive way, we may examine the DNN model to identify root causes. However, since the DNN model is a nonlinear function to exhibit some functional behavior, the software testing method using indirect test oracles is often employed; we feed evaluation data to the DNN model and check whether output results are valid or not [34].

In the case (a) above, the core of the machine learning infrastructure is a numerical program that solves an optimization problem, and the metamorphic testing method is known to be useful [35]. In the case (b), the learning model is not obviously flawed. It is to find an optimal or sub-optimal learning model for the target machine learning task, which has been, in a sense, one of the main challenges of the DNN technology [31]. In this chapter, we discuss the case (c), i.e., debugging and testing methods of training data.

5.2 Debugging Problems of Training Data

Debugging and testing of training data is to revise (add or delete) the training data so as to obtain a DNN model that exhibits the intended functional behavior. This view is based on the observation that the bias of the training data affects much the trained DNN model. In the following, we specifically consider the debugging problem of training data for supervised machine learning classification tasks.

5.2.1 Model Accuracy and Model Robustness

In the supervised task of classifying input data into C categories, a datapoint z is a tuple ($z = \langle x, y \rangle$) consisting of two types information, a multidimensional vector x and its correct answer tag (or simply a label) y (see Figure 5.1). The DNN model, derived from a given training dataset S ($S = \{z^{(k)} \mid k = 1, \dots, N\}$), is inspected against input evaluation data x. Its output is a C dimensional classification probability vector P_x corresponding to the data x. If $P_x[j]$ (the j-th component of P_x), the component with the largest value j, is equal to y ($y = \mathrm{argmax}_{(j \in [1,C])} P_x[j]$), then the DNN model is considered to return a correct answer. In this case, the multidimensional vector P_x, in particular, the probability of the j-th component $P_x[j]$, is one of the good indicators of the model accuracy for the data x. For a collection of evaluation data E ($E = \{\langle x^{(\ell)}, y^{(\ell)} \rangle \mid \ell = 1, \dots, M\}$), Accuracy is the number of correct answers (percentage of correct answers) for the collection. In addition, the variability of the probabilities of the classification categories (sometimes referred to as Gini Impurity) is an indicator of the model accuracy as well.

The accuracy for the training dataset S and the one for the other dataset E, different dataset from S, are compared. While the accuracy for S is good, the accuracy is sometimes worse for E. This phenomenon is known as overfitting to the training dataset. Usually, both S and E are constructed from one large data pool D, and are considered as different samples following the same data distribution; E in this case is sometimes called a testing dataset as compared with the training dataset of S. When there is no overfitting where the accuracies are not much different each other, the DNN model is considered to exhibit good generalization performance.

In the training data debugging problem, the evaluation data E may be selected from a dataset other than D. For example, in positive testing, where the goal is to confirm that the system exhibits the expected behavior, as in the evaluation of generalization performance, we can choose E from D, in which E is different from S. However, to test the behavior in exceptional situations, we may choose a dataset F for the evaluation that is not included in D. Model accuracy, measured with the percentage of correct answers, is not a good indicator for F. The evaluation criterion is model robustness, which expresses how the prediction probability is decreased depending on how much a data in F is deviate from data in D or S.

In the development in practice, if the expected prediction performance is not achieved for a given D, new data is collected and the training data itself is revised. Then, the DNN components are derived using the new training dataset, namely in an iterative manner. Moreover, during testing, we evaluate the model accuracy and model robustness in view of both positive and exceptional testing.

5.2.2 Memorization of Training Data

Overfitting or overlearning significantly affects the prediction performance (the model accuracy and model robustness) of DNN models. Therefore, basic machine learning methods have been studied extensively to mitigate those problems; the study includes regularization or dropout [36]. In spite that such methods are adopted, the expected prediction performance cannot be obtained if the training dataset is inadequately biased. The debugging problem of training data is to improve the prediction performance of DNN models by revising the training dataset. Simply, it is to eliminate the inappropriate bias. However, it is difficult to evaluate the degree of bias as well as the appropriateness or inappropriateness of the bias.

One traditional approach to evaluate the bias of the training data (sample) is to examine statistical characteristics of the sample. For example, given that $S = \left\{\langle x^{(k)}, y^{(k)}\rangle \,\middle|\, k = 1, \dots, N\right\}$, let $S^C = \{\langle x, c\rangle \mid \langle x, c\rangle \in S \text{ and } c = 1, \dots, C\}$ where c is a correct answer tag. If the sizes of S^C are equal in size, then we may say that there is no bias among S^C from the viewpoint of the correct answer tag. However, each S^C follows some data distribution ρ^C and we don't know whether S^C is sampled faithfully in regard to ρ^C. To check this, we need to know ρ^C, however, the data x is multidimensional, and such a multidimensional data distribution is not easy to estimate.

Alternatively, the prediction performance of DNN models is investigated by testing results with input evaluation data. DNN models derived from the same training data may exhibit different prediction performance, depending on the method of the machine learning. In other words, it is not enough to examine the statistical characteristics of the training data for the purpose of debugging the training data, but it is also necessary to consider how the bias of the training data is reflected in the trained DNN model.

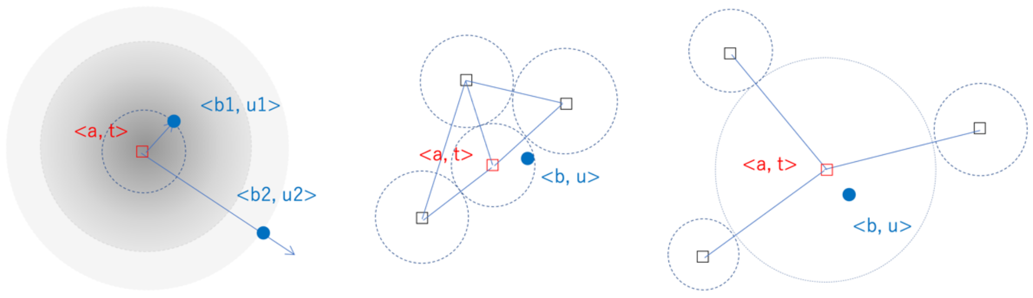
The relationship between DNN models and training data bias can be discussed in terms of the DNN models remembering the labels of the training data. Now, when the training data S contains a datapoint $\langle a, t\rangle$ ($\langle a, t\rangle \in S$), we can construct S' so that the $\langle a, t\rangle$ is removed from the training data S ($S' = S \setminus \{\langle a, t\rangle\}$). Let each DNN model obtained by training with either S or S' be M or M' respectively. Then, the result, P_a for M or P'_a for M', is obtained for the common input data a. If the classification result t for $P_a[t]$ is very likely and $P'_a[t]$ is less likely, then M is said to *memorize* the datapoint $\langle a, t\rangle$ used as one the training data. From this definition, we can see that the DNN model memorizes the training data in the overfitting situation, where $P_a[t]$ is apparently more likely than ${P'}_a[t]$.

For DNN models, it is known that the Membership Inference is possible. The problem is to find out if a datapoint $\langle x, y\rangle$ ($\langle x, y\rangle \in D$) was included in the training dataset ($\langle x, y\rangle \in S$) just from the information obtained by feeding data to the trained model, $M^S(x)$. Black box methods make use of the classification probability vector P_x [37], or white-box methods use the information of the loss function $\ell(Y(W; x), y)$ calculated in the process of executing $M^S(x)$ [38], where W is the training parameter or weight and $Y(W; x)$ is the internal representation of the prediction for the input x.

Intuitively, Membership Inference method is based on the observation that the distribution of P_x or $\ell(Y(W; x), y)$ is different depending on whether the datapoint $\langle x, y\rangle$ is included in the training dataset S or not. Furthermore, these differences in the distributions are somehow attributed to the memorization of training data including overfitting cases [38]. Thus, the approach to mitigate the threats of Membership Inference is to remove those data, that are memorized easily, from the training dataset, in addition to employing a machine learning method that avoids overfitting [39].

We now examine the situation involved with the memorization of training data. Consider a classification problem as in Figure 5.1; we assume $a \neq b$ whereas $t = u$. Figure 5.1 (a) illustrates a situation where the prediction probability of $\langle b, u \rangle$, a training data moved away from $\langle a, t \rangle$, decreases as the distance between them becomes large. Figure 5.1 (b) shows that removing that datapoint $\langle a, t \rangle$ from S does not significantly affect the prediction probability of the data $\langle b, u \rangle$ when the training data are dense in S. In other words, the removed training data is not memorized in that it does not significantly affect the prediction results. Figure 5.1 (c) represents a situation where the training data are sparse. Contrary to Figure 5.1 (b), it represents that the influence becomes large and is firmly remembered. Such outlier data significantly affects the predictive classification performance of the DNN model.

Finally, we consider the Membership Inference viewed from the training data debugging problem. In the situation where training data are memorized, the distribution of either P_x or $\ell(Y(W; x), y)$ is very different depending on whether the datapoint is included in the training dataset S or not. The Membership Inference method makes use of the fact that the predictive performance for z', far from training z datapoints, is poor. In other words, we can think of the Membership Inference as a test of model robustness; the phenomenon of training data memorization is related to model robustness.



(a) Predicted probability in the neighborhood (b) Dense region (c) Sparse region
Figure 5.1 Training data placement and prediction certainty.

Here, we refer to the schematic situation in Figure 5.1. Removing the dense data shown in Figure 5.1 (b) would have little impact on the model accuracy. On the other hand, removing the data in a sparse region as shown in Figure 5.1 (c) would improve model robustness, but would reduce model accuracy in the neighborhood because there would no longer be data to support their predictive classification results. Alternatively, adding new data in the neighborhood without removing this datapoint will make the region dense and improve the local model accuracy. Therefore, detecting outliers in the training data set S is important for debugging dataset.

Figure 5.1 schematically illustrates that the predictive classification performance of the input data is affected by the location relationship with the training data. However, it does not say how the location relationship is defined, i.e., from what aspects of the data, the location relationship is defined. Conversely, now the question is how the location relation should be defined when discussing the difference in prediction classification performance; the outlier detection problem will become clear when such criteria are precisely defined.

5.3 Outliers and Neuron Coverage

We consider outlier detection methods for the purpose of training data debugging.

5.3.1 Outliers in Training Data

The debugging problem of training data is to find out outliers in the training dataset and to decide how to deal with the outliers according to the purpose of the DNN model under development. How we handle the outliers is related to the requirements specification of the DNN model. Thus, the general discussion of training data debugging may be limited within establishing a technique for outlier detection.

In general, outliers are data that have different characteristics from the data that make up the majority, and whether or not they are outliers is defined based on the data distribution (statistical data model) that the collection of target data exhibits [40]. For example, if the probability density function of the data distribution is known, then we can check whether the data are outlier or not based on the likelihood of the data.

In a naive way, we consider whether it is an outlier or not based on the empirical distribution of the training data. However, the training data is a multi-dimensional vector, and it is difficult to know the empirical distribution in a compact form. For example, it is difficult to apply methods such as Kernel Density Estimation, and as a result, the outlier detection method based on likelihood is not practical. Alternatively, analysis methods similar to Combination Testing, which is known in the field of software testing, may be applied. By selecting components (features) that are considered having a large impact on the empirical distribution and focusing on such representative dimensions, we may conduct analysis as an approximate of the case on the whole empirical distribution. While practically applicable, outliers are rare by definition, and the effectiveness of this approximate method is questionable.

For a slight change of perspective, the robustness radius of the standard method of analyzing model robustness [41] is considered. For two datapoints $\langle x, y \rangle$ and $\langle x', y' \rangle$ and the predictive classification results for each of the outputs $P_x[y]$ and $P_{x'}[y']$, let the robust radius δ be the tolerance level ε of the difference between the outputs; for a given ε, the robust radius is the maximum difference of input data that satisfies δ ($| P_x[y] - P_{x'}[y'] | \leq \varepsilon$ when $| x - x' |_p \leq \delta$). Here, we define the difference of input data in terms of L_p-norm. In a naive way, for a given ε for given input data, we consider that the model robustness is good if the robustness radius δ is large. However, the calculated radius δ_p is dependent on the choice of the norm L_p. While the definition of model robustness by the robust radius is strict, the analysis in the space of input data requires further discussion or interpretation of whether the norm used is appropriate or not, which complicates the problem.

We consider now how the training data $\langle a, t\rangle$ affect the prediction results of the other data $\langle b2, u2\rangle$ that are classified to different classification categories ($t \neq u2$) (see Figure 5.1 (a)). The two datapoints have different classification categories and can be assumed to be far apart in the input data space. We assume that the training dataset S contains $\langle a, t\rangle$ and let S' be the one to be removed $\langle a, t\rangle$ from S. Further, let the DNN models obtained from S and S' be M and M' respectively, and let the predictive classification results for the data $\langle b2, u2\rangle$ be P_{b2} and P'_{b2}. With the method of Influence Functions, which analyzes how S and S' affect the error function, we are able to know that there exists $\langle b2, u2\rangle$ such that the values of $P_{b2}[u2]$ and $P'_{b2}[u2]$ are different [42]. It shows that the presence or absence of the training datapoint $\langle a, t\rangle$ affects the classification probability of $\langle b2, u2\rangle$. Therefore, it is difficult to obtain the desired information by analyzing the differences in the input data space ($a \neq b2$) .

From the above, we can see that it is difficult to systematically detect the desired outliers by analyzing a collection of training data in the input data space. The reason for this is that model accuracy and model robustness are affected not only by the training data but also by various factors involved in the training process, such as the machine learning method. However, we do not claim that the analysis in the input data space is completely ineffective. Such an analysis would give us a vague idea of the empirical distribution of the training data.

In this chapter, we think that even if the features of the input data space are related to model accuracy and model robustness, they are not appropriate as a systematic training data debugging method. We will study systematic methods for detecting outliers in training data.

5.3.2 Active Neurons

Neuron coverage is defined as the ratio of active neurons to the number of target neurons considered [43]. $M^S(x)$ denotes the situation where the input signal (of x) propagates through the DNN model and activates each neuron. When the output of a particular neuron exceeds a given threshold, we call it active, an active neuron.

Neuron coverage was initially proposed as a coverage criterion for coverage-driven test data generation [43]. The active neurons for the input data x provide a useful information in that they influence the output results. On the other hand, the neurons not involved in the predictive inference process, are considered to be inactive. The input data that produce inactive neurons do not effectively test all the neurons, and then new input test data are needed so that they further activate the inactive neurons. When a set of input data makes all the neurons active, the set of test data are considered to reach 100% of the coverage.

After the original proposal in [43], there have been several research works to study the practical usefulness of the neuron coverage as a test coverage criterion [44][45][46]. In particular, it has been recognized that 100% of the neuron coverage is not difficult to achieve and thus is weak as a test coverage criterion, which is similar to the case of the C0 criterion in conventional software testing methods.

On the other hand, we may consider that the training was appropriate in the first place, producing inactive neurons not involved in the predictive inference process. In this case, we can add new input data to the training data and conduct re-training [43]. This suggests the idea of using the neuron coverage as a criterion for evaluating the quality of the model M^S. The following is a discussion from the viewpoint of the neuron coverage as a model quality evaluation criterion [47].

In DNN models M^S for classification tasks, the upstream layers near the input perform encoding E' (Encoding), and is followed by classifying C'. Classifying is done after the encoding ($M^S = C' \circ E'$); $M^S(x) = (C' \circ E')(x) = C'\big(E'(x)\big)$. When the output is a classification probability vector, we place softmax functions in the final layer (logits) of the output; $M^S = \mathrm{SOFTMAX} \circ C \circ E'$. Next, we may place a layer of Fully Connected Network (FCN) between E' and C; $M^S = \mathrm{SOFTMAX} \circ C \circ \mathrm{FCN} \circ E$.

In FCN, a neuron in a layer considered is connected to all the neurons in the next layer, thus the output is swap-invariant, which means that the output is preserved when the neurons are exchanged within the same layer. Therefore, the neuron coverage may be useful to summarize the neuron activity in FCN layers. On the other hand, when the constituent neurons play a specific functional role, such as in SOFTMAX or CNN, it is questionable whether the neuron coverage, which considers all neurons equally, provides useful information. In fact, two different definitions are studied for CNNs, and depending on which one is adopted, the value of neuron coverage is different [48]. In this chapter, we consider neuron coverage for the FCN layer.

A series of experiments are conducted [48] in which training data are systematically generated by means of a classical data augmentation method and the effects on neuron coverage are investigated. The results showed that the difference in training data had influenced the neuron coverages at the E' layer, while only a small effect was made on the C layers. In addition, although the testing data are changed, very small differences are observed on the last layer in C (Penultimate Layer of the whole model). It implies that the differences in the training data are reflected in the FCN layer where $E' = \mathrm{FCN} \circ E$ as introduced early.

In addition, in previous experiments [35][47] in which we have measured the neuron coverage on the FCN located as the last layer of C, we observed little correlation between the classification prediction probability and neuronal coverage. Therefore, the neuron coverage may be considered to represent an aspect independent of the information contributing to the model accuracy. If it is found to be correlated with the model robustness, we can expect that the neuronal coverage on a particular layer is useful as a method to detect outliers for our purposes.

5.4 Experiments and Discussions

We experimentally investigate the relationship between neuron coverage and model robustness, and discuss whether information concerning neuron coverage is useful for debugging training data.

5.4.1 **Steps of Experiments**

The experiment consisted of five steps, including the preparation work: using the MNIST dataset, we assumed a learning model of the form $M^S = \mathrm{SOFTMAX} \circ C \circ \mathrm{FCN} \circ E$. Hereafter, the MNIST training dataset is denoted as LS (60,000 elements) and the test dataset is denoted as TS (10,000 elements).

- Step 1: Measure the active neurons derived by LS. Let M be the trained model obtained by training M^S with LS. Let Act^x be the collection of active neurons for the FCN layer in M generated by the multi-dimensional vector data x ($\langle x, \_ \rangle \in LS$). If the number of neurons constituting the FCN layer is denoted as $|\mathrm{FCN}|$, the neuron coverage is $NC^x = |Act^x|/|\mathrm{FCN}|$. $N^{LS} = \{NC^x \mid \langle x, \_ \rangle \in LS\}$ for the entire element of LS. Then, obtain the data distribution of N^{LS}.

- Step 2: Using the data distribution of N^{LS} as a basis, select training data systematically from LS to obtain LS_j^P. The specific selection method and the meanings of the subscripts P and j are explained in Section 5.4.2.2.

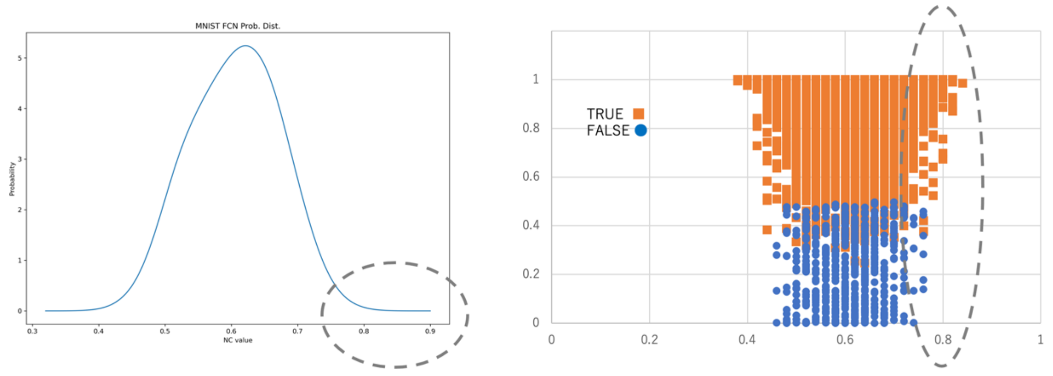
- Step 4: Synthesize data from TS to evaluate model robustness. Let the generating function be T_k and obtain the date ES_k for the evaluation data ($ES_k = T_k(TS)$). The specific synthesis method is described in Section 5.4.3.1. Depending on the data synthesis method T_k, the trained model M is used if necessary.

