博士学位申請論文

博士学位申請論文名

Output augmentation: A novel method of data augmentation

博士学位申請者氏名：　江口　脩　江口 (印)

（日本語訳）

出力拡張：データ拡張の新しい手法

（令和　3 年　12 月　20 日　提出）

Abstract

Since the 2000s, deep neural networks have been an active area of research, and we have succeeded in obtaining high performance by using deep neural networks.

In order to achieve the performance, it is well known that we need a sufficient amount of training data. When the amount of the given data is less or the data is biased, we usually apply data augmentation to them. The data augmentation often makes up for such weakness of the data and improves the generalization performance.

In this thesis, we propose a new method of data augmentation, which we call output augmentation, aiming to improve the generalization performance without using traditional methods of data augmentation. For some given training data, the output augmentation generates new data as if they are outputs from the neural network so that the average of this new data coincides with the training output data. We capture the effectiveness of output augmentation by demonstrating image classification for the CIFAR-10 and CIFAR-100. In our experiments, we confirmed that the method of output augmentation improves the accuracy for test data. From this, we expect that this output augmentation will become a new superior method of data augmentation than any others we already have.

Acknowledgments

This comment is dedicated to Professor Masaru Tanaka. Unfortunately, he has passed away on June 13th, 2021. I sincerely express my great appreciation for his extremely wide and deep kindness and cares as my supervisor which I have been given during the days of my research.

I would like to express my deepest gratitude to Professor Masaru Tanaka, who always gave advice and careful guidance to me. I would like to thank Professor Jun Fujiki, who always paid careful attention to my research. I am especially grateful for his easy-to-understand comments when I was being in trouble with my research. I would like to thank Professor Takafumi Amaba, who gave many useful comments to me, even though his field of expertise is apart from that of mine. My grateful thanks are also extended to Professor Takahiro Aoyama, who kindly helped me to improve my English.

Last but not least, I would like to thank the graduate students, and all the staffs of Department of Applied Mathematics in Fukuoka University for their heartful guidance, suggestions, and assistance over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Artificial intelligence (AI) is a broad concept including information processing of what humans perceive as intelligent and even the computers themselves which are equipped with such ability of information processing. Systems or algorithms to implement the ability into such devices are called *machine learning*. Machine learning is a subfield of AI and serves as a system which mimics the learning mechanisms of human learning. Among a variety of methods in machine learning, *deep learning* is still making great progress. Deep learning is a method of training a statistical model, which mimics the mechanism of how electrical signals transmitting neurons in the human brain are processed. The statistical model has a hierarchy consisting of many layers and is called an (artificial) *neural network*. Each layer has a lot of neurons, each behavior of which is controlled by parameters. Therefore the neural network has a lot of parameters. With this nature, deep learning is done by a large amount of computations, which we usually leave to computers. The study of neural networks has a long history which started beginning around 1940 and occured two booms.

The first boom was triggered by a psychologist D. O. Hebb, who researched about information processing and learning mechanisms in the brain. His research focused on neurons, which build neural networks in the brain. He proposed a hypothesis that when a signal is transmitted to a neuron, the signal is transmitted to the next neuron by the neuronal fire. He also advocated the Hebbian learning rule ([1]). The rule states that when a neuron fires, the weight associated with the input to the neuron increases, and the nexus of the neuron and the previous neurons (synaptic transmission efficiency) is strengthened. Hebb's research is still the basis of modern understanding of the brain. Next, W. McCulloch and W. Pitts considered a logic circuit, where we regard units in the circuit as neurons. The circuit outputs "1" if the input signal is true and "0" otherwise. This made an artificial neural network possible to solve the linear separation problem, which concerns partitioning the input data by a hyperplane ([2]). Next, F. Rosenblatt introduced a *perceptron*, which is a logic circuit consisting of an input layer and an output layer. Each nexus of units in it is assumed to obey the Hebbian learning rule. However, the first boom came to an end when M. Minsky and S. Papert showed that a perceptron consisting of input and output layers could only solve the linear separation problems for input data ([3]). Although Rosenblatt's work had shown that neural networks could solve nonlinear separation problems of data by increasing the number of hidden layers, it had not been known how to train the neurons. On the other hand, researchers had not known how to adjust the parameters in the many hidden layers.

The second boom was triggered by the invention of the *backpropagation* ([4]). Usually, a multilayer perceptron processes data from the input layer towards the output layer. The backpropagation is a learning method that adjusts the parameters in each layer backwards; it goes from the output layer to the input layer based on a loss function (error function), which is defined by using the output of the perceptron. D. E. Rumelhart and L. McClelland discovered that a huge number of neurons, which are distributed in the human brain, are connected together, and they form a network. This is the reason why the multilayer perceptrons came to be called *neural networks*. With these two contributions, it became possible to train the neural networks, which we couldn't in the first boom; This allowed us to solve several nonlinear separation problems of data, which had embarrassed us in the first boom.

Y. LeCun introduced *convolutional neural networks* (CNNs), which modeled the visual cortex of the brain. Some of the hidden layers are called *convolutional layers*. CNNs have achieved high performance in the handwriting classification ([5]). The classification problem refers to estimating a correct label associated with each input data. For example, any picture of a cat has the string "cat" as its label. After we make a model learn using a training dataset which consists of data with their true labels, we want to make the model estimate what to be the correct label when a new input data is given. In traditional neural networks, features of the data are extracted in each layer, and they form what is averaged over units in the layer. On the other hand, each convolutional layer in a CNN has *filters*, each of which consists of several parameters. The filters can get what parts of the data grasp the features. As a result, we recognize that it extracts unique features effectively.

We had these good equipment, but there were still problems during this second boom. First, neural networks had not reached a practical level of performance. The neural networks got high performance on data used to train the networks, but the *generalization performance*, which is a performance for *test data*, was still often low. The second problem was that the computers had taken a huge amount of time for computation since the performance of computers (GPUs) was not high enough to make the learning process. Because of these problems, the second boom was over around 1995, and the field of AI research had entered a winter period.

Since the 2000s, various models of neural networks have been proposed, and the performance of computers (GPUs) has been improved. Then we could make it easier to obtain large amounts of data because of the development of cloud systems. Along with this, research has been promoted and has received interest, and many researchers are still working.

To obtain a model with high generalization performance, we have to keep both the loss function for training data and that for test data decreasing during the training process. In many cases, the loss function for the training data gets down during the training process but may not for the test data. This is called *overtraining* or *overfitting*. For the reason of overtraining, the amount and quality of the training data are not sufficient to make the model obtain high performance on the test data. This is where the method of *data augmentation* is applied. With this, training data can be augmented to make up for their lack, to balance them when they are biased and to improve the quality of data as training data. Therefore, we can expect that the generalization performance is improved by using the data augmentation.

In this thesis, we propose *output augmentation* ([6]), which is a new approach to improve generalization performance without data augmentation. For some given training data, the output augmentation generates new data as if they are outputs from the neural network. We will show that output augmentation strongly helps to get higher generalization performance.

1.2 Purpose of the study and structure of this thesis

Purpose of the study

We capture the effectiveness of output augmentation by demonstrating image classification for the datasets CIFAR-10 and CIFAR-100, which are widely used in the field of computer vision. Specifically, we consider neural networks with a training dataset which is obtained with and without using data augmentation. Then we let them learn usually or by using output augmentation. Accordingly, we obtain four types of trained models, and we compare them by focusing on their generalization performance. We then show that the proposed output augmentation improves the generalization performance of image classification and then we obtained the following results:

- For the model trained without data augmentation, the method of output augmentation improves the accuracy for the test data.
- In the case of the model with data augmentation, the method of output augmentation seems to improve the accuracy not so much.

Structure of this thesis The thesis is organized as follows:

Chapter 1: Background, motivation, and positioning of this study.

Chapter 2: Related research on data augmentation.

Chapter 3: Overview of neural networks.

Chapter 4: Output augmentation.

Chapter 5: Experimenting image classification by using output augmentation.

Chapter 6: Conclusions and future work.

Chapter 2

Previous research

Data augmentation is a method to generate data from a given training dataset to prepare a new training dataset to use in training a model. With this, training data can be augmented to make up for their lack, to balance them when they are biased, and to improve the quality of data as training data. It applies to various types of a dataset, such as image and audio data. Then the generalization performance is naturally expected to be improved. We will mainly discuss image data since this thesis deals with them.

For example, in the case where we deal with image data, data augmentation often performs affine transformations and changing colors of images. The affine transformations have the effects of rotations, reflections, scalings (zooming in and out), and croppings. These give rise to geometric deformations in the image. This method allows us to increase the number of training data ([7]), and we can use the dataset as a new training dataset to use when we train models. It is then known that these procedures balance the dataset([8]), improve the quality of data and improve training efficiency. The most common methods of color modification include histogram equalization, contrast or brightness adjustment, white balancing, sharpening, and blurring ([9]). These methods are fast, repeatable, and are often used, and codes implementing them can be used within the modern deep learning frameworks such as TensorFlow ([10]), PyTorch ([11]), etc.

However, we have to take care not to lose the essence of objects in the image when using data augmentation. In other words, it is necessary to take into account the learning procedure of a neural network so that they can discriminate between the differences among objects in images after training. For example, in the handwritten hiragana classification, if an image of hiragana is inverted using data augmentation, it will be rendered meaningless as hiragana. Another example is the classification of handwritten numbers, where a “6” becomes a “9” when rotated 180 degrees. These matters are often overcome by introducing the so-called *domain knowledge* of data, knowledge of a specific field under consideration. If one does not pay attention to the domain knowledge of the data, applying data augmentation to datasets may lead to poor performance of trained models.

Among data augmentation, there is a very powerful method called *AutoAugment* ([12]) that automatically selects the appropriate one from several methods of data augmentation. AutoAugment tests various kinds of data augmentations on a subset of the training dataset and selects the best performing one among them. Therefore, AutoAugment assigns each of the training data with an optimal data augmentation. Throughout the whole procedure of training, the optimal method is then called and applied to each subset of the training dataset. This is very powerful, but it takes a very high computational cost because it needs to try many different data augmentation methods.

For audio data, several methods of data augmentation are used. For example, time-stretching is used to speed up or slow down the playback speed. There are also other methods of cutting specified frequencies of sound and turning down parts of sounds for a certain period of time ([13]). The sounds that we hear in our daily lives are often mixed with a lot of noise, making it difficult to hear them. In order to reproduce artificially such situations, there is a method of adding noise generated from a normal distribution to the audio data. There is also a method to increase the number of audio data by cutting out a portion of the speech time ([14]).

Chapter 3

Neural networks

3.1 Artificial intelligence and neural networks

In this section, we will explain the position of neural networks by following [15].

Artificial intelligence is a field of research containing three main subfields: machine learning, neural networks, and deep learning. The relation of them is shown in the figure below.

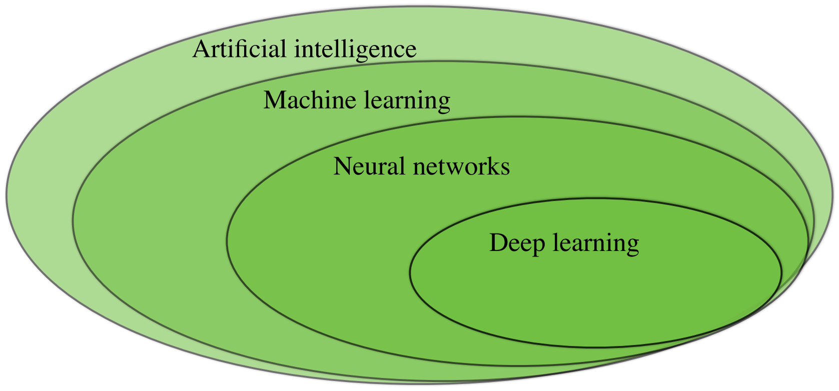

Figure 3.1: Positioning of neural networks.