- Step 5: Evaluate the trained learning model M_j^P with the data ES_k to obtain a model robustness index and investigate the difference in M_j^P, i.e., the impact of LS_j^P on the model robustness.

5.4.2 **Neuron Coverage Distribution**

5.4.2.1 Measurement of Neuron Coverage

A set of neuron coverage N^{LS} was obtained against all the data in the training dataset LS, and the data distribution (i.e. neuron coverage distribution) is shown in Figure 5.2. In Figure 5.2 (a), the horizontal axis represents the neuron coverage for the input data and the vertical axis shows the counts that yielded the corresponding neuron coverage values: the graph is a result of KDE. Figure 5.2 (b) is a scatter plot showing neuron coverage versus input data on the horizontal axis and prediction probability for the same input data on the vertical axis. Red dots represent data for which the prediction is correct (true) and blue dots represent data for which the prediction is incorrect (false). Both the training data accuracy and the test data accuracy were 99%.



(a) Neuron coverage vs data counts (b) Neuron coverage vs prediction probability

Figure 5.2 Results for training data

Figure 5.2 (a) shows that the measured neuron coverage is distributed between 0.38 and 0.84. The median and mean are both 0.60. Figure 5.2 (b) shows that there is little correlation between neuron coverage and predictive certainty (predictive probability value). In Figure 5.2, for example, the area circled by the oval corresponds to training data leading to neuron coverage that are greater than the mean. In particular, not only do the predicted probability values of the correct tag vary widely, even for the same value of neuron coverage, but they also include both correct and incorrect answers. Even if we focus on regions with small values of neuron coverage, the same trend is observed. Therefore, neuron coverage represents the internal state of the predictive inference process, but does not correlate with end-to-end output values.

Small neuron coverage suggests that the input data does not have significant information that contributes to classification. In the extreme case, if the pixel value of the input image is zero, there are no neurons to be activated, resulting in zero neuron coverage. The measurement results were in the small range of about 0.38, suggesting that the "weak" input signal was utilized to lead to the predictive classification results. On the other hand, large neuron coverage indicates that there are many activated neurons. A large neuron coverage value does not necessarily contribute to an improvement in the probability of predicting the correct answer. It suggests that the input data does not have information that leads to a significant classification result, i.e., it cannot distinguish between different classifications. In fact, when the number of neurons in the measured layer is small and the structural capacity is thus small, the correct answer rate is inferior while the neuron coverage is larger: it is consistent with the measured results（Figure 5.2 and Figure 5.3）. Therefore, under the assumption that structural capacity is sufficient, using data for training, where those are located near the center of the data distribution of neuron coverage, can be expected to actively select data that contribute to proper classification and model robustness without affecting model accuracy much.

In this chapter, we summarize the above observations into two hypotheses. Let LS be the training dataset to be debugged, and consider the data distribution of a collection N^{LS} of neuron coverage for all the elements in LS.

［Hypothesis 1］ Neuron coverage is correlated with model robustness

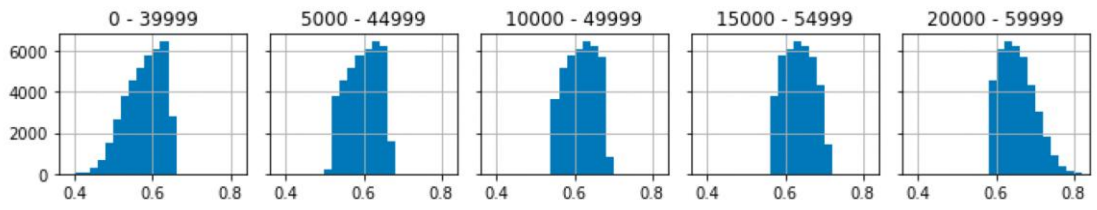
［Hypothesis 2］ The data distribution of N^{LS} provides the criteria for outliers.

Thereafter, experiments will be conducted to confirm the validity of the hypothesis, and guidelines for training data debugging will be outlined.

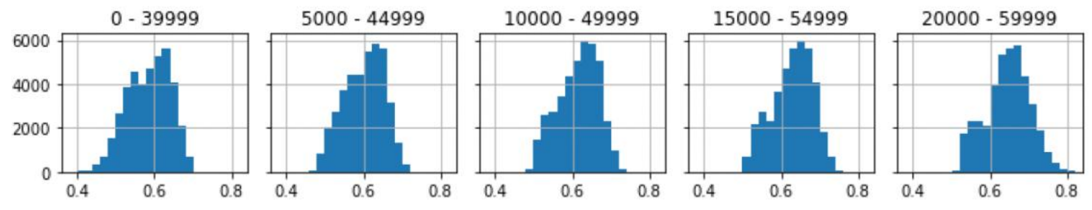
5.4.2.2 Segmentation of Training Data

Based on the data distribution of the neuron coverage N^{LS} of the training data (Figure 5.2 (a)), we systematically select the training data from the LS to obtain LS_j^P. Here, the same number (40,000) of training data LS_j^P are systematically extracted from LS of the size 60,000. Let the MNIST training data LS be divided into $LS^{\langle C\rangle}$ according to the classification class C（$LS = \cup LS^{\langle C\rangle}$）, the following two extraction methods are considered. When the designation of the selection mode P is N, we obtain 5 datasets LS_j^N over the whole LS, where the size interval of the neuron coverage is $[5000 \times j, 5000 \times j + 39999]$ $(j = 0,1,2,3,3,4)$. Alternatively, when the selection mode P is C, for each $LS^{\langle C\rangle}$, 400 elements are selected in the similar way and then combined to obtain LS_j^C. Histograms of each are shown in Figure 5.3.

Comparing LS_j^N in Figure 5.3 (a) and LS_j^C in (b), we can see that the outline of the distribution is different. n addition, the distribution of $LS^{\langle C\rangle}$ originally differed by the classification class C. According to the measurements, for example, 0 and 6 are biased toward regions with small values of neuron coverage (< 0.5), while 3 and 8 are biased toward regions with large values (> 0.5). LS_j^C which extracts and combines the same numbers of data for each $LS^{\langle C\rangle}$, is more gentle than LS_j^N, which shows a shape like a "sheer cliff".



(a) Segmentation of the entire training data（LS_j^N）



(b) Combination of segmentation by the classification class（LS_j^C）

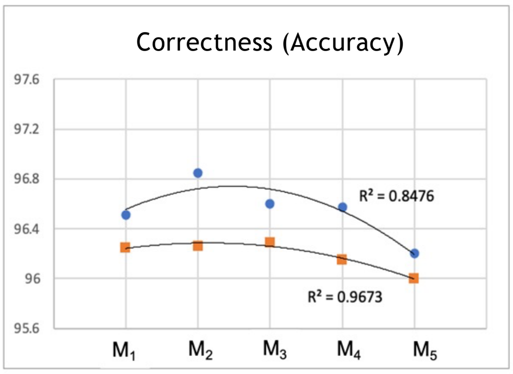
Figure 5.3 Segmentation of the training data

5.4.2.3 Indicator for Model Accuracy

Measure the model accuracy of the trained training model M_j^P obtained using the training data LS_j^P of size 40,000 (10 models in total). The MNIST test dataset TS is used as the data for the evaluation. TS is considered to have the same data distribution as LS, but different from LS_j^P. In fact, when compared in terms of neuron coverage, N^{LS} and N^{TS} showed the identical distribution. Therefore, the neuron coverage distributions for LS_j^P and TS are different. difference is anticipated to affect the quantitative results of model accuracy. The finding that N^{LS} and N^{TS} show the same distribution is consistent with the assumption that there is no sample selection bias between LS and TS.

Here, we investigate how the difference in training data LS_j^P affects the model accuracy in a qualitative way. The percentage of correct answers and Gini coefficients are shown in Figure 5.4 as the indicators of model accuracy. Blue represents the case of extraction from the whole (M_j^N) and red represents the case of extraction by class (M_j^C). The horizontal axis represents the difference in trained training models ($M_K = M_j^P$, $K = 1,2,3,4,5$). The percentage of correct answers is about 96%, not much difference between M_K. The relative differences between M_K are small, about 0.6% in the case of total extraction and 0.3% in the case of class-by-class extraction. The Gini coefficient, which represents the degree of variation among classes, is small (less than 0.002), indicating equal accuracy among classes. From the above, it is confirmed that the difference in LS_j^P has a small correlation with model accuracy.

For the Gini coefficients, the graph trends are different for the overall extraction (blue) and for the class-by-class extraction (red). The former is a gradual monotonic decrease and the latter is a gradual monotonic increase. As can be seen from Figure 5.4, there is a difference in the number of data per classification class between the overall extraction (blue) and the extraction by class (red). Compared to the percentage of correct answers, the Gini coefficient is probably affected more by training data from other classification classes, which may be related to the cause of the difference in monotonicity. though the graph is not shown, there was a strong correlation between the average percentage of correct answers and the average predicted probability.



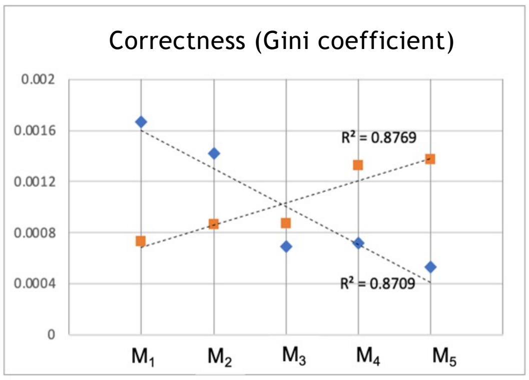


Figure 5.4 Model Accuracy

5.4.3 Measurement of Model Robustness

In empirical evaluation of the model robustness, data for evaluation should be prepared with statistical properties that differ from training or test data.

5.4.3.1 Data Augmentation for Evaluation

We outline the basic approach to discussing model robustness. The robust radius δ [41] is defined as $| x - x' |_p \leq \delta$, satisfying $| P_x[y] - P_{x'}[y'] | \leq \varepsilon$ for a given ε: the larger δ, the better the robustness.

In the measurement experiment, given a reference data point $\langle x, y \rangle$, we introduce a transformation function T that systematically finds x' ($x' = T(x)$, $y' = y$ in the classification task). We define the data distance $d_T(x) = |x - T(x)|$ using the L_2 norm. For a given ε, the robust radius δ can be obtained empirically by varying $d_T(x)$ and observing the difference in the prediction probability. However, as will be explained later, it is difficult to properly determine the transformation function T such that $d_T(x)$ varies smoothly. Here, we consider a transformation function T that produces data showing different qualitative properties. Specifically, Gaussian noise insertion, missing small areas, semantic noise insertion (frame, underline), and affine transformations (rotation, reduction and expansion) were used.

The elements of the MNIST test dataset TS are used as reference data and the transformation function T_k is applied to generate evaluation data ($ES_k = \{\langle T_K(x), y \rangle \mid \langle x, y \rangle \in TS\}$). In addition, for $diff_K(x) = | p_x(y) - p_{T(x)}(y) |$, let $Diff_K = \{ diff_K(x) \mid x \in TS \}$. Furthermore, let $d_K(x) = | x - T_K(x) |$. Then, we examine the characteristics of ES_k using the trained model M obtained by training with LS.

Figure 5.5 shows the generated images, scatter plots of the predicted probability $p_x(y)$ of TS data on the horizontal axis and $p_{T(x)}(y)$ of ES data on the vertical axis, and data distribution of $d_K(x)$ for eight different transformations. From the images, it can be seen that they all preserve the visual features of the data x chosen as the reference. The scatter plots show that the distribution differs depending on the data generation method for evaluation, and that its impact on the prediction probability is quite different.

The top left two are with Gaussian noise added, but the magnitude of the noise is different: the larger the noise, the greater the variability of the prediction probability. The distribution of $d_K(x)$ shows a sharp peak from the way the magnitude of the Gaussian noise given. It is easy to change $d_K(x)$ by increasing the noise, but on the other hand, the effect on the visual image is significant and the data is not appropriate for the evaluation. The two at the bottom left are corrupted data with a certain missing area. though $d_K(x)$ can be varied by changing the size of the missing region, the distribution of each $d_K(x)$ shows a sharp peak. Extremely large missing areas will destroy the original image and reduce the usefulness of the data for evaluation.

The top two on the right are semantic noise insertions (frame, underline). Due to the properties of the noise insertion method [35], the value of $p_{T(x)}(y)$ is approximately 1. It is clear from visual inspection that the frame has a larger value of $d_K(x)$: the distribution shows a sharp peak. The two affine transformations in the lower right corner have a sloping distribution. Taken as a whole, the distribution of $d_K(x)$ is skewed for each transformation method, but each shows a distinctive peak.

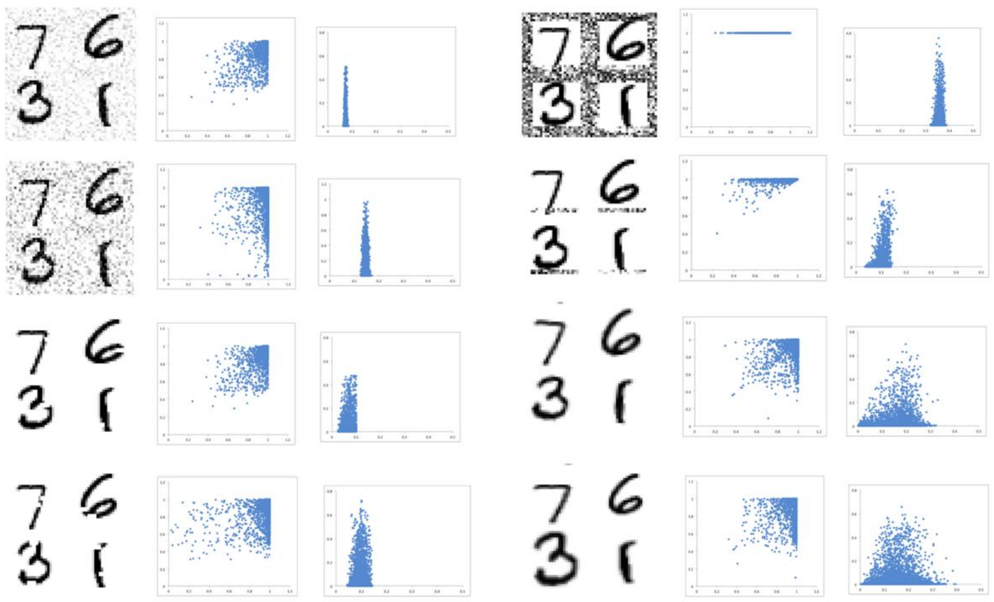


Figure 5.5 Data for Model Robustness Evaluation

Figure 5.6 plots the mean value of $d_K(x)$ on the horizontal axis and the mean value of $diff_K(x)$ on the vertical axis. he whole of the data for evaluation ES_k together means that the evaluation can be done with various $d_K(x)$ values. essence, the evaluation of model robustness is being conducted using a variety of data of different qualitative nature. In fact, it is important to know what kind of data of what nature to use for evaluation, which is the same "test case" issue important in traditional software testing as well.

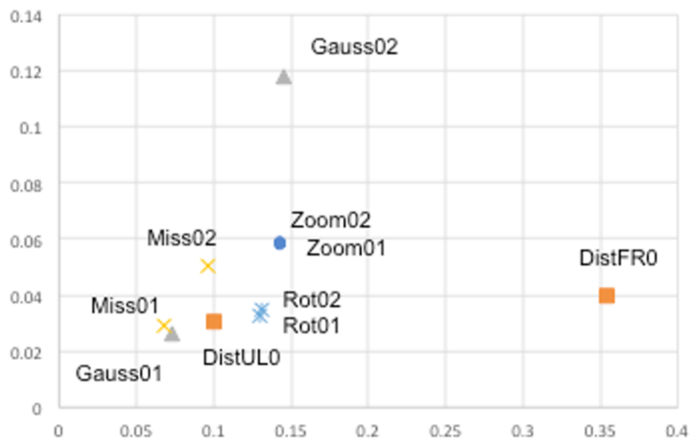


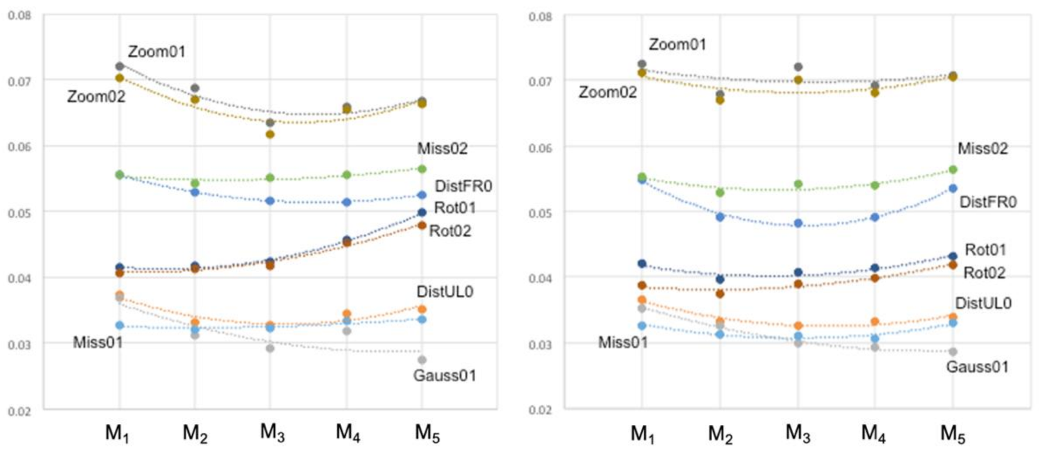
Figure 5.6 Mean Distance from Base Data

5.4.3.2 Indicator for Model Robustness

In the present experiment, the average value of $Diff_K$ defined previously is used as an indicator of empirical model robustness using data for evaluation. As in the case of model accuracy test, we measure the average value of each $Diff_K$ for the trained model M_j^P obtained using the training data LS_j^P. Figure 5.7 shows the results: (a) shows M_j^C and (b) shows M_j^N.

Although there are some differences among the evaluation data, the general trend is not monotonicity among M_K. In the case of model accuracy (Figure 5.4 (a)), the relative difference between M_K is less than 1%, whereas in Figure 5.7 the difference is about 10%. Differences in M_K, i.e., LS_j^P used for training, have a significant impact on model robustness.

From the way the training data LS_j^P constructed, we consider that the training data near the center (originally a region with large frequencies), where both sides of the neuron coverage distribution are removed, contributes much to the model robustness.



(a) Class-by-class extraction (b) Overall extraction

Figure 5.7 Indicators for Model Robustness

5.4.4 Debugging Strategy

We obtained trained training models M_j^P using training data LS_j^P that were systematically extracted based on the data distribution of neuron coverage N^{LS} for training data, and evaluated model accuracy and model robustness for these M_j^P. The results show that while the model accuracy of M_j^P is comparable, the model robustness is different. This is consistent with [Hypothesis 1] (neuron coverage is correlated with model robustness). By excluding the two-tailed hem of the N^{LS} data distribution as outliers, M_2^C and M_2^N were found to be generally reasonable due to their impact on model robustness. This is consistent with [Hypothesis 2] (N^{LS} data distribution is the criterion for outliers).