We will explain each of them in the following.

Artificial intelligence

Artificial intelligence refers to a general term for programs that models information-processing of how humans think, or for anything we find intelligent. Machine learning, neural networks, and deep learning, which will be explained below, are regarded as typical examples of artificial intelligence.

Machine learning
Machine learning is a general terminology of making computers and other devices gain abilities of mimicking how humans learn from their experiences. This usually consists of various computer algorithms. By discovering rules and patterns from the data used for learning, and then by applying the rules and patterns to new data, the machine gets the ability of prediction for unknown data.

Neural networks
The human brain is composed of more than hundreds of millions of neurons. When a neuron is stimulated, an electrical signal is transmitted. If the electrical signal exceeds a certain level called a *threshold*, then the signal is transmitted to the next neuron. A neural network is a model of how the neurons (Figure 3.2) in the human brain are connected to each other. In the neural network, the counterpart of a neuron in the brain is called a *unit*.

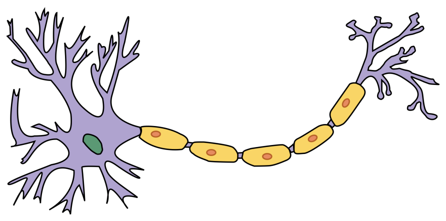

Figure 3.2: Neuron. This picture is from [16].

Deep learning
In some neural networks, all units in them are divided into some groups, called *layers*. A deep neural network is such a neural network which has many layers, and deep learning refers to training such deep neural networks. Generally, when we use machine learning, we can make computers learn patterns and rules efficiently by giving prior advice of what to take notice of the data. This advice is, however, not necessary for deep learning: Deep

neural networks can learn even that information. In other words, it is possible to train the networks efficiently without setting the features to be focused on, which is in turn the major advantages of deep learning. However, since the model can now learn such features by itself using deep learning, here arises a problem that we can not find what features the model extracted from the training dataset and the reason for decisions by the trained model.

3.2 Neural networks

As described above, a neural network is a model that mimics how the neurons build a network in the brain. We present the framework of neural networks by following [17, 18, 19]. The artificial substitutes of neurons in the model are called units (Figure 3.3). Each unit is equipped with similar mechanisms of neuronal input and output, as explained in the following. When an input signal u is given to a unit, the unit output a signal of the form

$$z = h(u + b).$$

Namely, if the value of input u exceeds the threshold $-b$ then the unit fires according to the value of the function h at the positive value of $u + b$. The function h is called an *activation function*.

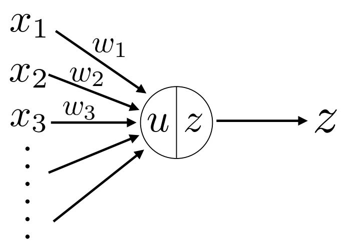

Figure 3.3: Unit.

There are many such units in a neural network. They are divided into several groups called layers and form a hierarchical structure according to which layer each unit belongs to. The number of units in a layer is called *width* of the layer. Layers in the network are called the *input layer* (first layer), second layer, third layer and so on, in order. The last layer of the network is called the *output layer*. The layers except the input layer and the output layer are called *hidden layers*. In the following, the number of all layers will be denoted by L, and for each $l = 1, 2, \dots, L$, we denote by n_l the number of units in the l-th layer. We number all units in the l-th layer as $1, 2, 3, \dots, n_l$.

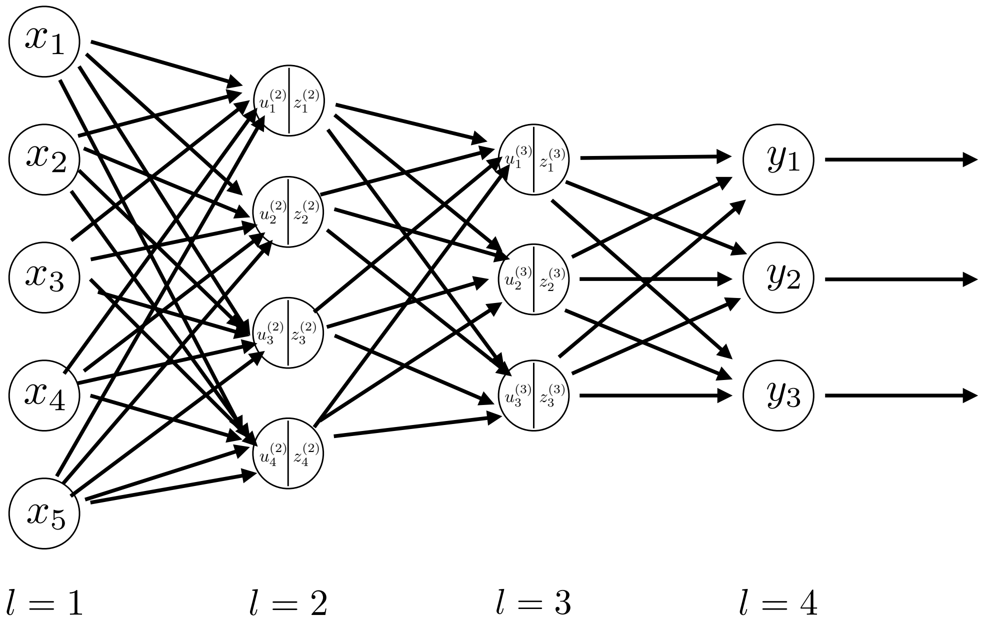

Figure 3.4: A neural network consisting of four layers.

Continuous functions that approximate the step function (Figure 3.5) are often used as activation functions of neural networks. Typical example is the sigmoid function shown in Figure 3.6.