Here, we present briefly a debugging strategy for the training data, referring to Figure 5.8. When the neuron coverage of a given training dataset shows the data distribution on the left side of the figure, the central region of the range containing a reasonable number of data is selected, i.e., the training data corresponding to the two-sided hemisphere is discarded, and the training data showing the data distribution on the right side of the figure is obtained. Next, from among the new training data candidates, the training data whose neuron coverage values correspond to the central region is selected and added to the training data set until a sufficient number of data is obtained, while maintaining the data distribution on the right side. Additionally, debugging experiments based on this strategy were conducted by an independent group. The proposed method was applied to a CNN-based image classification task to confirm that the debugging strategy described above is feasible.

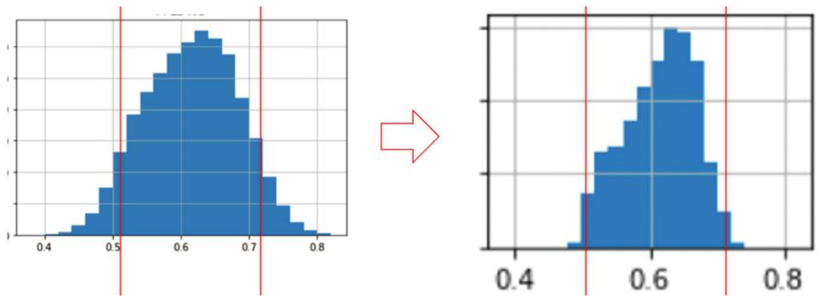


Figure 5.8 Debugging Strategy for Training Dataset

5.5 Final Remarks

To conclude this chapter, we review recent research in software engineering on the debugging problem of deep neural networks (DNN) software and position the technologies discussed in this chapter. In general, when DNN software exhibits defects, the direct cause is a flaw in the trained model. On the other hand, the construction of trained models involves diverse elements: (a) the machine learning infrastructure, (b) the learning model (the template for the DNN model), and (c) the training data. It is often not clear what the root cause is.

In practice, DNN software development is conducted on existing machine learning frameworks, and includes the programming tasks of scripts that use the APIs provided by the framework. At this time, the framework may have problems, and the machine learning mechanism (the program that solves the numerical optimization problem) itself may be inappropriate, or the library functions called by the script (e.g., learning model definition functions) may be flawed [49]. This is the case in (a) above and is the responsibility of the framework provider.

On the other hand, from the viewpoint of deep NN software development using machine learning frameworks, defects in the scripts are the cause of defects [50]. Depending on the type of defect, a learning model different from the required specification may be introduced, an error in data type (e.g., tensor dimension) may occur, or the configured hyperparameters may be inappropriate. This is the case in (b) above. In machine learning research, it is a matter of logical design to select an appropriate learning model for a given machine learning task. In contrast, the question is whether the appropriate "program" according to these higher-level specifications is being implemented. There are several studies, such as a method to localize the defective points of the learning model by monitoring the error function and gradient during the training process [51].

In general, the functional behavior of DNN software is governed by the training data. Although it is difficult to address this problem in general, there are studies that approach it as a fairness issue [52], where the sensitive attributes of the input data often affect the results. his method can be classified as (c) above in that it focuses on training data, but it is a feature selection problem and a debugging problem related to data definition.

This chapter addresses the problem in (c), where the root cause of the failure is a bias in the distribution of the training data. Intuitively, the problem set up is as follows. When a DNN model trained with given training data does not show the expected model performance (especially model robustness), the data distribution is transformed by selecting and adding training data to obtain a DNN model that achieves the desired performance.

In general, research approaching debugging problems from software engineering assumes that programs, such as scripts, have explicit symbolic representations. There has been little progress in exploring methods for debugging data distributions of continuous quantities. On the other hand, studies from machine learning recognize the importance of the data distribution and deal with the case by means of statistics-based methods, where the distribution function is known. However, machine learning deals with multi-dimensional vectors whose data distribution is difficult to represent in a straightforward manner. Even if a theoretical formulation is possible, it is not directly useful for the practice of debugging.

The method in this chapter uses the empirical distribution of data (continuous quantities) as guiding information for "debugging" while discarding individual data (discrete quantities). In particular, we simplified the problem by expressing the properties of the data to be debugged in terms of neuron coverage distributions, which can be attributed to the method of handling one-dimensional data distributions. It can be said to combine a statistical view with software engineering methods.

6 Evaluation and Improvement of Robustness

In this chapter, *robustness* means the ability that a machine-learned model keeps correct output even when noise is added to input (including adversarial examples). For example, it evaluates how much noise can be added to the model without changing the correct results. One of the measures of its robustness is the maximum safe radius (MSR). In this chapter, we explain adversarial example and the maximum safe radius in a classifier based on a feedforward neural network, and then report the results of a survey on techniques for estimating and increasing the maximum safe radius.

6.1 Robustness measure (maximum safe radius)

It is well known that machine-learned models on inference programs mis-classify input data even when very small perturbations are added. Such perturbated data are called adversarial examples [53], and adversarial examples have been actively researched in recent years. The set $Adv_\delta(x)$ of all adversarial examples contained in the δ-neighborhood (inside the sphere of radius $\delta \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers) of the input data sample $x \in \mathbb{R}^n$ is defined as follows:

$$Adv_\delta(x) = \{x' \mid \| x - x' \| \leq \delta \ \wedge \ f(x) \neq f(x')\},$$

where $f(x)$ is a function representing the machine-learned model that takes the input sample x and return the classification, and $\|x - x'\|$ is the distance between two data samples x and x'. The p-norm is often used to define the distance.

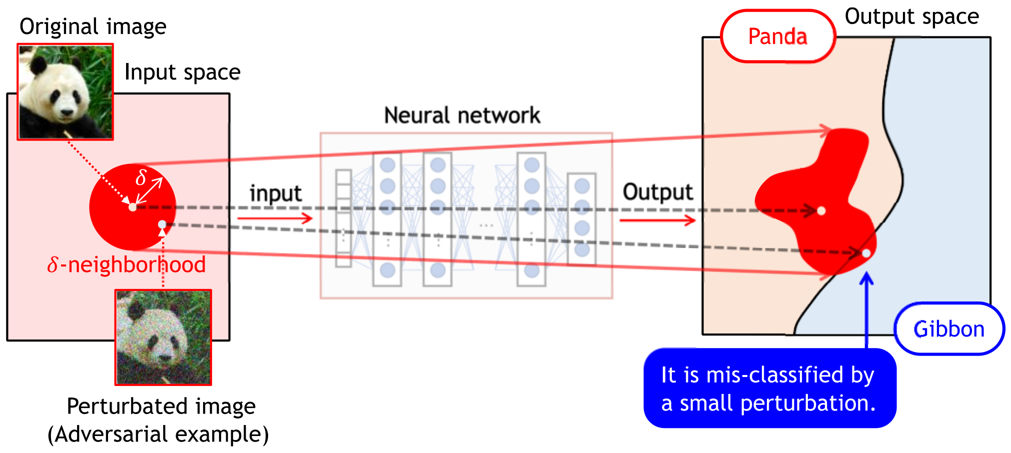


Figure 6.1 An adversarial example from an image of a panda, which is mis-classified into a gibbon

Adversarial examples are explained by Figure 6.1. The left side in Figure 6.1 shows the input space to the neural network and the right side shows the output space from the neural network. The center of the red sphere in the input space represents an original input image of a panda, and the inside of the sphere, whose radius is δ, (i.e., δ-neighborhood of the original image) represents the set of perturbated images obtained from the original image by adding noises whose sizes are less than δ. The set of outputs from the neural network for all the input images in the δ-neighborhood corresponds to the red region in the output space on the right. Here, a part (lower-right) of the red region in the output side is beyond the decision boundary and is mapped into the region of gibbons. It means misclassification, and the input images mapped to the lower-right part are adversarial examples.

If there is no adversarial example in the δ-neighborhood of the input data x (i.e., inside the sphere whose radius is δ and center is x), then δ is said to be the *safe radius* of x. Then, the maximum safe radius of x, denoted by $MSR(x)$, is defined as follows:

$$MSR(x) = \max\{\delta \mid Adv_\delta(x) = \emptyset\}.$$

When the maximum safe radius of x is large, it is difficult to generate adversarial examples. Therefore, the maximum safe radius can be used as a measure of the robustness to input perturbations, including adversarial examples, of machine-learned models.

The radius δ in Figure 6.1 is not a safe radius because some perturbated input images inside the δ-neighborhood are misclassified into gibbons. On the other hand, δ in the following Figure 6.2 is the maximum safe radius because all the input images inside the δ-neighborhood in Figure 6.2 are correctly classified.

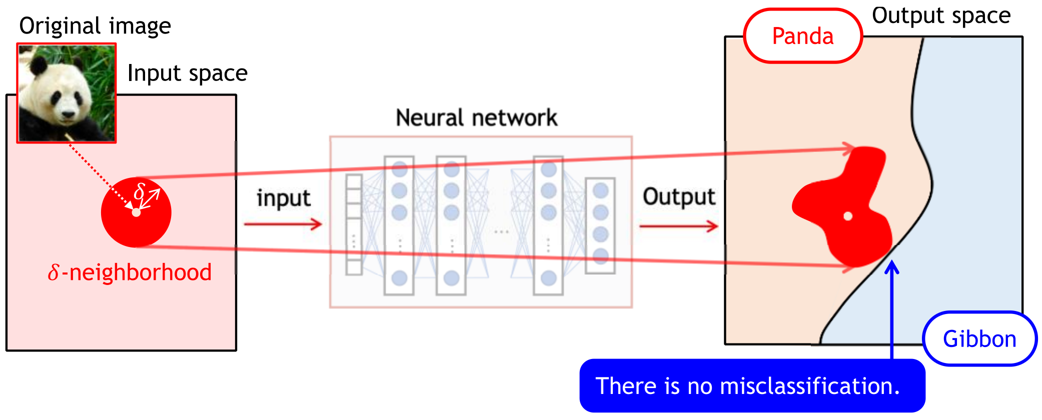


Figure 6.2 The maximum safe radius δ

6.2 A survey on methods for evaluation and improvement of robustness

Table 6.1 shows recent research papers on methods for evaluation and improvement of robustness, where each small box in the table represents a research paper with reference and the information on neural networks used in the experiments for evaluating the methods proposed in the paper. The information is useful for comparing applicable scales of the methods. Table 6.1 is categorized by the following perspectives:

Table 6.1 Methods for evaluation and improvement of robustness (MSR: Maximum Safe Radius)

			Evaluation of robustness	Improvement of robustness
Certified	Rigorous		Rigorous estimation of MSR **Katz et al. 2017 (Reluplex) [54]**: ACAS-XU-DNN, 300 ReLU nodes 6 hidden layers, (Limitation: hundreds of nodes) **Tjeng et al. 2019 [55]**: CIFAR-10, ResNet, 9-CNN, 2-layer, 107,496 ReLU units, 100~1,000 times faster than Reluplex	
Certified	Approximative	Deterministic	Estimation of a lower bound (LB) of MSR **Weng et al. 2018 (Fast-Lin) [56]**: CIFAR, 6-layer, 12,288 ReLU units About 10,000 times faster than Reluplex **Boopathy et al. 2019 (CNN-Cert)[57]**: CIFAR-10 (32x32x3), 5-layer, 10 filters, 29,360 hidden nodes, Faster than Fast-Lin	Training by detecting all the adversarial exes **Wong and Kolter 2018 [61]**: SVHN (32x32x3), 2-conv, 32-ch, 100, 10 hidden units, ReLU, (Non-applicable to ImageNet)
Certified	Approximative	Probabilistic	Estimation of a probabilistic LB of MSR **Weng et al. 2019 (PROVEN) [58]**: CIFAR, 5-layer, CNN, ReLU almost same as CNN-Cert	Randomized smoothing after training **Lecuyer at el. 2019 [62]**: ImageNet (299x299x3), Inception-v3 + auto-encoder **Cohen at el. 2019 [63]**: ImageNet (299x299x3), ResNet-50 (50-layer) Tighter certification than Lecuyer [62]
Uncertified			Estimation of an upper bound (UB) of MSR **Carlini and Wagner 2017 [59]**: ImageNet (299x299x3), Inception-v3 Estimation of an approximation of MSR **Weng et al. 2018 (CLEVER) [60]**: ImageNet (299x299x3), ResNet-50 (50-layer)	Training by detecting near adversarial exes **Madry et al. 2018 [64]**: CIFAR (32x32x3), 28-10 wide ResNet

- Columns in Table 6.1 (application):
  - Evaluation of robustness by estimating MSR
  - Improvement of robustness by increasing data samples with a specified MSR
- Row in Table 6.1 (certification and strictness):
  - Certification of no existence of adversarial examples in δ-neighborhood
    - Rigorous estimation of MSR
    - Approximative estimation of MSR
      - Deterministic (no adversarial example exist)
      - Probabilistic (the probability of no adversarial example is ρ%)
  - No certification of no existence of adversarial examples in δ-neighborhood

The methods in Table 6.1 are explained in the following Subsections 6.2.1~6.2.7.

6.2.1 Certified and rigorous evaluation of robustness

Katz et al. [54] proposed a method, Reluplex, to verify that a machine-learned model satisfies given properties. A demonstration tool that implements the method Reluplex has also been released. Properties are constraints on input-output relations of machine-learned models, and Reluplex can exhaustively and rigorously (soundly and completely) verify that there is no adversarial example in the δ-neighborhood of the input data sample. Therefore, the maximum safe radius (MSR) can be estimated by checking the existence of adversarial examples by changing the radius δ with binary search. Reluplex is an extended Simplex method (one of solvers for linear programming problems) with rules for the ReLU function and it is implemented by a satisfiability-checking tool (SMT-Solver) with a module for the theory of real numbers. Reluplex is a powerful tool to verify properties in addition to robustness, but the computational cost is expensive and the number of neurons it can verify is a few hundred ReLUs at most.

Tjeng et al. [55] proposed an efficient method for estimating maximum safe radii. Then, they implemented the method on a mixed integer linear programming (MILP) solver and demonstrated that the tool can exactly estimate the maximum safe radii of a neural network with 100,000 ReLU-type neurons. Although it is still difficult to apply the rigorous solver-based tools to practical large-scale machine-learned models, the scalability is being improved.

6.2.2 Certified, approximative, and deterministic evaluation of robustness

Weng et al. [56] proposed a method, Fast-Lin, to approximate the maximum safe radii of ReLU-type neural network. Fast-Lin linearly approximates the output region with a polytope and estimates an approximation δ that is slightly smaller than the maximum safe radius, as shown in Figure 6.3. It is guaranteed that there is no adversarial example inside the δ-neighborhood because the approximation δ does not exceed the maximum safe radius (i.e. sound). It means δ is a safe radius and is a lower bound of the maximum safe radius ($\delta \leq MSR(x)$). It was reported that Fast-Lin is 10,000 times faster than the rigorous method Reluplex by approximative convex outer polytopes.

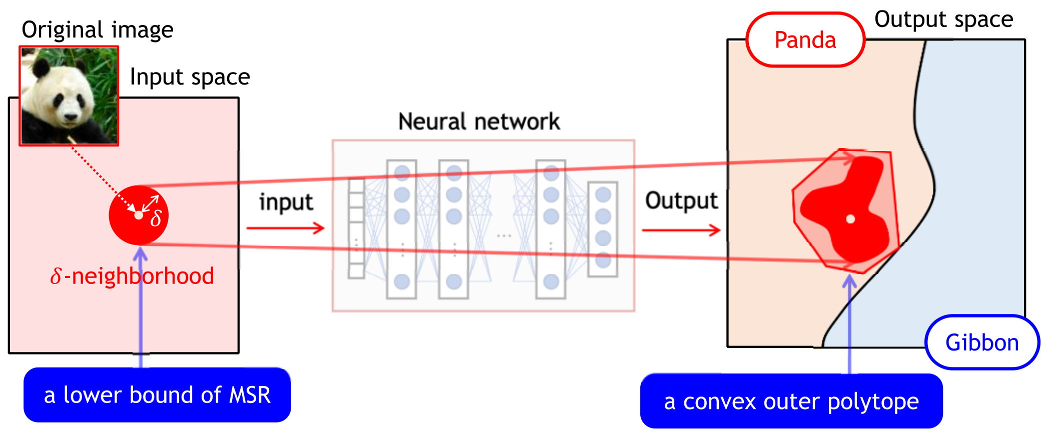


Figure 6.3 An approximation δ that is slightly smaller than the maximum safe radius (MSR)

Boopathy et al. proposed CNN-Cert, which is an improved version of Fast-Lin [57]. CNN-Cert also supports convolutional networks including not only the activation function ReLU but also sigmoid, tanh, and arctan, and it improves approximation accuracy and is faster than Fast-Lin.

6.2.3 Certified, approximative, and probabilistic evaluation of robustness

Weng et al. [58] proposed a method, PROVEN, to approximate probabilistic maximum safety radii. As shown in Figure 6.4, the probabilistic maximum safe radius δ with a probability ρ means that there is no adversarial example inside the δ-neighborhood with a probability ρ. In other words, it permits the existence of adversarial examples with the probability $(1-\rho)$. PROVEN has been developed based on CNN-Cert, and the computational complexity has not significantly increased from CNN-Cert.

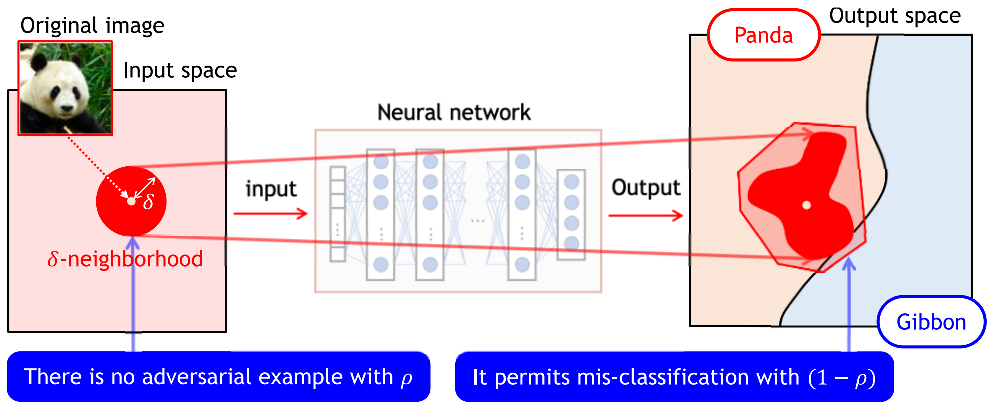


Figure 6.4 An approximation δ that is slightly smaller than the probabilistic MSR with ρ

6.2.4 **Uncertified evaluation of robustness**

Carlini and Wagner [59] proposed a method to detect the (almost) closest adversarial example to the input data sample x and estimate the distance δ as an approximative maximum safety radius by using an existing optimization tool (Adam). However, it is not guaranteed that the distance δ estimated by the method is the shortest distance to the adversarial example, and there is a possibility that there are adversarial examples closer than the distance. In other words, it is an upper bound of the maximum safe radius ($MSR(x) \leq \delta$). Although it is not guaranteed that the distance δ estimated by the method is a safe radius, it is often used for evaluation in recent papers on robustness as a measure of the maximum safe radius.

Weng et al. [60] proposed the method CLEVER to estimate an approximate maximum safe radius as an evaluation measure of robustness independent of attack methods. It was reported that the method could be applied to relatively large neural networks and the image recognition model Inception-v3 was evaluated in about 10 seconds. The method estimates an approximative maximum safe radius based on the maximum effect in output caused by small changes in input, where the maximum effect is approximated by the extreme value theory. As shown in Figure 6.5, the estimated value δ can be larger than the maximum safe radius, and thus there is a possibility that adversarial examples exist inside the δ-neighborhood (i.e., it is not guaranteed that δ is the safe radius).