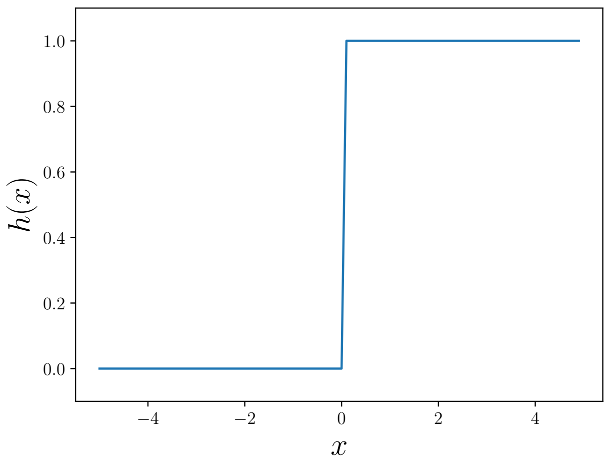

Figure 3.5: Step function.

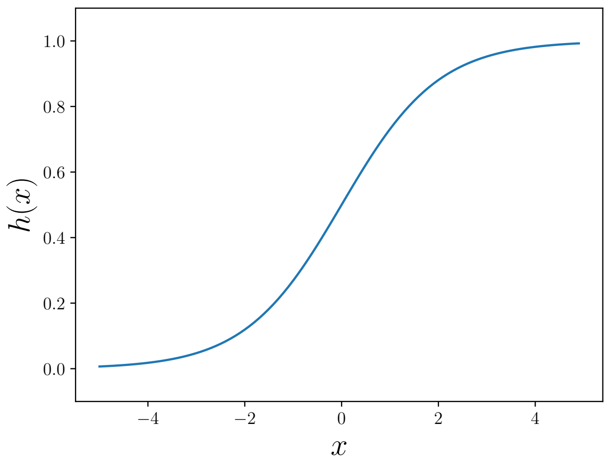

Figure 3.6: Sigmoid function.

3.2.1 Input layer ($l = 1$)

The layer that receives data to the neural network is called the input layer. In the neural network shown in Figure 3.4, the input layer consists of five units, so the input vector x has five components as follows.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}.$$

All units in the input layer do not have activation functions, therefore the output $z^{(1)} = \left[x_1, x_2, x_3, x_4, x_5\right]^\top$ from the input layer ($l = 1$) is equal to the input vector x. Namely, we have

$$z^{(1)} = x,$$

or equivalently to say,

$$z_i^{(1)} = x_i \quad \text{for } i = 1, 2, 3, 4, 5.$$

3.2.2 Hidden layers ($l = 2, \ldots, L-1$)

The output signal x from the input layer is then received to the next layer through a linear transformation represented by $W^{(2)} = \left[w_{ji}^{(2)}\right]_{ji}$. This matrix is a numerical representation of the strength of the connection between i-th unit in the input layer and j-th unit in the next layer.

The input $u_j^{(l)}$ of j-th unit in l-th layer
Let $z_i^{(l-1)}$ be the output from i-th unit in $(l-1)$-th layer. This is of the form

$$z^{(l-1)} = \left[z_1^{(l-1)}, z_2^{(l-1)}, \ldots, z_{n_{l-1}}^{(l-1)}\right]^\top \in \mathbb{R}^{n_{l-1}}.$$

This is received to the next layer after applying linear transformation described as

$$u^{(l)} = W^{(l)} z^{(l-1)},$$

where $W^{(l)} = \left[w_{ji}^{(l)}\right]_{ji}$ is an (n_{l-1}, n_l) matrix representing strength of the connection between i-th unit in the $(l-1)$-th layer and j-th unit in the l-th layer. With its components, the output $u^{(l)}$ can be written as follows.

$$u^{(l)} = \left[u_1^{(l)}, u_2^{(l)}, \ldots, u_{n_l}^{(l)}\right]^\top \in \mathbb{R}^{n_l}.$$

For each $j = 1, 2, \dots, n_l$, the component $u_j^{(l)}$ represents the input signal of the j-th unit in the l-th layer.

In the case of the neural network shown in Figure 3.4, the input $u_1^{(3)}$ to first unit of the third layer is calculated as follows:

$$\begin{aligned} u_1^{(3)} &= \sum_{i=1}^{4} w_{1i}^{(3)} z_i^{(3)} \\ &= w_{11}^{(3)} z_1^{(3)} + w_{12}^{(3)} z_2^{(3)} + w_{13}^{(3)} z_3^{(3)} + w_{14}^{(3)} z_4^{(3)}. \end{aligned}$$

The output signal $z_j^{(l)}$ of j-th unit in l-th layer

For a given input $u_j^{(l)}$ to j-th unit in l-th hidden layer, we consider the output $z_j^{(l)}$ from the unit. The output $z_j^{(l)}$ is obtained as a result of transformation given by applying the activation function $h^{(l)}$ to the output signal $u_j^{(l)}$ from the previous layer, which is shifted by a vector of bias parameters, namely

$$b^{(l)} = \left[b_1^{(l)}, b_2^{(l)}, \dots, b_{n_l}^{(l)}\right]^\top.$$

Consequently, we have the following: for $j = 1, 2, \dots, n_l$,

$$\begin{aligned} z_j^{(l)} &= h^{(l)}\left(u_j^{(l)} + b_j^{(l)}\right) \\ &= h^{(l)}\left(\sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} + b_j^{(l)}\right). \end{aligned}$$

We also write this relation by omitting subscripts as follows.

$$\begin{aligned} z^{(l)} &= h^{(l)}\left(u^{(l)} + b^{(l)}\right) \\ &= h^{(l)}\left(W^{(l)} z^{(l-1)} + b^{(l)}\right), \end{aligned}$$

where we reinterpret $h^{(l)}$ as a mapping $\mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ by applying $h^{(l)}$ componentwisely.

In the case of neural network shown in Figure 3.4, the output $z_1^{(3)}$ from the first unit in the third layer is calculated as follows:

$$\begin{aligned} z_1^{(3)} &= h^{(3)}\left(u_1^{(3)} + b_1^{(3)}\right) \\ &= h^{(3)}\left(\sum_{i=1}^{4} w_{1i}^{(3)} z_i^{(2)} + b_1^{(3)}\right) \\ &= h^{(3)}\left(w_{11}^{(3)} z_1^{(3)} + w_{12}^{(3)} z_2^{(3)} + w_{13}^{(3)} z_3^{(3)} + w_{14}^{(3)} z_4^{(3)} \ + b_1^{(3)}\right). \end{aligned}$$

The output $z_j^{(l)}$ from j-th unit in l-th layer has been so far written as

$$\begin{aligned} z_j^{(l)} &= h^{(l)}\left(u_j^{(l)} + b_j^{(l)}\right) \\ &= h^{(l)}\left(\sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} + b_j^{(l)}\right). \end{aligned}$$

Sometimes we let the bias $b_j^{(l)}$ have a meaning of a weight related to "0-th unit" in the $(l-1)$-th layer by considering the output of the unit is one, namely, we set $w_{j0}^{(l)} = b_j^{(l)}$ and $z_0^{(l)} = 1$. This convention makes the notation be simple. In fact, the output $z_j^{(l)}$ from the j-th unit in the l-th layer is then written as follows:

$$\begin{aligned} z_j^{(l)} &= h^{(l)}\left(u_j^{(l)} + b_j^{(l)}\right) \\ &= h^{(l)}\left(\sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} + b_j^{(l)}\right) \\ &= h^{(l)}\left(\sum_{i=0}^{n_{l-1}} w_{ji}^{(l)} z_i^{(l-1)}\right). \end{aligned}$$

3.2.3 Output layer ($l = L$)

Finally, we consider the output $\hat{y} = z^{(L)}$ from the L-th layer, i.e., the output layer. If output from the $(L-1)$-th layer is given by $z^{(L-1)} = \left[z_1^{(L-1)}, z_2^{(L-1)}, \ldots, z_{n_{L-1}}^{(L-1)}\right]^\top$, then the input to the L-th layer, which we denote by $u^{(L)} = \left[u_1^{(L)}, u_2^{(L)}, \ldots, u_{n_L}^{(L)}\right]^\top$, is given as follows:

$$\begin{aligned} u^{(L)} &= \left[u_1^{(L)}, u_2^{(L)}, \ldots, u_{n_L}^{(L)}\right]^\top \\ &= \left[\sum_{i=0}^{n_{L-1}} w_{1i}^{(L)} z_i^{(L-1)}, \sum_{i=0}^{n_{L-1}} w_{2i}^{(L)} z_i^{(L-1)}, \ldots, \sum_{i=0}^{n_{L-1}} w_{n_L i}^{(L)} z_i^{(L-1)}\right]^\top, \end{aligned}$$

where we have set $z_0^{(L-1)} = 1$ and for each $j = 1, 2, \ldots, n_L$, $w_{j0}^{(L)}$ stands for a bias. If we put $W^{(L)} = \left[w_{ji}^{(L)}\right]_{ji}$, this relation is also written as $u^{(L)} = W^{(L)} z^{(L-1)}$. Then an activation function $h^{(L)}$ applies to this componentwisely and we obtain the output as follows:

$$\begin{aligned}\hat{y}_j &= z_j^{(L)} \\ &= h^{(L)}\left(u_j^{(L)}\right) \\ &= h^{(L)}\left(\sum_{i=0}^{n_{L-1}} w_{ji}^{(L)} z_i^{(L-1)}\right).\end{aligned}$$

To summarize, we can think of a neural network as a family of functions described as follows:

$$\begin{aligned}\hat{y} &= h^{(L)}(u^{(L)}) \\ &= h^{(L)}(W^{(L)} z^{(L-1)}) \\ &= h^{(L)}\left(W^{(L)} h^{(L-1)}(u^{(L-1)})\right) \\ &\quad \vdots \\ &= h^{(L)}\Big(W^{(L)} h^{(L-1)}(W^{(L-1)} h^{(L-2)} \\ &\qquad (\cdots W^{(l+1)} h^{(l)}(W^{(l)} h^{(l-1)}(\cdots W^{(2)} h^{(1)}(x))) \cdots))\Big),\end{aligned}$$

where the activation function $h^{(1)}$ in the input layer is assumed to be the identity mapping.

This completes the explanation of the mechanism of neural networks. Let us now summarize this into the mathematical definition in the following.

Definition 1. Let L be a natural number. For given natural numbers $n_1, n_2, \ldots, n_L$ and functions $h^{(l)} : \mathbb{R} \to \mathbb{R}$, $l = 2, 3, \ldots, L$, a neural network of depth L is the family $\{\hat{y}(x; w)\}_w$ of functions

$$\hat{y}(x; w) = \Psi \circ \Phi_{L-1,L-2}^{(W^{(L-1)}, b^{(L-2)})} \circ \cdots \circ \Phi_{l,l-1}^{(W^{(l)}, b^{(l-1)})} \circ \cdots \circ \Phi_{3,2}^{(W^{(3)}, b^{(2)})}(W^{(2)} x),$$

where w, $\Phi_{l,l-1}^{(W^{(l)}, b^{(l-1)})}$, Ψ are described as follows:

(1) $w = (W^{(l)}, b^{(l)})_{l=2}^{L}$, $(W^{(l)}, b^{(l)}) \in (\mathbb{R}^{n_l} \otimes \mathbb{R}^{n_{l-1}}) \times \mathbb{R}^{n_l}$, $l = 2, 3, \ldots, L$.

(2) For each l, $\Phi_{l,l-1}^{(W^l, b^{l-1})} : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ is a mapping defined by

$$\Phi_{l,l-1}^{(W^l, b^{l-1})}(u) = W^{(l)} h^{(l-1)}(u^{(l-1)} + b^{(l-1)}), \quad u^{(l-1)} \in \mathbb{R}^{n_{l-1}}.$$

(3) The mapping $\Psi : \mathbb{R}^{n_L} \to \mathbb{R}^{n_L}$ is defined by

$$\Psi(u^{(L)}) := h^{(L)}(u^{(L)} + b^{(L)}) := \begin{bmatrix} h^{(L)}(u_1^{(L)} + b_1^{(L)}) \\ h^{(L)}(u_2^{(L)} + b_2^{(L)}) \\ \vdots \\ h^{(L)}(u_{n_L}^{(L)} + b_{n_L}^{(L)}) \end{bmatrix}.$$

Then w is called the parameter, n_l is called the width of l-th layer, and $h^{(l)}$ is called the activation function in l-th layer.