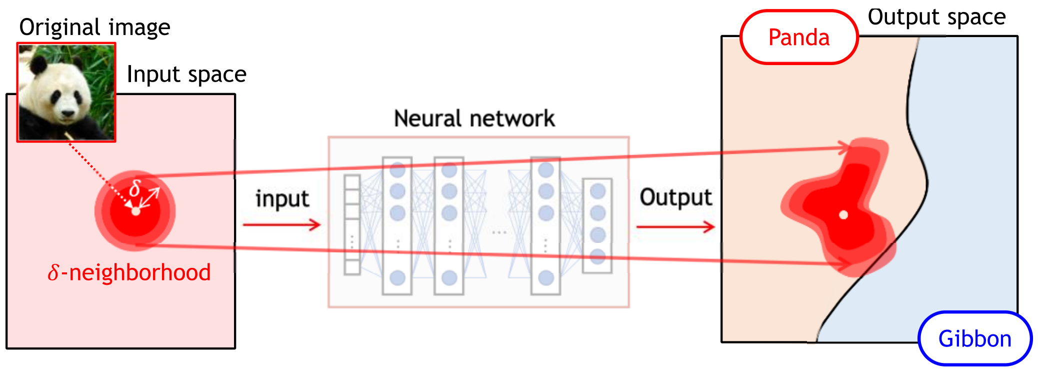


Figure 6.5 An approximation of the maximum safe radius (uncertified)

6.2.5 **Certified, approximative, and deterministic improvement of robustness**

Wong et al. [61] proposed a method (robust training) to train such that the maximum safe radius of each data in the training dataset to be a specified value δ. Although this method does not guarantee that the maximum safe radius δ is obtained for every training data sample after training, it also gives a method to estimate an approximative value (a safe radius) of the maximum safe radius for each input data sample. In the robust training, neural networks try to learn such that they correctly make inferences for not only training data samples but also the δ-neighborhood of every sample.

A sketch of the robust training is shown in Figure 6.6, where the black dotted line in the output space represents the decision boundary learned by a normal training, and the red solid line represents the decision boundary learned by the robust training. The six training data samples in the input space are correctly classified by both the boundaries, but some data in the δ-neighborhood of each sample are misclassified by the dotted boundary (normal training). On the other hand, data in the δ-neighborhood of each sample are also learned in the robust training as shown in the red boundary. The robust training can guarantee some safe radii, but it is difficult to apply the training to practical large scale neural networks due to the low scalability. Wong et al. [61] reports that the robust training was successfully applied to the datasets of images, MNIST (28×28) and SVHN (32×32) but was not applicable to ImageNet (256×256).

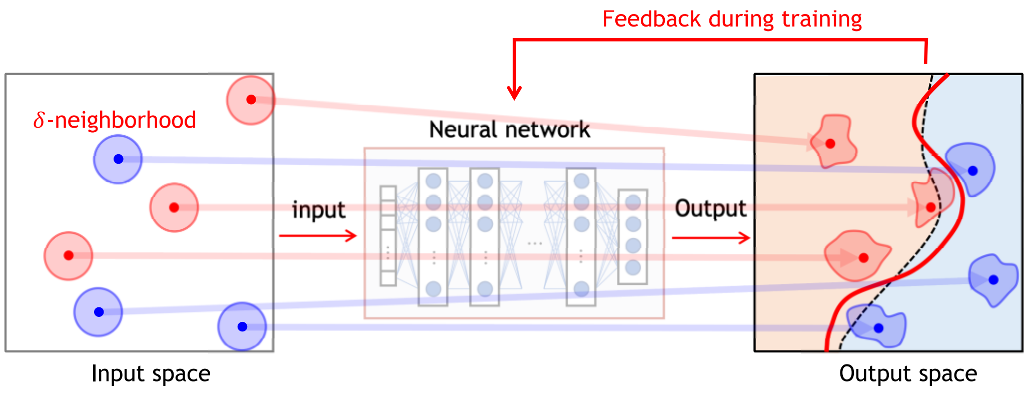


Figure 6.6 Robust-trining by input data with δ-neighberhood

6.2.6 Certified, approximative, and probabilistic improvement of robustness

Lecuyer et al. [62] proposed a method to estimate maximum safe radii that can be probabilistically guaranteed by randomized smoothing. In the randomized smoothing, the inference for the same input is repeated in a neural network where a noise layer is added after training, and the final output is the average of the outputs obtained by the repeated inferences.

A sketch of the randomized smoothing is shown in Figure 6.7, where the black dotted line in the output space represents the decision boundary without randomized smoothing, and the red solid line represents the decision boundary with randomized smoothing. The randomized smoothing of Lecuyer et al. [62] improves robustness by smoothing decision boundaries with certification of safe radii and has been successfully applied to guarantee the robustness of machine learned models for large-scale input data such as ImageNet ($299 \times 299 \times 3$). When the variance of the added noise is increased, the guaranteed safe radius also increases, but on the other hand, the correctness (e.g., accuracy) decreases. Lecuyer et al. [62] applied the technique of differential privacy, where the output for two similar inputs is made statistically indistinguishable, to clarify the relations between certifiable approximative probabilistic maximum safe radii, the standard deviation of noise, the number of inferences, and so on.

Cohen et al. [63] proposed a randomized smoothing based method that can estimate tighter certifiable approximative probabilistic maximum safe radii than one of Lecuyer et al. [62].

Although randomized smoothing needs repeated inferences (tens or hundreds of times experimentally) for an input, it can probabilistically guarantee robustness even for large-scale networks.

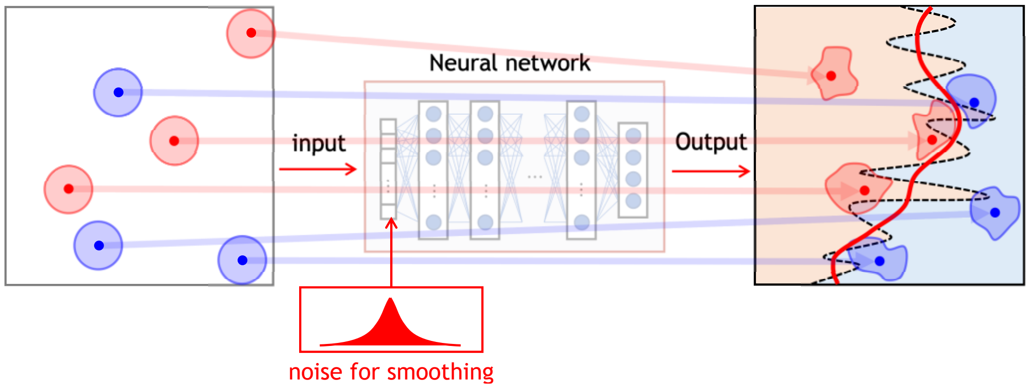


Figure 6.7 Improvement of robustness by randamized smoothing

6.2.7 Uncertified improvement of robustness

Madry et al. [64] proposed a method (adversarial training) to train such that maximum safe radius of each data in the training dataset to be a specified value δ. In the adversarial training, samples to be potentially adversarial examples in δ-neighborhood are detected during training and are also used as training data. Compared to the robust training of Wong et al. [61], the adversarial training cannot guarantee robustness, but it is more applicable to larger networks. In addition, compared to randomized smoothing, the adversarial training does not require repeated inferences.

6.3 Conclusion

In general, improvement of robustness tends to decrease accuracy, and currently accuracy is often more important. However, if robustness is not considered, accuracy may rapidly decrease even by small input perturbations. Therefore, robustness is important in critical systems. The methods related to the maximum safe radius, which is a measure of robustness, explained in this chapter have been proposed recently, and environments for applying such methods have not been established well yet. Since such methods have been experimentally applied also to practical machine learned models, we think that the maximum safe radius can be one of measures of robustness in a few years.

7 Estimation of Generalization Error Bounds

In Machine Learning Quality Management (MLQM) Guideline [1] introduced in Chapter 1, the following two internal quality properties are described:

– *the correctness of trained models*, that represents that trained models correctly behave for datasets (data samples), and
– *the stability of trained models*, that represents that trained models reasonably behave even for unseen input data not included in datasets.

Although widely used measures such as recall, precision, and accuracy based on testing datasets, are useful for evaluating the correctness of trained models, they are not sufficient for guaranteeing the stability of trained models, that requires stability even for unseen data.

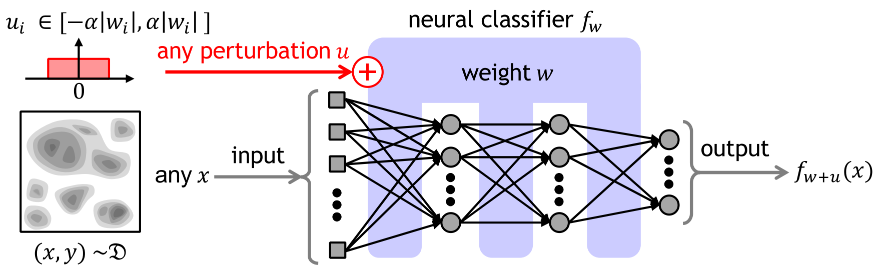


Figure 7.1 Inference by a neural classifier f_w with weight-perturbations

In this chapter, in order to statistically guarantee such stability with a confidence for *any* input, including unseen input, we explain how to estimate the upper bounds of *weight-perturbed generalization errors* of *neural classifiers* that are feed-forward neural networks trained for classification. The neural classifiers are henceforth referred to simply as "*classifiers*". The weight-perturbed generalization error represents the expected value of the misclassification-rate of the classifier when perturbations are added on weight-parameters between neurons during inference for any input, as shown in Figure 7.1. The weight-perturbed generalization errors are thought to be useful for evaluating stability because Jiang et al. [65] reported that such errors have high correlation with generalization performance.

At first, weight-perturbed generalization errors are defined in Section 7.1, and formal expressions are presented for estimating their upper bounds in Section 7.2. Then, it is explained how to determine thresholds for worst weight-perturbations in Section 7.3, and it is demonstrated by experiments in Section 7.4. Finally, related works are introduced in Section 7.5, and it is concluded that weight-perturbed generalization errors are useful for evaluating stability of classifiers in Section 7.6.

7.1 Weight-perturbed generalization error

In this chapter, the following two types of weight-perturbations are used:

- *random weight-perturbations*, that are randomly selected from uniform distribution with specified range, and
- *worst weight-perturbations*, that are selected towards misclassification within the range.

Although even worst perturbations do not necessarily cause misclassification, perturbations really causing misclassification are called *adversarial perturbations*.

Figure 7.2 shows examples of decision boundaries and output deviation areas of weight-perturbed classifiers A and B. In the magnified part in Figure 7.2, the central dot represents the output of the classifier A without weight-perturbation and the small area around the dot represents the possible output deviation range when weight-perturbations within a specified range are added. The shaded area in the magnified part corresponds to the set of adversarial perturbations, that change the output to misclassification. In Figure 7.2, random perturbations (e.g., natural noise) can degrade the classifier A, but they little degrade classifier B because the areas of adversarial perturbations in the classifier B are very small. Nevertheless, it is possible to search rare adversarial perturbations even for the classifier B, and therefore it is important to check the existence (i.e., risk) of adversarial perturbations. The worst perturbations are useful for evaluating such risk.

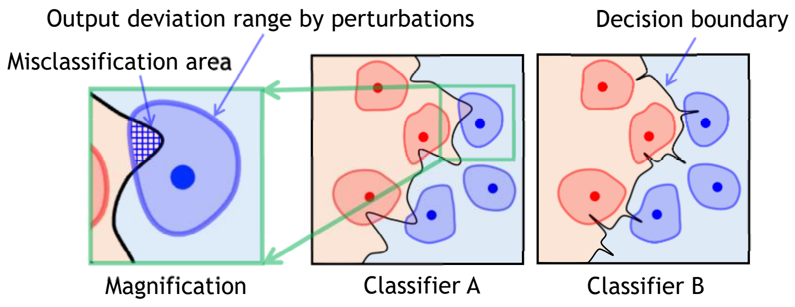


Figure 7.2 Decision boundaries and output deviation areas of weight-perturbed classifiers

In this chapter, a neural classifier is modeled by a *function* f_w that represents the relation between input x and output y; thus $y = f_w(x)$, where $w \in \mathbb{R}^\omega$ represents the weights (i.e., training parameters) on connections between neurons in the neural network, and ω is the number of weights. The set $U_{w,\alpha}$ of weight-perturbations is defined such that the ratio of magnitude of a perturbation u_i to the magnitude of each weight w_i is bounded by a given constant α, as follows:

$$U_{w,\alpha} \coloneqq \{\,(u_1, \dots, u_\omega) \in \mathbb{R}^\omega \mid \forall i.\, |u_i| \le \alpha |w_i|\,\}. \tag{7.1}$$

The perturbations in the set $U_{w,\alpha}$ are often called *magnitude-aware perturbations* [65]. The multivariate uniform-distribution for randomly selecting a perturbation from the set $U_{w,\alpha}$ is denoted by $\mathcal{U}_{w,\alpha}$, and therefore, if $u \sim \mathcal{U}_{w,\alpha}$, then $u_i \sim \mathbf{U}(-\alpha|w_i|, \alpha|w_i|\,)$. Henceforth, if the subscripts w and α are clear from context, then $U_{w,\alpha}$ and $\mathcal{U}_{w,\alpha}$ are simply denoted by U and $\mathcal{U}$, respectively.

Then, for a pair (x, y), for any weight-perturbation $u \sim \mathcal{U}$, the expected value, called the *weight-perturbed individual error*, of misclassification rate is defined as follow:

$$\mathbf{r}^{\alpha}_{(x,y)}(f_w) \coloneqq \mathbb{E}_{u \sim \mathcal{U}}[\ell(f_{w+u}(x), y)], \tag{7.2}$$

where $\ell(y, y')$ is a loss function, and the following 0-1 loss function is used in this chapter:

$$\ell(y, y') \coloneqq \mathbb{I}[y \neq y'], \tag{7.3}$$

where $\mathbb{I}[b]$ is the following indicator function, and therefore, the loss function (7.3) above means that the loss is 1 if misclassified, and 0 otherwise.

$$\mathbb{I}[b] \coloneqq \text{ if } b \text{ then } 1 \text{ else } 0 \tag{7.4}$$

The weight-perturbed individual error corresponds to the ratio of the shaded area (i.e., misclassification) to the possible deviation area in the magnified part of Figure 7.2.

Now, the *randomly weight-perturbed generalization error* $\mathbf{R}^{\alpha}(f_w)$ of the classifier f_w is defined as the expected value of the weight-perturbed individual error $\mathbf{r}^{\alpha}_{(x,y)}(f_w)$ for any pair $(x, y) \sim \mathcal{D}$ as follows:

$$\mathbf{R}^{\alpha}(f_w) \coloneqq \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\mathbf{r}^{\alpha}_{(x,y)}(f_w)\right], \tag{7.5}$$

where $\mathcal{D}$ is the distribution of pairs (x, y) of input x and output (class) y.

The randomly weight-perturbed generalization error hardly increases when the weight-perturbed individual error is very small as shown in the classifier B in Figure 7.2. Therefore, for evaluating the *risk* where adversarial weight-perturbations exist more than a small threshold, the *worst weight-perturbed generalization error* $\mathbf{W}^{\alpha}_{\theta}(f_w)$ of the classifier f_w is defined as the expected value of 0/1-boolean value that is 1 if the weight-perturbed individual error $\mathbf{r}^{\alpha}_{(x,y)}(f_w)$ is greater than the threshold $\theta_{(x,y)}$ and 0 otherwise for any pair $(x, y) \sim \mathcal{D}$ as follows:

$$\mathbf{W}^{\alpha}_{\theta}(f_w) \coloneqq \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\mathbb{I}\left[\mathbf{r}^{\alpha}_{(x,y)}(f_w) > \theta_{(x,y)}\right]\right] \tag{7.6}$$

$$\Theta \coloneqq \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\theta_{(x,y)}\right] \tag{7.7}$$

where $\theta_{(x,y)}$ is the threshold that can depend on (x, y), and Θ is the expected value for $\mathcal{D}$. The threshold means the accepted ratio of existence of adversarial perturbations. If "*worst* weight-perturbation" is *exactly* defined in $\mathbf{W}^{\alpha}_{\theta}(f_w)$, then the threshold $\theta_{(x,y)}$ should be zero because it must be checked whether one or more adversarial weight-perturbations exist or not. However, it is reasonable to set the threshold to be appropriately small values because the zero-threshold often too strong to apply to practical classifiers and it is often unrealistic from the perspective of computation cost. The appropriate thresholds are explained later in Section 7.3.

7.2 Estimation of weight-perturbed generalization error bounds

Although it is difficult in general to exactly compute weight-perturbed generalization errors because there can be infinitely many data pairs (x, y) and perturbations u, there are various existing works on the bounds of generalization errors. In this section, expressions for estimating randomly weight-perturbed generalization errors and worst weight-perturbed generalization errors are introduced.