3.2.4 Activation function

In this section, we introduce various activation functions.

Identity function

The identity function $h(x)$ returns the input as it is, that is, it holds that

$$h(x) = x.$$

In the case of regression problems, it is used in the output layer.

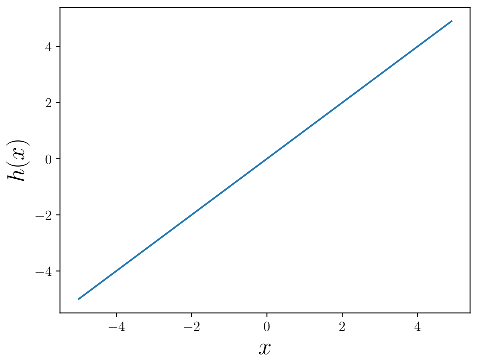

Figure 3.7: Identity function.

Step function

The step function outputs 1 if the input value is positive, and 0 otherwise. Namely, it is defined by

$$h(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases}$$

The step function is one of the oldest neuron models proposed by W. MacCulloch and W. Pitts in 1943 ([2]). Although the neuron model had been mainly used during the first boom of neural networks, it has not been for now. This is because the derivative of the step function can not be defined at $x = 0$ and it is 0 elsewhere, and therefore the function is not appropriate for the gradient descent method explained in Section 3.3.2.

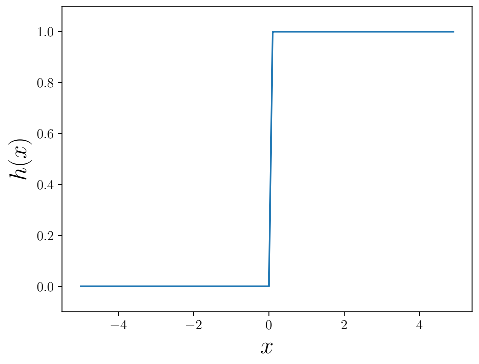

Figure 3.8: Step function.

Sigmoid function

The sigmoid function is an approximation of the step function and it is differentiable at any points. The representation of the sigmoid function is given by

$$h(x) = \frac{1}{1 + e^{-x}}. \tag{3.1}$$

The advantage of using the sigmoid function as a neuron model is that the derivative of the sigmoid function can be written by using the sigmoid function itself as follows:

$$\begin{aligned} h'(x) &= \frac{-e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\frac{-e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\ &= h(x)\left(1-h(x)\right). \end{aligned}$$

Therefore we can compute the derivative $h'(x)$ only by knowing the value of $h(x)$. As the sigmoid function returns values in the range from 0 to 1, the values are often interpreted as probabilities.

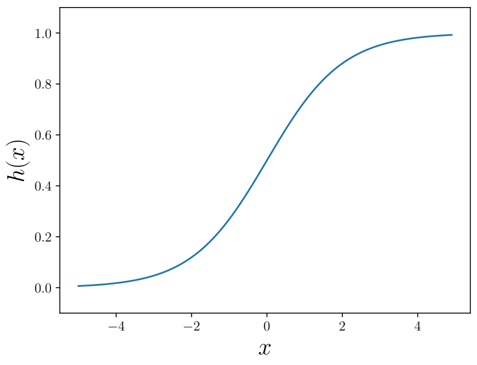

Figure 3.9: Sigmoid function.

Hyperbolic tangent function

When we need both positive and negative values of an activation function, the hyperbolic tangent function is a possible choice as an activation function. The representation of it is given by

$$h(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Using this, the sigmoid function (3.1) can be expressed as follows:

$$\frac{\tanh(x/2)+1}{2}.$$

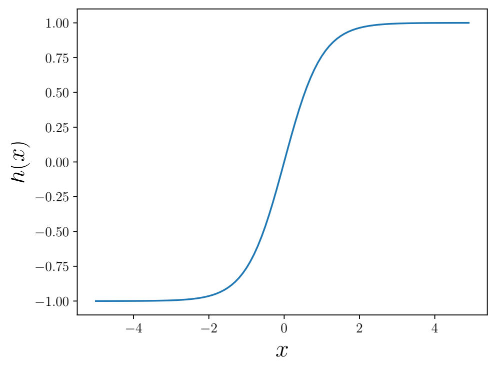

Figure 3.10: Hyperbolic tangent function.

Hard hyperbolic tangent function

The hard hyperbolic tangent function is a piecewise linear approximation of the hyperbolic tangent function. It is defined by

$$h(x) = \begin{cases} 1 & \text{if } 1 \leq x, \\ x & \text{if } -1 < x < 1, \\ -1 & \text{if } x \leq -1. \end{cases}$$

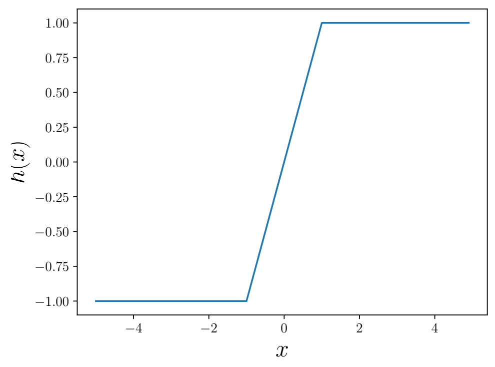

Figure 3.11: Hard hyperbolic tangent function.