For randomly weight-perturbed generalization errors, there are many existing works. For example, by the combination of the Maurer bounds (Theorem 5 in [67]) and the Sample Convergence bounds (Theorem 2.5 in [68]), the following inequality (7.8) holds with probability (i.e., confidence) at least $(1-\delta)$ for any $\delta \in (0,1)$ [69]:

$$\mathbf{R}^{\alpha}(f_w) \leq \overline{\mathbf{R}}^{\alpha}_{T,V,\delta_0,\delta}(f_w), \tag{7.8}$$

where $T \sim \mathcal{D}^n$ is a testing dataset (size n) that is not used for training, $V \sim \mathcal{U}^m$ is a set (size m) of samples of random weight-perturbations, the uncertainty $\delta_0 \in (0,\delta)$ is the acceptable degradation of confidence caused by using the perturbation samples instead of any perturbation, and the right-hand side $\overline{\mathbf{R}}^{\alpha}_{T,V,\delta_0,\delta}(f_w)$ is defined by

$$\overline{\mathbf{R}}^{\alpha}_{T,V,\delta_0,\delta}(f_w) \coloneqq kl^{-1}\left(\overline{\overline{\mathbf{R}}}^{\alpha}_{T,V,\delta_0}(f_w), \frac{1}{n}\ln\left(\frac{2\sqrt{n}}{\delta-\delta_0}\right)\right), \tag{7.9}$$

where $\overline{\overline{\mathbf{R}}}^{\alpha}_{T,V,\delta_0}(f_w)$ is an upper bound of the weight-perturbed testing error $\widehat{\widehat{\mathbf{R}}}^{\alpha}_{T,V}(f_w)$ by the testing dataset T and perturbation samples V, and they are defined by

$$\overline{\overline{\mathbf{R}}}^{\alpha}_{T,V,\delta_0}(f_w) \coloneqq kl^{-1}\left(\widehat{\widehat{\mathbf{R}}}^{\alpha}_{T,V}(f_w), \frac{1}{m}\ln\left(\frac{2}{\delta_0}\right)\right), \tag{7.10}$$

$$\widehat{\widehat{\mathbf{R}}}^{\alpha}_{T,V}(f_w) \coloneqq \frac{1}{nm}\sum_{u \in V}\sum_{(x,y)\in T} \ell(f_{w+u}(x), y), \tag{7.11}$$

where $kl^{-1}(q,b)$ is defined by

$$kl^{-1}(q,b) \coloneqq \sup\{\, p \in [q,1] \mid kl(q \parallel p) \leq b \,\}, \tag{7.12}$$

and $kl(q \parallel p)$ is the binary Kullback-Leibler divergence, and is defined as follows:

$$kl(q \parallel p) \coloneqq q\ln\left(\frac{q}{p}\right) + (1-q)\ln\left(\frac{1-q}{1-p}\right). \tag{7.13}$$

On the other hand, for the worst weight-perturbed generalization errors, there are some (not many) existing works on estimating them. For example, the following inequality (7.14) holds with probability (i.e., confidence) at least $(1-\delta)$ for any $\delta \in (0,1)$ [70]:

$$\mathbf{W}^{\alpha}_{\theta}(f_w) \leq \overline{\mathbf{W}}^{\alpha}_{\theta,T,V,\delta_0,\delta}(f_w), \tag{7.14}$$

where the parameters $T \sim \mathcal{D}^n, V \sim \mathcal{U}^m$, and $\delta_0 \in (0,\delta)$ are the same as the parameters in (7.8), and the right-hand side $\overline{\mathbf{W}}^{\alpha}_{\theta,T,V,\delta_0,\delta}(f_w)$ is defined by

$$\overline{\mathbf{W}}^{\alpha}_{\theta,T,V,\delta_0,\delta}(f_w) \coloneqq kl^{-1}\left(\overline{\overline{\mathbf{W}}}^{\alpha}_{\theta,T,V,\delta_0}(f_w), \frac{1}{n}\ln\left(\frac{2}{\delta-\delta_0}\right)\right), \tag{7.15}$$

where $\overline{\overline{\mathbf{W}}}^{\alpha}_{\theta,T,V,\delta_0}(f_w)$ is an upper bound of the ratio of the size of a *risky* dataset to the size of the testing dataset, and it is defined from an upper bound $\bar{\mathbf{r}}^{\alpha}_{(x,y),V,\delta_1}(f_w)$ of the weight-perturbed individual error as follows:

$$\bar{\bar{\mathbf{W}}}^{\alpha}_{\theta,T,V,\delta_0}(f_w) := \frac{1}{n}\sum_{(x,y)\in T}\left[\mathbb{1}\left[\bar{\mathbf{r}}^{\alpha}_{(x,y),V,\frac{\delta_0}{n}}(f_w) > \theta_{(x,y)}\right]\right], \tag{7.16}$$

$$\bar{\mathbf{r}}^{\alpha}_{(x,y),V,\delta_1}(f_w) := kl^{-1}\left(\hat{\mathbf{r}}^{\alpha}_{(x,y),V}(f_w), \frac{1}{m}\ln\left(\frac{2}{\delta_1}\right)\right), \tag{7.17}$$

$$\hat{\mathbf{r}}^{\alpha}_{(x,y),V}(f_w) := \frac{1}{m}\sum_{u\in V}\ell(f_{w+u}(x), y). \tag{7.18}$$

The expected value Θ of the threshold θ is explained in the next Section 7.3.

7.3 Thresholds for worst weight-perturbations

The randomly weight-perturbed generalization error bound $\bar{\mathbf{R}}^{\alpha}_{T,V,\delta_0,\delta}(f_w)$ can be estimated by measuring the average $\hat{\hat{\mathbf{R}}}^{\alpha}_{T,V}(f_w)$ of misclassification rates of the classifier f_w for the testing dataset T and perturbation samples V. On the other hand, for estimating the worst weight-perturbed generalization error bound $\bar{\mathbf{W}}^{\alpha}_{\theta,T,V,\delta_0,\delta}(f_w)$, it is important how to determine the threshold θ, and there are two practical and reasonable approaches for deciding the threshold. One of them, called *fixed threshold*, is explained Subsection 7.3.1, and the other one, called *adaptive threshold*, is explained in Subsection 7.3.2.

7.3.1 Fixed threshold

Let $V\sim\mathcal{U}^m$ be a set of weight-perturbation samples and $T\sim\mathcal{D}^n$ be a testing dataset. Then, T is partitioned to the risky dataset T_1 and the rest set $T_0 = T - T_1$, where T_1 is defined by

$$T_1 := \{(x,y)\in T \mid \exists u \in V.\ f_{w+u}(x) \neq y\}. \tag{7.19}$$

In this case, the testing error bound $\bar{\bar{\mathbf{W}}}^{\alpha}_{\theta,T,V,\delta_0}(f_w)$ defined in the expression (7.16) can be simply estimated by

$$\bar{\bar{\mathbf{W}}}^{\alpha}_{\theta^{fix}_{m,n,\delta_0},T,V,\delta_0}(f_w) = \frac{n_1}{n}, \tag{7.20}$$

where n_1 is the size of T_1, and $\theta^{fix}_{m,n,\delta_0}$ is called the *fixed threshold* and is defined by

$$\theta^{fix}_{m,n,\delta_0} := 1 - \left(\frac{\delta_0}{2n}\right)^{\frac{1}{m}}. \tag{7.21}$$

Therefore, it is sufficient for the fixed threshold to count the number of risky data samples where one or more adversarial weight-perturbations exists in the set V of samples. In this case, the expected threshold Θ is equal to $\theta^{fix}_{m,n,\delta_0}$ because it is independent from individual data.

If a fixed threshold θ_0 is specified, the size m of weight-perturbation samples can be determined to satisfy the following condition that is obtained by transforming the expression (7.21):

$$m \geq \frac{\ln(\delta_0 \,/\, (2n))}{\ln(1-\theta_0)}\,. \tag{7.22}$$

For example, if $n = 5000$, $\delta_0 = 0.05$, and $\theta_0 = 0.01$ (i.e., the fixed threshold 1%), then the required size of weight-perturbation sample is 1215. Higher robustness requires a smaller threshold, but it means to require more samples (i.e., more computation cost). It will be necessary to set a practically reasonable threshold.

7.3.2 Adaptive threshold

In the case that the ratios of adversarial weight-perturbations are very small as shown in the classifier B in Figure 7.2, the possibility that adversarial weight-perturbations are contained in the set of randomly selected samples is very low. For effectively finding such adversarial weight-perturbations, it is useful to search them based on gradients of loss functions (e.g., Algorithm 3 in [65]), although such search cannot guarantee that there is no adversarial weight-perturbations even if it cannot find them. Therefore, both of random samples and such search are used in [70], where gradient-based search is applied to the dataset T_0 that is the subset of the testing dataset T explained in Subsection 7.3.1. Then, the dataset T_0 is partitioned to the risky dataset T_{01} and the rest set $T_{00} = T_0 - T_{01}$, where T_{01} is the dataset, for which one or more adversarial weight-perturbations are found by the search (not found by the random samples). In this case, the testing error bound $\bar{\bar{\mathbf{W}}}^{\alpha}_{\theta,T,V,\delta_0}(f_w)$ defined in the expression (7.16) is simply estimated by

$$\bar{\bar{\mathbf{W}}}^{\alpha}_{\theta^{ada}_{m,n_{00},\delta_0},T,V,\delta_0}(f_w) = \frac{n_{01}+n_1}{n}\,, \tag{7.23}$$

where n_{00}, n_{01}, and n_1 are the sizes of T_{00}, T_{01} and T_1, respectively, and $\theta^{ada}_{m,n_{00},\delta_0,(x,y)}$ is called the *adaptive threshold* and is defined by

$$\theta^{ada}_{m,n_{00},\delta_0,(x,y)} \coloneqq \begin{cases} 0 & \text{if } (x,y) \in T_{01} \cup T_1 \\ \theta^{fix}_{m,n_{00},\delta_0} & \text{if } (x,y) \in T_{00} \end{cases}\,. \tag{7.24}$$

The expected threshold Θ of the adaptive threshold can be bound by $\bar{\Theta}$ defined by

$$\bar{\Theta} = \frac{\bar{n}_{00}}{n}\theta^{fix}_{m,\bar{n}_{00},\delta_0}, \tag{7.25}$$

where $(\bar{n}_{00}/n)$ is the upper bound of (n_{00}/n) and can be estimated by

$$\bar{n}_{00} \coloneqq n \times kl^{-1}\left(\frac{n_{00}}{n}, \frac{1}{n}\ln\left(\frac{2}{\delta}\right)\right). \tag{7.26}$$

7.4 Estimation experiments of weight-perturbed generalization error bounds

In this section, it is reported that experiments and the results for estimating weight-perturbed generalization error bounds by applying expressions (7.9) and (7.15) explained Section 7.3. In the experiments, we estimated the weight-perturbed generalization error bounds of 8 classifiers, named CNN#1~8, that are trained convolutional neural networks by the dataset MNIST (pixels: 28×28, grayscale: [0,1]) of handwritten digit images with the training hyper-parameters shown in Table 7.1. The size n of testing dataset T is 5000, the size m of the set V of random weight-perturbation samples is 1215, the uncertainty δ (i.e., the acceptable degradation of confidence) for generalization error bounds is 0.1 (10%), and the uncertainty δ_0 caused by random weight-perturbation samples is 0.05 (5%). Here, the size 1215 of perturbation samples is determined for making the fixed threshold 1%, as explained Subsection 7.3.1. For the adaptive threshold, an I-FGSM (iterative fast gradient sign method) like algorithm is used for searching for adversarial weight-perturbations in vertexes of the possible perturbation area (the hyper-rectangular). Perturbations are added to the weights and the biases (the total number is 121930) in the CNNs, but they are not added to training parameters (scale γ and shift β) of the batch normalization.

Table 7.1 The training hyper-parameters for the 8 classifiers CNN#1~8

Batch normalization		unavailable		available	
Dropout rate		0	0.1	0	0.1
L^2-Regularization	0	#1	#3	#5	#7
	0.001	#2	#4	#6	#8

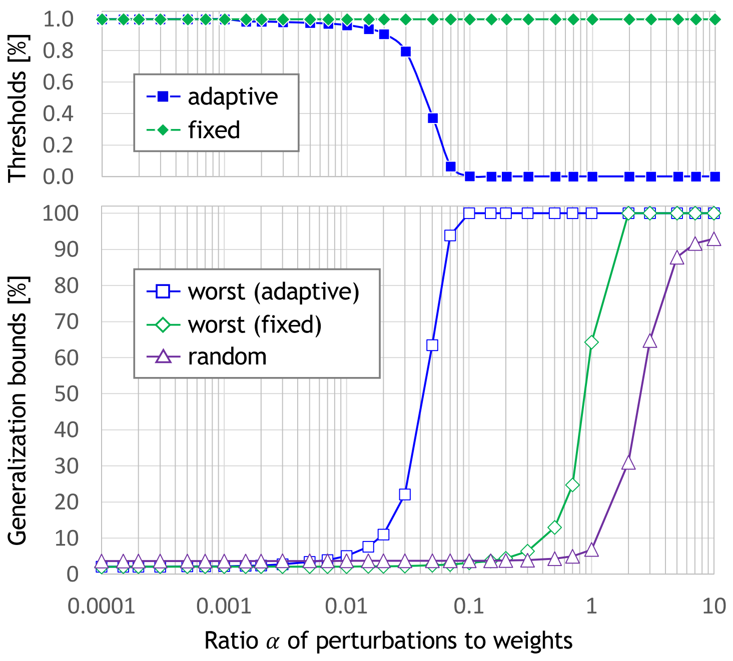


Figure 7.3 The estimation results of randomly/worst perturbed generalization bounds of CNN#4

Figure 7.3 shows the estimation results (confidence 90%) of randomly weight-perturbed generalization error bounds and worst weight-perturbed generalization error bounds for the fixed threshold and the adaptive threshold of CNN#4. The horizontal axis represents the ratio α of weight-perturbations to weights. The errors start to increase at $\alpha = 0.01$, 0.1, and 1, for worst weight-perturbations with the adaptive threshold, worst weight-perturbations with the fixed threshold, and random weight-perturbations, respectively. Figure 7.3 also shows the bounds of the expected values of thresholds, and the adaptive threshold decreases while the generalization error (i.e., the number of found adversarial perturbations) increases.

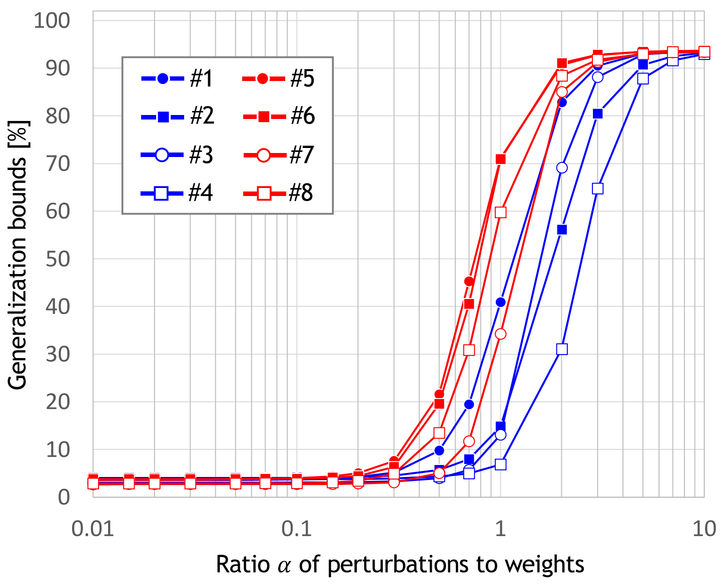


Figure 7.4 The estimation results of randomly perturbed generalization error bounds

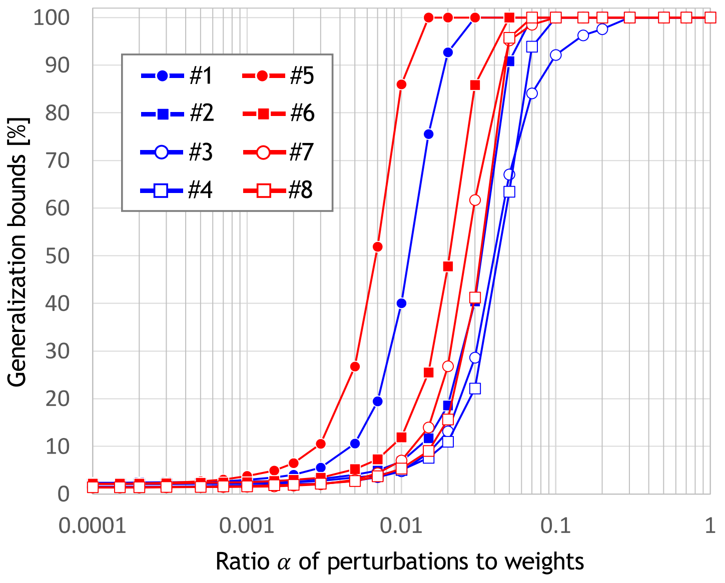


Figure 7.5 The estimation results of worst perturbed generalization error bounds (adaptive)

Figure 7.4 and Figure 7.5 show the estimation results of randomly and worst weight-perturbed generalization error bounds with the adaptive threshold, respectively. It is noted that the classifiers have different tendencies for random perturbations and worst perturbations. For example, the classifier CNN#3 is clearly less robust (faster increasing the error) for random perturbations than CNN#4, while there is little difference between CNN#3 and CNN4 for worst perturbations.

For the cases of $\alpha = 0.3$ in Figure 7.4 and $\alpha = 0.003$ in Figure 7.5, the generalization gap Δ_G (the difference between the generalization bound and the perturbed testing error), the perturbation gap Δ_P (the difference between the perturbed testing error and testing error), and the testing error TE for each classifier are shown in Figure 7.6 and Figure 7.7. The generalization gaps are about 1~2%, and therefore, tight bounds are estimated. Even when there are only slight differences between the testing errors TE, there are often clear differences between perturbed testing errors $(TE + \Delta_P)$.

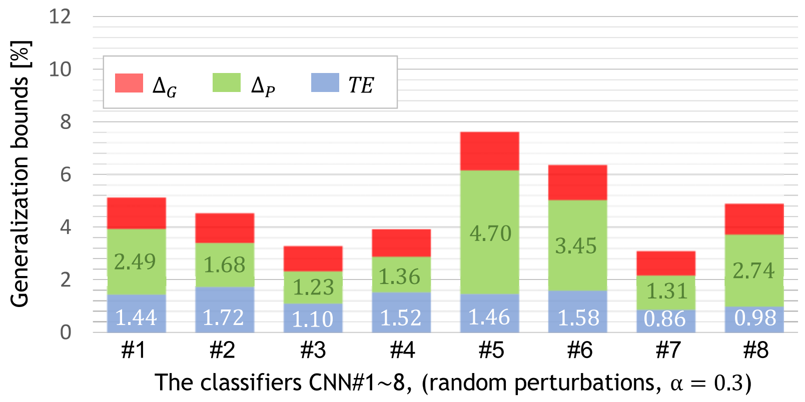


Figure 7.6 The generalization gaps Δ_G and the random-perturbation gaps Δ_P

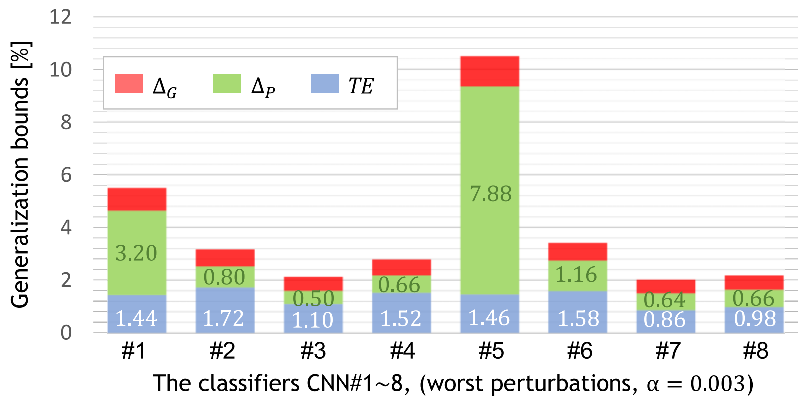
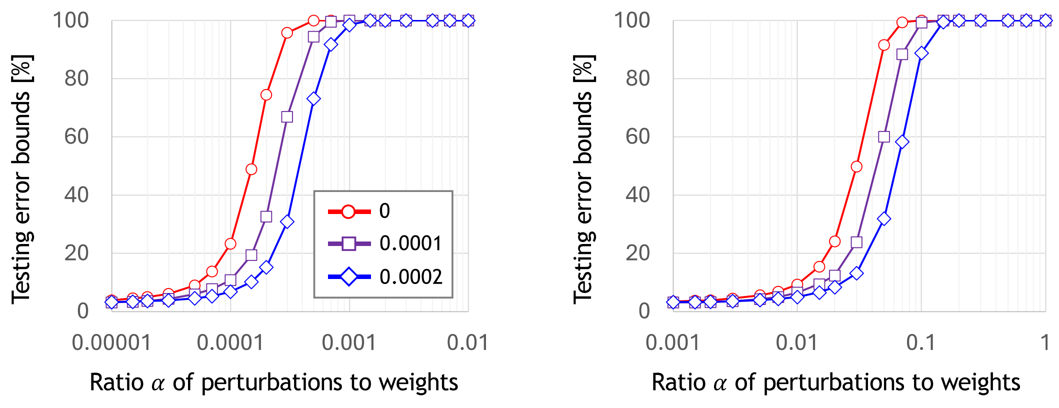


Figure 7.7 The generalization gaps Δ_G and the worst-perturbation gap Δ_P

The program, named *WP-GEB-Estimator*, used in the experiments in this section has been published from the website [71]. The "WP-GEB" in the tool name stands for Weight-Perturbed Generalization Error Bounds.

7.5 Related work

Although we have introduced theorems that guarantee generalization error bounds based on *testing errors* (i.e., by using testing datasets) in this chapter, there are many theorems, e.g., PAC Bayesian theorems [67][73], based on *training errors* (i.e., by using training datasets). The advantage of generalization error bounds based on training errors is that they can be applied to the theoretical study of training (algorithms) for reducing generalization errors. The other advantage is that they can be estimated only by training datasets without additional datasets such as testing datasets. However, it was reported [74] that the estimation results of the generalization error bounds based on training errors are often near 100% (i.e., *vacuous*). Recently, several methods were proposed for estimating *non-vacuous* generalization error bounds (less than 100%) even based on training errors. For example, such methods use distributions of classifiers (i.e., input-output functions instead of weights) in the PAC-Bayes bounds [74], or random labelled data in training [75], or model compression [76]. But it is not easy to practically reduce the generalization error bounds based on training errors. On the other hand, although the generalization error bounds based on testing errors needs testing datasets, they can estimate bounds close to generalization errors. For example, the generalization gaps Δ_G are less than 2% in Figure 7.6 and Figure 7.7. In this chapter, we have focused the generalization error bounds based on testing errors from the perspective of practical evaluation of classifiers.



(a) Based on Theorem 2 in Tsai et al. [77] (b) Based on the expression (7.23)

Figure 7.8 The estimation results of worst weight-perturbed testing errors

There are some existing works on estimating worst weight-perturbed generalization errors. For example, Tsai et al. [77] theoretically analyzed worst weight-perturbed feed-forward neural networks, and presented the formal expressions of pairwise class margin by accumulating maximum errors in each layer (Theorem 2 in [77]). The worst weight-perturbed generalization error bounds can be estimated based on the pairwise class margin with confidence 100%. For example, Figure 7.8 shows the estimation results of the worst weight-perturbed testing errors of 3 classifiers, that are trained small 3-layer feed-forward neural networks by MNIST with different regularizations (L^2-regularization: 0, 0.0001, and 0.0002), and Figure 7.8 (a) is estimated based on Theorem 2 in [77] and (b) is estimated based on the expression (7.23) in this chapter. The estimation based on Theorem 2 in Tsai et al. corresponds the case of the fixed threshold 0% with the confidence 100%. Consequently, Figure 7.8 (a) shows larger testing error bounds than Figure 7.8 (b) that is estimated by the adaptive threshold less than 1% with confidence 95% at least. Although the perturbation-scale is difference between Figure 7.8 (a) and (b), they show similar tendencies about robustness for worst weight-perturbations. The estimation method based on Theorem 2 in [77] is sophisticated, but it should be noted that it strongly depends on the architectures of neural networks.

7.6 Towards the evaluation of "the stability of trained models"

The stability of trained models is one of the 14 internal quality properties described in Machine Learning Quality Management Guideline [1] and it represents that machine-learned components reasonably behave even for unseen input data. In this chapter, we have focused on the randomly/worst weight-perturbed generalization errors based on testing errors, have explained how to estimate them, and have demonstrated the usefulness by experiments. It is expected that such generalization error bounds will be useful for evaluating "the stability of trained models" by the following reasons:

(1) Why are *perturbed* generalization error bounds estimated?
As shown in the experiment results (e.g., see Figure 7.4 and Figure 7.5) in Section 7.4, the potential differences of performance can be clearly observed by adding weight-perturbations. In fact, it has been reported [65][66] that there are high correlations between the generalization performance and the robustness for (especially worst) weight-perturbations. The both of random weight-perturbations and worst weight-perturbations are useful for evaluating classifiers because they often show different tendencies.

(2) Why are perturbations added to *weights* instead of *inputs*?
Although perturbations are often added to inputs (mainly images), it is difficult to apply them to unsorted inputs (e.g., city-names in tabular data). Weight-perturbations can be applicable to neural-classifiers for any type of inputs.

(3) Why are *generalization* error bounds estimated?
Instead of weight-perturbed generalization error bounds, weight-perturbed testing errors ($TE + \Delta_P$ in Figure 7.6 and Figure 7.7) seem to be sufficient for evaluating the classifiers. However, such testing errors cannot guarantee behaviors for unseen data not included in samples. The weight-perturbed generalization error bounds can guarantee that the expected value of misclassification is less than a constant with probability (i.e., confidence) for any input selected from a distribution, and they can be easily estimated from the weight-perturbed testing errors.

(4) Why are generalization error bounds based on *testing* errors used?
Although most of recent research papers on generalization error bounds are based on training errors, the estimation results of such bounds are often vacuous as explained in Section 7.5. Currently, it is thought to be practical for evaluating realistic classifiers

to apply generalization error bounds based on testing errors.

When providing trained classifiers to third parties, it will be helpful for the users to include the estimation results such as Figure 7.4 and Figure 7.5 of the randomly/worst weight-perturbed generalization error bounds in the performance specifications of the classifiers because they can statistically guarantee the upper bounds of misclassifications with probability.

8 Adversarial Example Detection

8.1 Research summary

With the goal of practically establishing a method for determining whether a given input image is an adversarial example, we focus on the following points regarding attacks and detection methods that generate adversarial examples. We are conducting a survey of typical technologies.

- Supporting adversarial example detection program code and confirmation by computational experiment
- Reproduction of experimental results of adversarial example detection method papers
- Implementation of the framework for detecting adversarial examples

Adversarial example detection stands for detecting adversarial examples from given inputs, and existing state-of-the-art adversarial example detection methods can be divided into four main categories.

① Metric based approaches (example [78])
② Denoisers approaches (example [79])
③ Prediction inconsistency based approaches (example [80])
④ Neural Network Invariant Checking (NIC) approaches (example [81])

In this chapter, we report the results of additional test experiments to compare and evaluate adversarial example detection methods based on each of these approaches ① to ④. As reported in the paper [81], it was confirmed that the approach of ④ (NIC: Neural Network Invariant Checking) shows the highest detection rate among ① to ④. In this follow-up experiment, the published implementation code was used for ① to ③, but the implementation code was not published for ④, so a computer experiment was conducted by implementing the NIC according to the paper [81]. Therefore, this chapter mainly describes the NIC ④.

After explaining the outline of the four approaches, the method of detecting adversarial examples by the NIC is explained, and the implementation method is described. Then, the results of the follow-up experiments of each approach and the experiments by the NIC are described. Finally, we report the implementation of the NIC framework and the effectiveness evaluation.

8.2 Overview of adversarial example detection approaches

In this section, the four state-of-the-art approaches to adversarial example detection are overviewed.

8.2.1 Metric based approaches

A method of performing statistical measurements of inputs (and outputs of each neuron) to detect adversarial examples, Ma et al. recently proposed the use of a measurement called Local Intrinsic Dimensionality (LID) [78]. This method estimates the LID value that evaluates the space-filling capacity of the area surrounding the sample by calculating the distance distribution of the sample and the number of neighbors in each layer, and the adversarial example tends to have a large LID value. It uses certain properties to detect adversarial examples. LID is superior to traditional kernel density estimation (KD) and Bayesian uncertainty (BU) for detecting adversarial examples and is currently the state-of-the-art technology for this type of detector.

8.2.2 Denoisers approaches

It is a method of detecting adversarial examples by removing noise in a preprocessing step for each input. In this method, the training model or noise remover (encoder and decoder) is trained to filter the image so that the key components in the training model can be highlighted. This filter can be used to remove noise added by an attacker to generate adversarial examples and correct misclassification. MagNet [79] is a method of detecting adversarial examples using detectors and reformers (trained automatic encoders and automatic decoders).

8.2.3 Prediction inconsistency based approach

A method of detecting adversarial examples by measuring the discrepancy between the original neural network and the neural network enhanced by human perceptible attributes. Feature Squeezing [80], the state-of-the-art detection technique of this method, can achieve very high detection rates against a variety of attacks. Feature squeezing focuses on detecting gradient-based attacks, focusing on the ability of attackers to generate adversarial examples through the unnecessarily large input feature space of deep neural networks DNN. The procedure for detecting adversarial examples by feature squeezing is shown below.

1. Apply squeezing technology (a technology that reduces the color depth of an image and smooths the image) to the original input image to generate multiple squeezed images.
2. Input the original input image and multiple squeeze images into the deep neural network, and measure the distance between the inference result (prediction vector) of the input image and the inference result of each squeeze image.
3. When one of the differences (distances) between the original input image and the squeeze image exceeds the threshold value, the original input image is detected as an adversarial example.

8.2.4 Neural Network Invariant Checking (NIC) approaches

The NIC (Neural Network Invariant Checking) method focuses on value invariants (VIs) and provenance invariants (PIs) inside deep neural networks [81]. The value invariant VI is the distribution of possible neuron values in each layer, and the provenance invariant PI is the possible neuron value pattern of two consecutive layers (summary of correlation between features across two layers). If an input violates these invariants, the input is detected as an adversarial example. The NIC [81] method trains these invariant VIs and PIs with benign input data and model them as a one-class classification (OCC) problem that detects adversarial examples. A higher detection rate has been reported than the methods based on (1) to (3) explained above. The outline and the implementation of the NIC system design are explained in detail in Sections 8.3 and 8.4, respectively.

8.3 NIC system design overview

The procedure for building the NIC detector (steps A to C: during training, D to E: during execution) is explained by using Figure 8.1 [81]. This invariant VI, PI training uses only non-adversarial benign data.

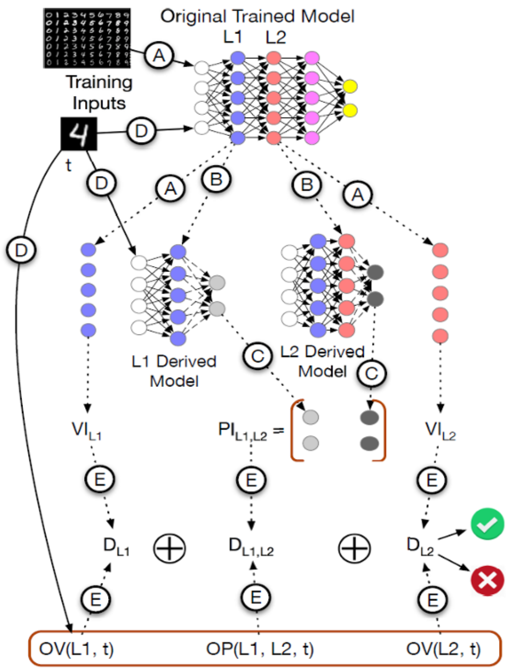


Figure 8.1 Outline of system design (Fig. 8 of thesis [81])

- **Step A**: Collect the output value of each neuron at each layer of each training data input.
- **Step B**: For each layer k (e.g., L1, L2), extract the sub-models from the input layer to the k layer and add a new softmax layer with the same output label as the original model. Then create a derived model (DerivedModel in Figure 8.1)

- **Step C**: Enter each benign training data for all derived models and collect the final output of these models (i.e., the output probability values of the individual classes). For each set of consecutive layers, we train using the distribution of the classification results of this derivative model. This trained distribution is the PI for these two layers.

- **Step D**: Input each test data t (for example, the image of "4" in Figure 8.1) to all derivative models in addition to the original model, and observe the activation value of each layer of the original model. Collect the value OV (for example, OV(L1, t) in Figure 8.1) and the classification result (set) of the derivative model of consecutive layers. From this classification result, the observed source OP (for example, OP(L1, L2, t), etc.) is obtained.

- **Step E**: Calculate the probability D that the OV and OP fit the corresponding VI and PI distributions. The possibility that the input t is adversarial is predicted at the same time by aggregating all these D values.

8.4 NIC system implementation

In order to detect adversarial examples based on NIC, a direct sum space (vector) is constructed from PI and VI, and for classifying this vector, an OSVM (One Class Support Vector Machine) is constructed. When the input to the layer l of the trained DNN (Deep Neural Network) model (hereinafter referred to as M) is x_l, the output f_l of the layer l is given by the following equation:

$$f_l = \sigma(x_l \cdot w_l^T + b_l),$$

where σ is the activation function of the layer l, w_l^T is the weight matrix, and b_l is the bias. At this time, the direct sum spaces classified by VI, PI, and OSVM are obtained as follows.

- VI calculation: The VI of each layer l of model M is determined by solving the following optimization problem.

$$VI_l = \min\left[\sum_{x\in X_b} J(f_l \circ f_{l-1} \circ \cdots \circ f_1(x) \cdots w^T - 1)\right]$$

  Here, J is the error evaluation function, and X_b is the batch used to create M. Also, $\circ$ is a monoid, in this case a vectorized version of f_k.

- PI calculation: $PI_{l,l+1}(x)$ is based on the classification output of the derived models of the layers l and $l+1$. The probability that x is benign (non-adversarial) is estimated by solving the following optimization problem.

$$PI_{l,l+1}(x) = \min\left[\sum_{x\in X_b} J\big(concat\big(D_l(x), D_{l+1}(x)\big) \cdots w^T - 1\big)\right]$$

Here, a derivative model D_l of the layer l is defined as follows, with the softmax layer added after the layer l.

$$D_l = \mathrm{softmax} \circ f_l \circ f_{l-1} \circ \cdots \circ f_1$$

– Direct sum space of PI and VI: From the VI and PI obtained by the above optimization, the following direct sum space (vector) is created for each batch of training data of model M.

$$VI_1 \oplus PI_{1,2} \oplus VI_2 \oplus PI_{2,3} \cdots VI_B \oplus PI_{B-1,B} \oplus VI_B$$

This vector is $L \times 3$ dimensions (L is the number of layers of M), which is the vector space (direct sum space) of the number B. The NIC performs OSVM on this space.

8.5 Computer experiment

In order to confirm the effect of adversarial example detection technology (NIC), the experiment of the paper [81] was retested in the following experimental environment.

– Hardware environment: AIST ABCI [82]
– Datasets: Two common image datasets, MNIST [83] and CIFAR-10 [84], were used for image classification experiments. MNIST is a grayscale image dataset used for handwritten digit recognition, and CIFAR-10 is a color image dataset used for object recognition. For NIC, we also conducted an experiment on LFW (face image) [85].
– Attacks: Non-targeted attacks (FGSM L^2,L^∞), targeted attacks JSMA, and gradient-based attacks (CW L^2) were used to generate adversarial examples. The Cleverhans library [86] was used to implement FGSM and JSMA

First, in order to evaluate the adversarial example detection method based on each of the approaches ① to ③, the published implementation code of LID [78], MagNet [79], and feature squeezing [80] was used to evaluate each paper. Then, follow-up experiments were conducted. As the result, the detection rates reported in each paper were able to be confirmed, and among these three, feature squeezing showed the highest detection rate.

Next, in order to evaluate the adversarial example detection method based on the approach ④, an experiment was conducted using the NIC code implemented in Section 8.4. Table 8.1 to Table 8.3 show the results of adversarial example detection and computational experiments on the MNIST, CIFAR-10, and LFW datasets, respectively. Here, the correct answer rate is the rate at which adversarial examples are input to the classifier (OSVM) described in Section 7.4 and are determined to be adversarial examples. The CNN model used in the experiment is LeNet5, and the OSVM Kernel is RBF (MNIST: γ = 0.1 to 0.27, CIFAR-10: γ = 0.11 to 0.2, LFW: γ = 0.005 to 0.90). In the results of this experiment, high detection performance was confirmed not only for the dataset and attack method reported in the paper [81], but also for the unreported dataset LFW and attack method (FGSM L^∞).

Table 8.1 Adversarial example detection computational experiment results for MNIST dataset

Data Set	Attack	Invariant	Performance	Number of data	Performance reported in the paper [81]
MNIST	FGSM L^2	VI	97%	2800	100%
		PI	98%		84%
		NIC	97%		100%
	FGSM L^∞	VI	98%	2800	—
		PI	98%		—
		NIC	98%		—
	JSMA	VI	100%	280	83%
		PI	100%		100%
		NIC	100%		100%
	CW2	VI	100%	280	95%
		PI	100%		96%
		NIC	100%		100%
	Trojan	VI	100%	3200	100%
		PI	100%		100%
		NIC	100%		100%

Table 8.2 Adversarial example detection computational experimental results for CIFAR-10 dataset

Data Set	Attack	Invariant	Performance	Number of data	Performance reported in the paper [81]
CIFAR-10	FGSM L^2	VI	99%	6400	100%
		PI	99%		52%
		NIC	99%		100%
	FGSM L^∞	VI	100%	6400	—
		PI	100%		—
		NIC	100%		—
	JSMA	VI	97%	320	62%
		PI	95%		100%
		NIC	96%		100%
	CW2	VI	98%	320	88%
		PI	95%		89%
		NIC	96%		100%
	Trojan	VI	100%	3200	100%
		PI	100%		100%
		NIC	100%		100%

Table 8.3 Adversarial example detection computational experiment results for LFW dataset

Data Set	Attack	Invariant	Performance	Number of data	Performance reported in the paper [81]
LFW	FGSM L^2	VI	98%	28222	—
		PI	98%		—
		NIC	98%		—
	FGSM L^∞	VI	100%	2822	—
		PI	100%		—
		NIC	100%		—
	JSMA	VI	100%	280	—
		PI	100%		—
		NIC	100%		—
	CW2	VI	100%	840	—
		PI	100%		—
		NIC	100%		—
	Trojan	VI	100%	3200	—
		PI	100%		—
		NIC	100%		—

8.6 Implementation of the NIC framework

We have implemented a simplified NIC method based on Sections 8.3 and 8.4 in order to conduct the computer experiments for confirming the effectiveness of NIC in Section 8.5. In the simplified implementation, we have found some implementation issues in the original paper [81]. In this section, while clarifying the issues, we reconsider the algorithm in order to construct the *NIC framework* for high detection rates of adversarial examples on the testbed, that is used for creating an environment (attack, defense and detection) to benchmark vulnerability to adversarial examples.

8.6.1 Overview of the NIC framework

The NIC framework consists of five parts: taking output from each layer; calculating VI and PI for normal data; calculating VI, PI and NIC for adversarial examples; evaluating OSVM and displaying results. The use case of the NIC framework is shown in Figure 8.2. In addition, the process steps for detecting adversarial examples are shown in Figure 8.3.