Rectified linear unit (ReLU)
If the gradient of an activation function is too small, it causes the so-called *vanishing gradient problem*, which matters in an efficient learning using gradient descent method. The sigmoid function or the hyperbolic tangent function may be in the case as the gradients of them are less than 1. To avoid the problem, the rectified linear unit (ReLU) is widely used. The definition is as follows:

$$h(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases}$$

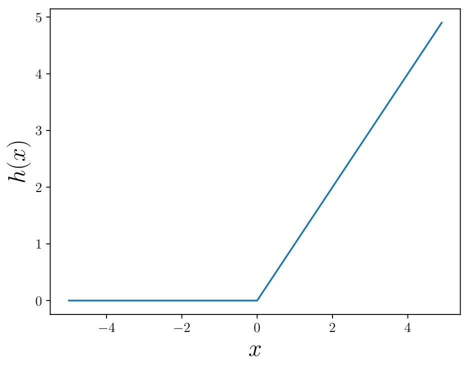

Figure 3.12: ReLU.

The derivative of ReLU is 1 for positive input, which resolves the vanishing gradient problem.

Softplus function
The softplus function is an approximation of ReLU and it is smooth. It is defined by

$$h(x) = \log(1 + e^x).$$

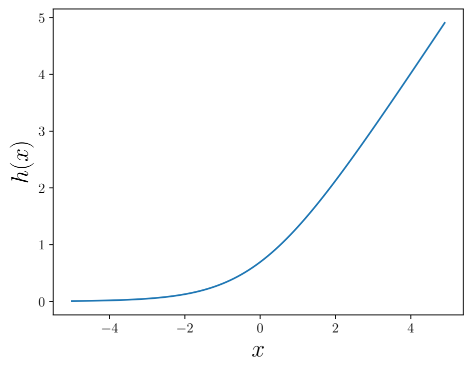

Figure 3.13: Softplus function.

Leaky rectified linear unit (Leaky ReLU)

The leaky rectified linear unit is an improvement of ReLU to reflect the effect of negative input. For a given $\alpha \in (0, 1]$, the representation of the leaky ReLU is given by

$$h(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x & \text{if } x \leq 0. \end{cases}$$

Usually the value of α is set to 0.01.

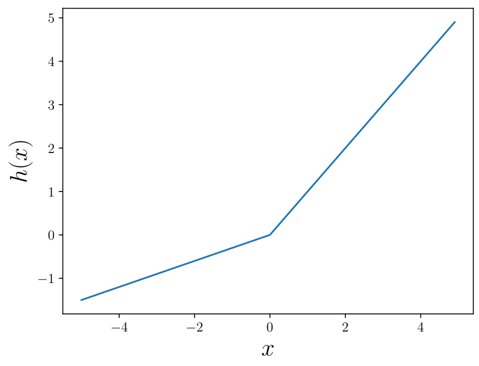

Figure 3.14: Leaky ReLU, $\alpha = 0.3$.

Softmax function

The activation function of the output layer of a neural network for classification problems is known as the *discriminant function*. The discriminant function is often chosen as a softmax function

$$h_i(x) = \frac{\exp(x_i)}{\sum_{j=1}^{J} \exp(x_j)}, \quad x = \left[x_1, x_2, \ldots, x_J\right]^\top.$$

The value $h_i(x)$ is interpreted as a probability that x belongs to i-th class.

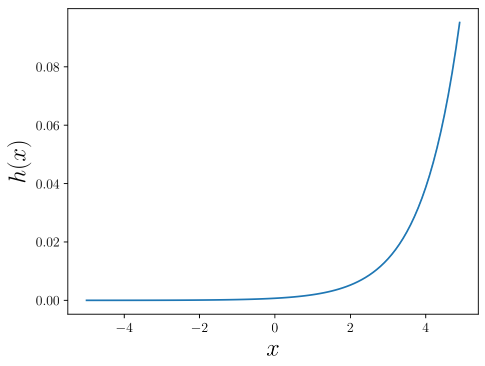

Figure 3.15: Softmax function.

3.3 Training a neural network

The training of a neural network means how to take "good" parameters w and is usually performed based on a loss function. The loss function is usually defined by using output of the network whose inputs are from a training dataset. Then the parameters are taken so that the loss function attains its minimum value.

3.3.1 Loss function

A loss function is a concept of how much we penalize the present parameters in the model and is usually realized as a function of parameters. We think that the smaller the penalty for the parameters, the closer to the "truth" or proper output (which is sometimes called a *label*, *target*, teacher signal and so on).

There are various types of loss functions, they are used according to what types of problems we want to solve. In the following, we introduce some of the most commonly used.

- Mean Square Error
  The mean square error is the basis of the loss function used not only in neural networks but also in various other models and problems. Typical one is a regression problem of fitting models to data.

  For a given training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, output $\hat{y}$ of a neural network, and the parameters w of the network, the mean square error is the sum of the squares of the difference between the output $\hat{y}(x_i; w)$ associated to x_i and the correct output y_i (the teacher signal):

  $$\mathcal{L}\left(\hat{y}(x_1; w), \hat{y}(x_2; w), \ldots, \hat{y}(x_N; w)\right) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}(x_i; w) - y_i\right)^2 .$$

  We denote this by $\mathcal{L}(w)$ if it causes no confusion.

- Cross-entropy for J-class classification problems
  Cross-entropy is often used for classification problems. In this setting, the training dataset is a form of $\mathcal{D} = \{(x_i; y_{i1}, y_{i2}, \ldots, y_{iJ})\}_{i=1}^N$, where for each i, only one of $y_{i1}, y_{i2}, \ldots, y_{iJ}$ is 1 and the others are 0. If $y_{ij} = 1$, then we think that "x_i belongs to the class j". Output $\hat{y}$ are modeled as probabilities and then we consider the cross-entropy of the output $\hat{y}(x_i; w)$ (probability) relative to the correct output $y_i = (y_{i1}, y_{i2}, \ldots, y_{iJ})$:

  $$\mathcal{L}\left(\hat{y}(x_1; w), \hat{y}(x_2; w), \ldots, \hat{y}(x_N; w)\right) = \frac{1}{N} \sum_{i=1}^{N} \left(- \sum_{j=1}^{J} y_{ij} \log \hat{y}_j(x_i; w) \right) .$$

  Minimizing this quantity is equivalent to maximizing the log-likelihood of the output of the network with respect to the training dataset. We denote this again by $\mathcal{L}(w)$ if it causes no confusion.