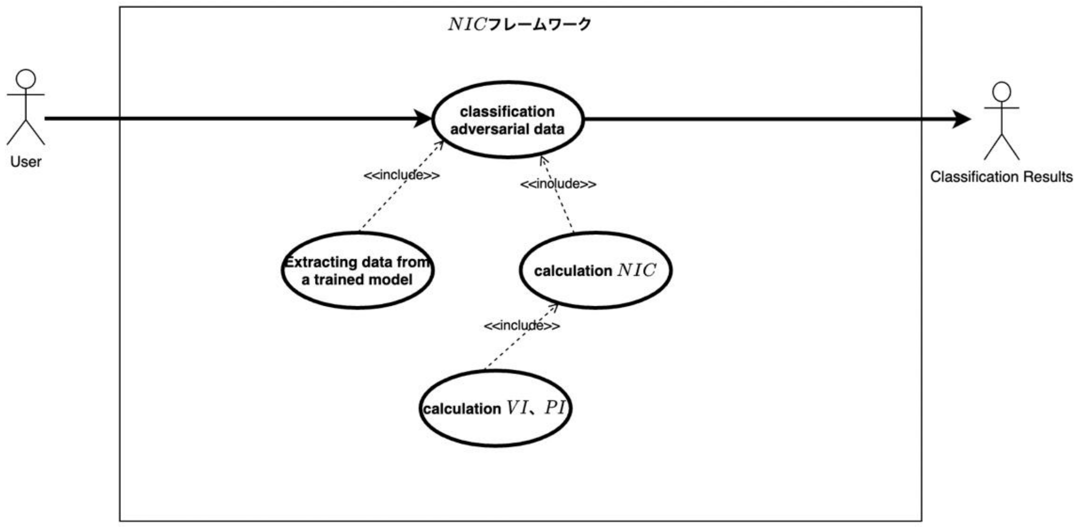


Figure 8.2 NIC framework use cases

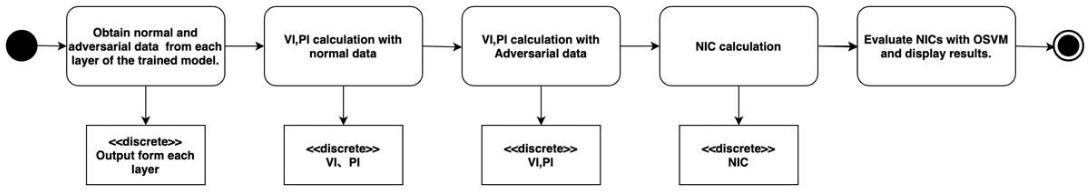

Obtain normal and adversarial data from each layer of the trained model.
VI,PI calculation with normal data
VI,PI calculation with Adversarial data
NIC calculation
Evaluate NICs with OSVM and display results.
<<discrete>> Output form each layer
<<discrete>> VI、PI
<<discrete>> VI,PI
<<discrete>> NIC

Figure 8.3 Processing procedures for adversarial example detection by the NIC framework

As shown in Figure 8.3, the overall processing procedure for adversarial example detection by the NIC framework consists of five parts. The function of each part (input, processing and output) is shown in Table 8.4.

8.6.2 Output of OSVM evaluation results

The NIC framework has been implemented using scikit-learn, that is a Python machine learning library. For example, the scikit-learn's OneClassSVM class is used for implementing the final part of the OSVM as shown in Figure 8.3 as follows.

```
class sklearn.svm.OneClassSVM(array, kernel='rbf', gamma='auto', nu=0.3)
```

Here, the meaning of each argument is as follows.

- array: parameters trained by normal data and used for detecting adversarial examples in NIC.
- kernel: the RBF kernel is used as the algorithm for One Class SVM.
- gamma: the gamma parameter of the RBF kernel is set to 'auto'.
- nu: the upper limit for the percentage of training error and the lower limit for the percentage of support vector are set to 0.3 in this case.

Table 8.4 Functions of the parts comprising the adversarial example detection process procedure

Output extraction from each layer	
input	Normal data (images)
	Adversarial examples (image).
	Trained models, trained on normal data (models trained on normal data)
processing	Obtain the output of each layer of the trained model for normal and adversarial examples and save it in 'numpy in numpy' format.
output (e.g. of dynamo)	Output data from each layer

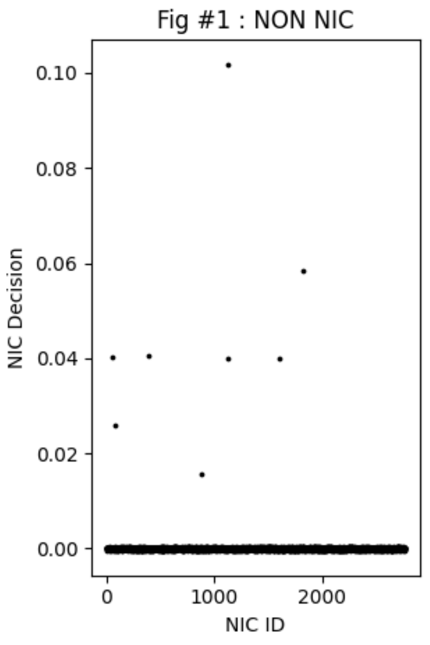
VI and PI calculations for normal data	
input	Output data from each layer of normal data
processing	Calculate VI, PI from the output data of each layer of normal data.
output (e.g. of dynamo)	VI, PI

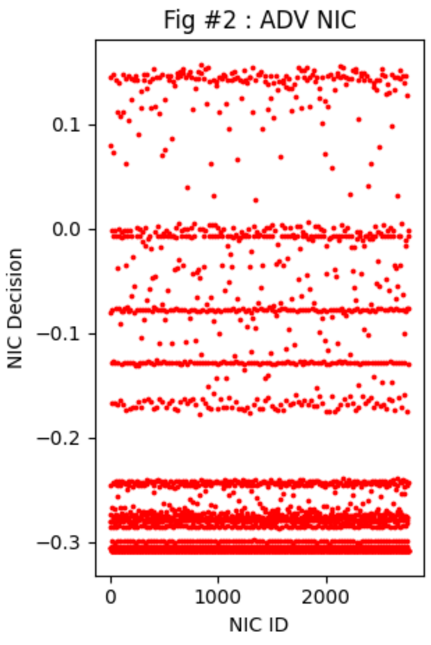
VI and PI calculations for adversarial examples	
input	Output data from each layer of adversarial examples
	Created at the time of calculation to PI with normal data Derived model of PI
processing	Compute VI, PI from the output of each layer of adversarial examples.
output (e.g. of dynamo)	VI, PI

Calculation of NIC	
input	VI of normal data, PI
	VI of adversarial examples, PI
processing	NIC of normal data is created from VI and PI of normal data and NIC of adversarial examples is calculated from VI and PI of adversarial examples, respectively.
output (e.g. of dynamo)	NIC for normal data, NIC for adversarial examples

Evaluation and display of results in OSVM.	
input	NIC of normal data
	NIC for adversarial examples.
processing	Train OSVM on normal data to create a model, and use this trained model to judge adversarial examples; OSVM uses sk-learn's one class svm API. The judgement results are then displayed.
output (e.g. of dynamo)	Assessment Results

Figure 8.4 shows output values from each layer when one normal data and its adversarial examples are input to the NIC framework, where the horizontal axis is the ID of the model derived to calculate the NIC at each layer (Note. There are multiple outputs from each layer for one image, for example, a convolution layer in CNN), and the vertical axis represents the signed distance of each NIC to the One Class SVM classification hyperplane of the NIC of the normal data, that is the closeness to the normal data in this case. The black dots in Figure 8.4 (a) represent the output relative to the normal data, the red dots in Figure 8.4 (b) are the outputs for adversarial examples. In this calculation, the adversarial examples in Figure 8.4 (b) were generated by using the FGSM L^{∞} attack method.





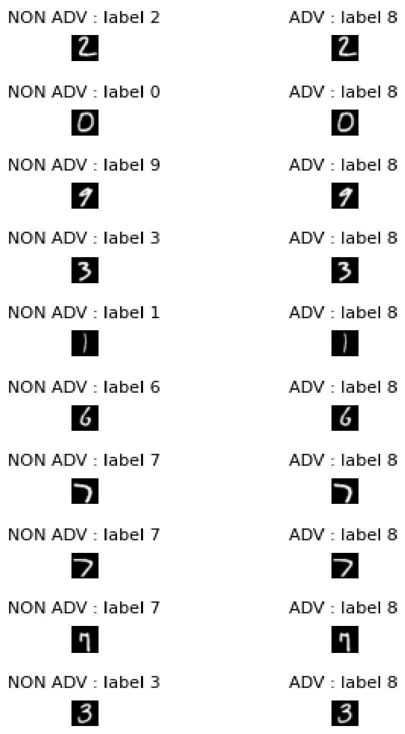
(a) Normal input data (b) Adversarial input data

Figure 8.4 Comparison of NIC framework outputs

After training One Class SVM by normal data, One Class SVM function $f(x)$ can be used for detecting adversarial examples such that if $f(x) \geq 0$ then the input x is normal otherwise it is adversarial. Most of the output for normal data are close to zero as shown in Figure 8.4 (a), while approximately 94% of the outputs for adversarial examples are explicitly less than zero as shown in Figure 8.4 (b). This difference of the output between Figure 8.4 (a) and (b) explains that NIC can effectively detect adversarial examples.

8.6.3 Generation of adversarial examples

As shown in Figure 8.3, NIC framework does not include the program for generating adversarial examples. We recommend for using CleverHans [87] if adversarial examples are necessary. Figure 8.5 shows some examples in the normal (original) images of handwritten numbers (MNIST) and the adversarial examples generated from the normal images by attack method FGSM L^2 with the misclassified labels inferred for the adversarial images. As shown in the inference results (label 8) in Figure 8.5 (b), all generated adversarial examples are misclassified as 8.



(a) Original MNIST data (b) Generated adversarial examples

Figure 8.5 Example of adversarial example generation from MNIST (handwritten numbers) images and its decision results

8.6.4 Reducing calculation costs for VI, PI and VIC

The calculation method for VI, PI and NIC in the original paper [81] has been explained in Section 8.4, but if the calculation method is used, then the dimension of each data (vector) becomes very large, due to the problem so-called 'dimension demon'. Therefore, we have tried to reduce the dimension as much as possible. In the following section, we explain how each calculation is simplified.

- **Calculation of VI**: in the NIC framework, let $X_B = 1$ for clarifying the correspondence between the input data (both normal and adversarial data) and the VI, PI and NIC (i.e., for the accuracy of the verification). In addition, as all input data are normalized and calculated, the following simplified formula is used:
$$VI_l = f_l \circ f_{l-1} \circ \cdots \circ f_2 \circ f_1.$$
- **Calculation of PI**: as in the case VI above, let $X_B = 1$. Then, the following simplified

formula is used:

$$PL_{l,l-1} = concat(D_l, D_{l-1}) \circ \cdots \circ concat(D_2, D_1).$$

– **NIC calculations**: for dimensionality suppression, X_B is set as follows:

$$X_B = (\text{The number of layers from which output are obtained})$$

8.7 Evaluation of the effectiveness of NIC with the Kullback-Leibler divergence

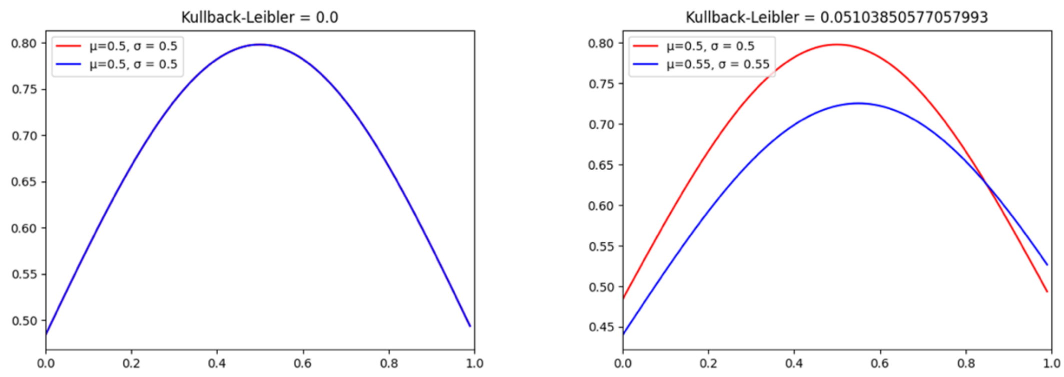
This section reports the results of the evaluation of the effectiveness of the NIC by calculating the degree of divergence between the images of normal and adversarial examples and the NIC by using the Kullback-Leibler divergence.

8.7.1 Kullback-Leibler divergence

The Kullback-Leibler divergence, denoted by $KL(P \parallel Q)$, is a measure of the degree of divergence between two probability distributions P (the probability density functions p) and Q (the probability density function q). The Kullback-Leibler divergence is defined by the following equation.

$$KL(P \parallel Q) = \int p(x) \log \frac{p(x)}{q(x)}$$

The Kullback-Leibler divergence is 0 when the two distributions are the same, and it increases as the divergence increases (the convergence is not guaranteed due to the presence of log). Figure 8.6 shows a simple calculation example of the Kullback-Leibler divergence. In Figure 8.6 (a), both of the distributions P and Q are the same normal distribution whose mean and variance are 0.5 and 0.5, respectively, and then the $KL(P \parallel Q)$ is 0. In Figure 8.6 (b), the means of P and Q are 0.5 and 0.55, and the variance of them are 0.5 and 0.55, respectively, and then the $KL(P \parallel Q)$ is 0.053.



(a) In the case of the same distributions (b) In the case of the different distributions

Figure 8.6 Example of Kullback-Leibler divergence calculation

8.7.2 **Kullback-Leibler divergence estimation**

The Kullback-Leibler divergence assumes that the probability distributions to be compared are fixed, but in practice, both normal and adversarial data are simply sets of images and the distributions are unknown. Fortunately, a method for approximating the Kullback-Leibler divergence between sets with unknown probability distributions [88] is known. The outline of the approximation method calculates the Kullback-Leibler divergence as a solution of an optimization problem on the following linear polynomial of $r_\theta(x)$ as the constraint for minimizing the density ratio $r(x) = p(x)/q(x)$:

$$r_\theta(x) = \sum_{j=1}^{b} \theta_j \psi_j(x) = \boldsymbol{\theta}^T \boldsymbol{\psi}(x),$$

where $\psi_j(x)$ is the RBF kernel and is defined by

$$\psi_j(x) = \exp\left(-\frac{\|x - x'\|^2}{2h^2}\right),$$

where h is a determinable constant and is the bandwidth.

Then, the Kullback-Leibler divergence can be approximately calculated by the linear polynomial $r_\theta(x)$ obtained as the solution of the optimization problem as follows [88]:

$$KL(P \parallel Q) \sim \frac{1}{n}\sum_{i=1}^{n} \log r(\boldsymbol{x}_i)$$

8.7.3 **Effectiveness evaluation of NIC**

In Section 8.5, we have shown that the NIC method can effectively detect adversarial examples as anomaly data by experiments. In this section, we show the degree of divergence between normal data and adversarial examples by comparing the Kullback-Leibler divergence of them for explaining the reason why NIC is effective.

At first, Figure 8.7 shows the computational results of the Kullback-Leibler of normal data and adversarial examples (generated by the attack method FGSM L^2) for 50 image data samples, as shown in Figure 8.5. The approximate value of the Kullback-Leibler divergence for the FGSM in Figure 8.7 is 0.46. Here, note that the average value of the multiple Kullback-Leibler divergence is shown in Figure 8.7 because there are multiple values of NIC for one image as explained in Figure 8.4.

Next, Figure 8.8 shows the computational results of the Kullback-Leibler of NIC of the normal data and the adversarial examples used in Figure 8.7. The approximate value of the Kullback-Leibler divergence in Figure 8.8 is 4.47. Therefore, the Kullback-Leibler divergence in Figure 8.8 is about 10 times larger than one in Figure 8.7. We conjecture that the results mean that the perturbations added to normal data can be extracted as more explicit difference by the NIC method.

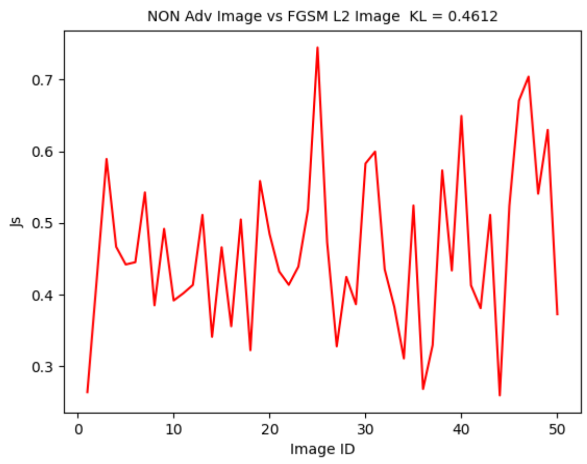


Figure 8.7 The Kullback-Leibler divergence for normal and adversarial examples

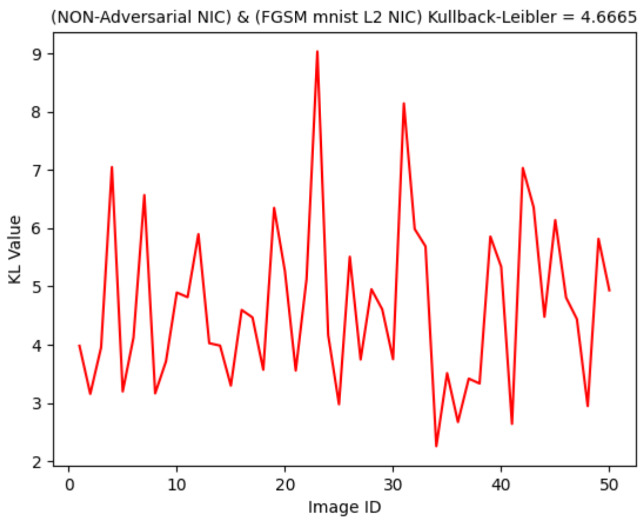


Figure 8.8 The Kullback-Leibler divergence of NIC for normal and adversarial examples

9 AI Quality Management in Operation

In this chapter, we report on the results of a survey on the latest technologies for detecting changes in data distribution over time, called concept drift, and adapting machine learning models to the changed distribution for AI quality management during operation. In addition, we also introduce the results of a survey on the latest unsupervised domain adaptation technologies published at recent international major conferences on machine learning and computer vision for further development of the AI quality management technologies.

Concept drift is one of the main causes of performance degradation of machine learning models running in AI systems during operation. In order to maintain quality that is satisfied at the beginning of the operation of the system throughout the operation period, it is necessary to continuously monitor whether drift occurs or not. In addition, if necessary, we retrain the machine learning models in the system with the latest data to adapt them to the distribution of data changed after the drift occurs. As the use of machine learning technologies has been expanded in recent years, AI systems operating with such technologies will require processing a large amount of data without their true labels (ground truths) in a short period of time, including types of data that have not been handled in the past.

In the fiscal year 2019-2020, we conducted a survey on the latest technologies for detecting and adapting to the concept drift to maintain the performance of machine learning models during operation. As a result of this survey, we found that most of the methods developed so far are supervised methods that use true labels of data additionally acquired during operation for the detection and adaptation. However, such true labels are not always available or are often costly even if they are available. In order to expand the applicability of the detection and adaptation methods and reduce their operational costs, we found that an "unsupervised method" that does not use the true labels or a "semi-supervised method" that uses only a limited number of the true labels is promising. We summarized the results of the surveys organized and discussed from this perspective.

For details on the survey on detection methods, see Section 7.8 of the Machine Learning Quality Management Guidelines [1]. In addition, adaptation methods are summarized in our survey result [89]. Table 9.1 shows the comparison of our survey with the other existing surveys on concept drift detection and adaptation methods. Gama et al. summarized their survey result in [90] and Lu et al. added recently published drift detection and adaptation methods in [91]. Those survey papers mainly focus on introducing "supervised" methods that use true labels of operational data for drift detection and adaptation. On the other hand, Ishida et al. introduced "unsupervised" concept drift detection methods that do not use true labels of data for drift detection in [92]. In comparison with those existing survey results, we introduced "unsupervised" and "semi-supervised" concept drift adaptation methods that do not use or use only a limited number of true labels as mentioned above. Furthermore, we introduced those drift adaptation methods based on the characteristic of each method. In detail, we listed ten remarkable unsupervised/semi-supervised drift adaptation methods and classified them according to: i) types of drift that can be dealt with effectively, ii) processes where true labels of data are required during operation and the percentage of the labeled data used in verifications shown in the papers, and iii) machine learning models or clustering methods used in each method. Finally, we closed our survey by discussing further development of unsupervised and semi-supervised concept drift adaptation methods using knowledge obtained from relevant unsupervised domain adaptation techniques.

Table 9.1 Comparison of survey papers on concept drift detection and adaptation

	Detection	**Adaptation**
Supervised	Gama et al.[90], Lu et al.[91]	
Unsupervised / Semi-supervised	Ishida et al.[92]	Okawa and Kobayashi [89], [93] (Ours)

In the future operation of AI systems, there is a growing need for new adaptation techniques that do not use the original training data (i.e., source data) to adapt machine learning models from the viewpoint of data privacy and portability in addition to that can deal with changes other than those in the distribution of input data. In particular, adaptation techniques that do not depend on such training data (source data) are called "source-free domain adaptation techniques" or "test-time adaptation techniques (if they adapt online)”. These source-free and test-time adaptation technologies have been attracting more attention because they can reduce costs not only on management and transmission of source data for adaptation but also on security for data storage.

In FY2021, following the above-mentioned surveys, we conducted a survey on the latest research trends in unsupervised adaptation techniques to data changes presented at major international conferences in the fields of machine learning and computer vision held in 2019-2021, focusing on unsupervised concept drift adaptation techniques and unsupervised domain adaptation techniques. The result of this survey is summarized in [93]. In detail, we listed and introduced 15 remarkable concept drift detection and unsupervised domain adaptation methods and classified them according to: i) kinds of adaptation problems, ii) kinds of data and labels used in detection and adaptation, iii) availability for adaptation to label shift, and iv) kinds of validation tasks. According to the results of this survey, it is shown that there has been development of the source-free adaptation and test-time adaptation techniques mentioned above and adaptation techniques that are able to adapt to changes other than the distribution of input data, such as label shifts. Furthermore, some techniques have been validated not only for image classification problems, but also for semantic segmentation and object detection problems. These research trends in unsupervised adaptation techniques are expected to solve new problems in AI operations, such as maintaining data privacy, and to be used in various situations in future AI operations.

10 References

Chapter 1:

[1] National Institute of Advanced Industrial Science and Technology (AIST), Machine Learning Quality Management Guideline (4th English Edition), Digital Architecture Research Center, Cyber Physical Security Research Center, Artificial Intelligence Research Center. The 4th English Edition will be published soon, while the 3rd English Edition is currently available as Technical Report DigiARC-TR-2023-01/ CPSEC-TR-2023001, 2023. https://www.digiarc.aist.go.jp/en/publication/aiqm/

[2] Yuri Miyagi, Masaki Onishi, Machine Learning Model Comparison Visualization Focusing on Worker Information, The 24th Meeting on Image Recognition and Understanding 2021, I31-22, 2021.

[3] Yuri Miyagi, Masaki Onishi, Comparative Visualization Method Focusing on Workers for Evaluation of Machine Learning Models, The 49th Symposium on Visualization, OS12, 2021.

[4] Tomoumi Takase, Dynamic batch size tuning based on stopping criterion for neural network training, Neurocomputing, Volume 429, pp.1-11, 2021.

[5] Shin Nakajima, Software Testing with Statistical Partial Oracles, 10th SOFL+MSVL, 2021.

Chapter 2:

[6] Satoshi Hara, My Bookmark : Interpretability in Machine Learning, Journal of Japanese Society for Artificial Intelligence, vol. 33, no. 3, pp. 366-369, 2018 (in Japanese).

[7] Fred Hohman, Minsuk Kahng, Robert Pienta, Duen Horng Chau, Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers, IEEE Transactions on Visualization and Computer Graphics, vol. 25, no. 8, pp. 2674-2693, 2018.

[8] Bilal Alsallakh, Amin Jourabloo, Mao Ye, Xiaoming Liu, Liu Ren, Do Convolutional Neural Networks Learn Class Hierarchy?, IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 1, pp. 152-162, 2018.

[9] Mengchen Liu, Jiaxin Shi, Kelei Cao, Jun Zhu, Shixia Liu, Analyzing the Training Processes of Deep Generative Models, IEEE Transactions on Visualization and Computer Graphics, vol.24, no.1, pp.77-87, 2018.

[10] Jorge Piazentin Ono, Sonia Castelo, Roque Lopez, Enrico Bertini, Juliana Freire, Claudio Silva, PipelineProfiler: A Visual Analytics Tool for the Exploration of AutoML Pipelines, IEEE Transactions on Visualization and Computer Graphics, vol.27, no.2, pp.390-400, 2021.

[11] Saleema Amershi, Maya Cakmak, W. Bradley Knox, Todd Kulesza, Power to the People: The Role of Humans in Interactive Machine Learning. AI Magazine, vol.35, no.4, pp.105-120, 2014.

[12] Heungseok Park, Jinwoong Kim, Minkyu Kim, Ji-Hoon Kim, Jaegul Choo, Jung-Woo Ha and Nako Sung, VISUALHYPERTUNER: VISUAL ANALYTICS FOR USER-DRIVEN

HYPERPARAMTER TUNING OF DEEP NEURAL NETWORKS, 2019.

Chapter 3:

[13] Cubuk, E. D., Dyer, E. S., Lopes, R. G., and Smullin, S., Tradeoffs in Data Augmentation: An Empirical Study. In Proceedings of International Conference on Learning Representations, 2021.

[14] Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q., RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In Neural Information Processing Systems, 33, 2020.

[15] Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D., Mixup: Beyond Empirical Risk Minimization. In International Conference on Learning Representations, 2018.

[16] Takase, T., Feature Combination Mixup: Novel Mixup Method Using Feature Combination for Neural Networks, Neural Computing and Applications, 2023.

[17] Verma, V., Lamb, A., Beckham, C., Najafi, A., Mitliagkas, I., Lopez-Paz, D., and Bengio, Y., Manifold Mixup: Better Representations by Interpolating Hidden States. In International Conference on Machine Learning, pp. 6438–6447, PMLR, 2019.

[18] Kim, J-H., Choo, W., and Song, H. O., Puzzle mix: Exploiting saliency and local statistics for optimal mixup. In International Conference on Machine Learning, 2020.

[19] Beckham, C., Honari, S., Verma, V., Lamb, A., Ghadiri, F., Hjelm, R. D., Bengio, Y., and Pal, C. On adversarial mixup resynthesis. In Neural Information Processing Systems, 2019.

[20] Yun, S., Han, D., Oh, S. J., Chun, S., Choe, J., and Yoo, Y. CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features. In Proceedings of the IEEE/CVF International Conference on Computer Vision, 6023–6032, 2019.

[21] Lopes, R. G., Yin, D., Poole, B., Gilmer, J., and Cubuk, E. D., Improving Robustness Without Sacrificing Accuracy with Patch Gaussian Augmentation. arXiv preprint arXiv: 1906.02611, 2019.

Chapter 4:

[22] Nakajima, S., Quality Issues in Machine Learning from Software Engineering Viewpoints, Maruzen Publisher, 2020. (in Japanese)

[23] Pei, K., et al., DeepXplore: Automated Whitebox Testing of Deep Learning Systems, In Proc. 26th SOSP, 2017, pp.1-18.

[24] Nakajima, S., Distortion and Faults in Machine Learning Software, In Post-Proc. 9th SOFL+MSVL, 2020, pp.29-41.

[25] Ma, L., et al., DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems, In Proc. ASE, 2018, pp.120-131.

[26] Tian, Y., et al., DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars, In Proc. 40th ICSE, 2018, pp.303-314.

[27] Zhang, M., et al., DeepRoad: GAN-Based Metamorphic Testing and Input Validation

Framework for Autonomous Driving Systems, In Proc. ASE, 2018, pp.132-142.
[28] Zhang, P, et al., CAGFuzz: Coverage-Guided Adversarial Generative Fussing Testing of Deep Learning Systems, arXiv:1911.07931, 2019.
[29] Harel-Canada, F., et al., Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks? In ESEC/FSE, 2020, pp.851-862.
[30] Kim, J. et al., Guiding Deep Learning System Testing Using Surprise Adequacy, In Proc. 41st ICSE, 2019, pp.1039-1049.

Chapter 5:

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, The MIT Press 2016.
[32] Simon Haykin, *Neural Networks and Learning Machines (3ed.)*, Pearson India 2016.
[33] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Anath Grama, MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection, In Proc. 26th ESE/FSE, pp.175-186, 2018.
[34] Shin Nakajima, Software Testing with Statistical Partial Oracles – Applications to Neural Network Software, In Proc. 10th SOFL+MSVL, pp.275-192, 2021.
[35] Shin Nakajima and Tsong Yueh Chen, Generating Biased Dataset for Metamorphic Testing of Machine Learning Programs, In Proc. 31st ICTSS, pp.56-64, 2019.
[36] Gregor Montavon, Genevieve B. Orr, and Klaus-Robert Muller (eds.), *Neural Networks: Tricks of the Trade (2ed.)*, Springer 2012.
[37] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov, Membership Inference Attacks Against Machine Learning Models, arXiv:1610.05820v2, 2017.
[38] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha, Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting, arXiv:1709.01604v5, 2018.
[39] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen, Understanding Membership Inferences on Well-Generalized Learning Models, arXiv:1802.04489, 2018.
[40] Charu C. Aggarwal, *Outlier Analysis (2ed.)*, Springer 2017.
[41] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer, Replux: An Efficeint SMT Solver for Verifying Deep Neural Networks, In Proc. 29th CAV, pp.97-117, 2017.
[42] Pang Wei Koh and Percy Liang, Understanding Black-box Predictions via Influence Functions, arXiv:1703.04730v3, 2020.
[43] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana, DeepXplore: Automated Whitebox Testing of Deep Learning Systems, In Proc. 26th SOSP, pp.1-18, 2017.
[44] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems, In Proc. 33rd ASE, pp.120-131, 2018.
[45] Yizhen Dong, Peixin Zhang, Jingyi Wang, Shuang Liu, Jun Sun, Jianye Hao, Xinyu Wang, Li Wang, Jin Song Dong, and Dai Ting. There is Limited Correlation between Coverage and Robustness for Deep Neural Networks. arXiv:1911.05904, 2019.

[46] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, and Miryung Kim, In Proc. 28th ESEC/FSE, pp.851-862, 2020.

[47] Shin Nakajima, Distortion and Faults in Machine Learning Software, In Proc. 9th SOFL+MSVL, pp.29-41, 2019.

[48] Stephanie Abrecht, Maram Akila, Sujan Sai Gannamaneni, Konrad Groh, Christian Heinzemann, Sebastian Houben, and Matthjas Woehrle, Revisiting Neuron Coverage and Its Application to Test Generation, In Proc. SAFECOMP 2020 Workshop, pp.289-301, 2020.

[49] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang, Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source, In Proc. 44th ICSE, pp.995-1007, 2022.

[50] Md Johirul Islam, Giang Nguyen, Rangeet Plan, and Hridesh Rajan, A Comprehensive Study on Deep Learning Bug Characteristics, In Proc. 27th ESEC/FSE, pp.510-520, 2019.

[51] Jiakun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung, DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs, In Proc. 44th ICSE, pp.573-585, 2022.

[52] Yanhui Li, Linghan Meng, Lin Chen, Li Yu, Di Wu, Yuming Zhou and Baowen Xu, Training Data Debugging for the Fairness of Machine Learning Software, In Proc. 44th ICSE, pp.2215-2227, 2022.

Chapter 6:

[53] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus, Intriguing properties of neural networks, The International Conference on Learning Representations (ICLR 2014), pp.1-10, 2014. https://arxiv.org/abs/1312.6199

[54] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer, Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks, International Conference on Computer-Aided Verification (CAV), 2017. https://arxiv.org/abs/1702.01135

[55] Vincent Tjeng, Kai Xiao, and Russ Tedrake, Evaluating robustness of neural networks with mixed integer programming, International Conference on Learning Representations (ICLR), 2019. https://arxiv.org/abs/1711.07356

[56] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S. Dhillon, and Luca Daniel, Towards Fast Computation of Certified Robustness for ReLU Networks, International Conference on Machine Learning, PMLR 80, pp.5276-5285, 2018. https://arxiv.org/abs/1804.09699

[57] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel, CNN-Cert: An Efficient Framework for Certifying Robustness of Convolutional Neural Networks, The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019), pp.3240-3247, 2019. https://arxiv.org/abs/1811.12395

[58] Tsui-Wei Weng, Pin-Yu Chen, Lam Nguyen, Mark Squillante, Akhilan Boopathy, Ivan Oseledets, and Luca Daniel, PROVEN: Verifying Robustness of Neural Networks with a Probabilistic Approach, International Conference on Machine Learning (ICML 2019), PMLR

vol. 97, pp.6727-6736, 2019. http://proceedings.mlr.press/v97/weng19a.html

[59] Nicholas Carlini and David Wagner, Towards Evaluating the Robustness of Neural Networks, IEEE Symposium on Security and Privacy (SP), pp.39-57, 2017. https://arxiv.org/abs/1608.04644

[60] Tsui-Wei Weng, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, Dong Su, Yupeng Gao, Cho-Jui Hsieh, and Luca Daniel, Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach, International Conference on Learning Representations (ICLR 2018), 2018. https://arxiv.org/abs/1801.10578

[61] Eric Wong and J. Zico Kolter, Provable defenses against adversarial examples via the convex outer adversarial polytope, International Conference on Machine Learning (ICML 2018), PMLR vol. 80, pp.5283-5292, 2018. https://arxiv.org/abs/1711.00851

[62] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana, Certified Robustness to Adversarial Examples with Differential Privacy, The IEEE Symposium on Security and Privacy (SP), 2019. https://arxiv.org/abs/1802.03471

[63] Jeremy M Cohen, Elan Rosenfeld, and J. Zico Kolter, Certified Adversarial Robustness via Randomized Smoothing, The 36th International Conference on Machine Learning (ICML 2019), 2019. https://arxiv.org/abs/1902.02918

[64] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu, Towards Deep Learning Models Resistant to Adversarial Attacks, The Sixth International Conference on Learning Representations (ICLR 2018), 2018. https://arxiv.org/abs/1706.06083

Chapter 7:

[65] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio, Fantastic Generalization Measures and Where to Find Them, International Conference on Learning Representations (ICLR 2020). https://arxiv.org/abs/1912.02178

[66] Gintare Karolina Dziugaite, Alexandre Drouin, Brady Neal, Nitarshan Rajkumar, Ethan Caballero, Linbo Wang, Ioannis Mitliagkas, and Daniel M. Roy, In search of robust measures of generalization, NeurIPS 2020. arXiv:2010.11924. https://arxiv.org/abs/2010.11924

[67] Andreas Maurer, A Note on the PAC Bayesian Theorem, arXiv:cs/0411099, 2004. https://arxiv.org/abs/cs/0411099

[68] John Langford and Rich Caruana, (Not) Bounding the True Error, NIPS, 2001 https://papers.nips.cc/paper_files/paper/2001/hash/98c7242894844ecd6ec94af67ac8247d-Abstract.html

[69] María Pérez-Ortiz, Omar Rivasplata, John Shawe-Taylor, and Csaba Szepesvári, Tighter risk certificates for neural networks, Journal of Machine Learning Research, 2021. arXiv:2007.12911. https://arxiv.org/abs/2007.12911

[70] Yoshinao Isobe, Estimating Generalization Error Bounds for Worst Weight-Perturbed Neural Classifiers, *Proceedings of the 38th Annual Conference of the Japanese Society for Artificial Intelligence*, 2024 (in Japanese).

[71] Yoshinao Isobe, WP-GEB-Estimator -- WP-GEB: Weight-Perturbed Generalization Error Bounds. https://staff.aist.go.jp/y-isobe/wp-geb-estimator

[72] John Langford, Tutorial on Practical Prediction Theory for Classification, JMLR, vol.6, No.10, pp.273–306, 2005. https://jmlr.org/papers/v6/langford05a.html

[73] Oliver Catoni, PAC-Bayesian Supervised Classification: The Thermodynamics of Statistical Learning, Institute of Mathematical Statistics, Lecture Notes-Monograph Series, vol. 56, 2007. https://www.jstor.org/stable/i20461497

[74] Guillermo Valle-Pérez and Ard A. Louis, Generalization bounds for deep learning, arXiv:2012.04115v2, 2020. https://arxiv.org/abs/2012.04115

[75] Saurabh Garg, Sivaraman Balakrishnan, J. Zico Kolter, and Zachary C. Lipton, RATT: Leveraging Unlabeled Data to Guarantee Generalization, ICML 2021. arXiv:2105.00303. https://arxiv.org/abs/2105.00303

[76] Wenda Zhou, Victor Veitch, Morgane Austern, Ryan P. Adams, and Peter Orbanz, Non-vacuous Generalization Bounds at the ImageNet Scale: a PAC-Bayesian Compression Approach, ICLR 2019. https://arxiv.org/abs/1804.05862

[77] Yu-Lin Tsai, Chia-Yi Hsu, Chia-Mu Yu, and Pin-Yu Chen, Formalizing Generalization and Adversarial Robustness of Neural Networks to Weight Perturbations, NeurIPS 2021. https://proceedings.neurips.cc/paper/2021/hash/a3ab4ff8fa4deed2e3bae3a5077675f0-Abstract.html

Chapter 8:

[78] X. Ma, Characterizing adversarial subspaces using Local Intrinsic Dimensionality, 2018.

[79] D. Meng , Magnet: a two-pronged defense against adversarial examples, in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017.

[80] W. Xu, Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks, in Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS), 2018.

[81] Shiqing Ma, NIC: Detecting Adversarial Samples with Neural Network Invariant Checking, Network and Distributed Systems Security Symposium (NDSS), NDSS 2019.

[82] National Institute of Advanced Industrial Science and Technology (AIST), AI Bridging Cloud Infrastructure, https://abci.ai/ja/

[83] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE, vol. 86, no. 11, pp.2278–2324, 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[84] A. Krizhevsky and G. Hinton, Learning multiple layers of features from tiny images, 2009.

[85] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, Labeled faces in the wild: A database for studying face recognition in unconstrained environments, University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.

[86] Nicolas Papernot, Ian Goodfellow, Ryan Sheatsley, Reuben Feinman, and Patrick McDaniel. cleverhans v1.0.0: an adversarial machine learning library. arXiv preprint arXiv:1610.00768,

2016.

[87] CleverHans, https://github.com/cleverhans-lab/cleverhans

[88] Masashi Sugiyama, Taiji Suzuki, and Takafumi Kanamori, Density Ratio Estimation in Machine Learning, Cambridge University Press, 2012.

Chapter 9:

[89] Yoshihiro Okawa and Kenichi Kobayashi, A Survey on Concept Drift Adaptation Technologies for Unlabeled Data in Operation, *Proceedings of the 35th Annual Conference of the Japanese Society for Artificial Intelligence*, pp.1-4, 2021 (in Japanese), https://doi.org/10.11517/pjsai.JSAI2021.0_2G4GS2f03.

[90] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia, A survey on concept drift adaptation, *ACM Computer Surveys*, vol. 46, no. 4, pp.1-37, 2014.

[91] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan ZhangJ, Learning under Concept Drift: A Review, in *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346-2363, 2019.

[92] Tsutomu Ishida, Hiroaki Kingetsu, Yasuto Yokota, Yoshihiro Okawa, Kenichi Kobayashi, and Katsuhito Nakazawa, Evaluation of Concept Drift Detection Methods for Unlabeled Data in Operation, *Proceedings of the 34th Annual Conference of the Japanese Society for Artificial Intelligence*, pp.1-4, 2020 (in Japanese), https://doi.org/10.11517/pjsai.JSAI2020.0_4Rin105.

[93] Yoshihiro Okawa and Kenichi Kobayashi, Recent Research Trends in Unsupervised Adaptation Techniques for Data Changes, *Proceedings of the 36th Annual Conference of the Japanese Society for Artificial Intelligence*, pp.1-4, 2022 (in Japanese), https://doi.org/10.11517/pjsai.JSAI2022.0_3Yin240.