3.3.2 Gradient descent method

To find a minimum point of a loss function built by using a neural network is highly difficult and we can not often find its closed form expression. This motivates us instead to try a numerical calculation of the minimum point.

In this section, we introduce the *gradient descent method*, which is one of the methods to find the parameter w^* numerically that minimizes the loss function $\mathcal{L}(w)$. The concept is to get an optimal parameter by rolling a ball on the graph of the loss function as shown in Figure 3.16. The place where the ball finally stops represents the point $(w^*, \mathcal{L}(w^*))$.

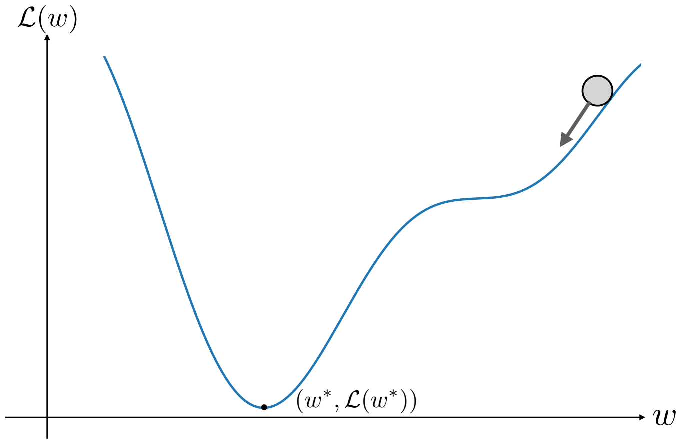

Figure 3.16: Gradient descent method 1.

For a given starting point of the ball, the gradient descent method determines the position

$$w(t) = \begin{bmatrix} w_1(t) \\ w_2(t) \\ \vdots \\ w_d(t) \end{bmatrix}$$

on the graph at time $t = 1, 2, 3 \ldots$. The initial value of the parameter $w(0)$ is often determined by following a d-multivariate normal distribution $\mathcal{N}(\mathbf{0}, \Sigma)$. We adopt this convention. The slope of the place where the ball places is calculated by the gradient of the loss function,

$$\nabla \mathcal{L}(w) = \frac{\partial \mathcal{L}(w)}{\partial w} = \begin{bmatrix} \dfrac{\partial \mathcal{L}(w)}{\partial w_1} \\ \dfrac{\partial \mathcal{L}(w)}{\partial w_2} \\ \vdots \\ \dfrac{\partial \mathcal{L}(w)}{\partial w_d} \end{bmatrix}.$$

Then the parameters are updated by using the following equation:

$$w(t+1) = w(t) - \eta \left. \nabla \mathcal{L}(w) \right|_{w=w(t)},$$

where, η is a hyperparameter called the *learning rate*, which adjusts how much of the parameter is updated at each time and has to be determined before we use the gradient descent method.

3.3.3 Stochastic gradient descent method

The gradient descent method allows us to update parameters by a simple algorithm. However, it might be problematic when the ball is trapped at the local minimum w^{**} which is not the global minimum, as shown in Figure 3.17. Depending on how large the learning rate is, the ball might be trapped in the well of the local minima eternally.

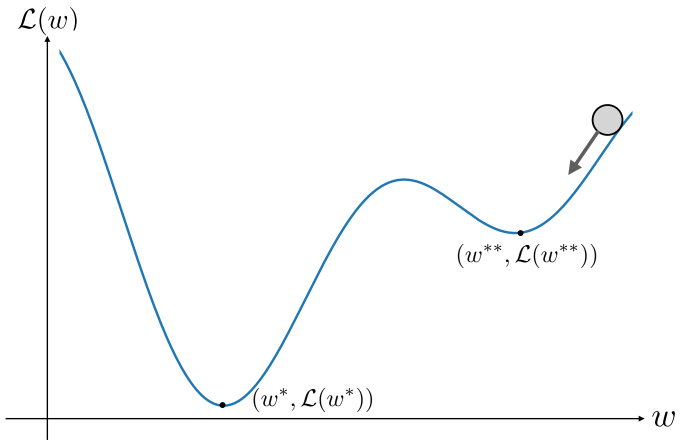

Figure 3.17: Gradient descent method 2: The point $(w^*, \mathcal{L}(w^*))$ is the goal which we want the ball to reach. But the ball might be trapped in the well around $(w^{**}, \mathcal{L}(w^{**}))$ if the velocity of the ball is not sufficient.

To avoid such a situation, the method of *stochastic gradient descent* (SGD) is also used. This method adds a random noise to the process of gradient descent method.

A subset of a given training dataset is called a *minibatch*. Training using a minibatch is called *minibatch learning* of the model. To make a clear difference, the training with the whole dataset is called a *batch learning*. Stochastic gradient descent or *online learning* refers to a collection of minibatch learnings associated with randomly-selected subsets of the whole dataset.

By using the output $\hat{y}(x_i; w)$ of the neural network and the training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, the loss function $\mathcal{L}(w)$ for batch learning is given by

$$\mathcal{L}(w) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(w),$$

where $\mathcal{L}_i(w)$ is a loss function associated with the datum (x_i, y_i). Some examples of $\mathcal{L}_i$ are described in the following.

- Summand of Mean Square Error

$$\mathcal{L}_i \left(\hat{y}(x_1; w), \hat{y}(x_2; w), \ldots, \hat{y}(x_N; w) \right) = \left(\hat{y}(x_i; w) - y_i \right)^2 . \tag{3.2}$$

- Summand of Cross-entropy for J-class classification

$$\mathcal{L}_i\left(\hat{y}(x_1;w),\hat{y}(x_2;w),\ldots,\hat{y}(x_N;w)\right) = -\sum_{j=1}^{J} y_{ij}\log\hat{y}_j(x_i;w). \tag{3.3}$$

Recalling that the parameters are updated by using the learning rate η and the following rule

$$w(t+1) = w(t) - \eta\,\nabla\mathcal{L}(w)\big|_{w=w(t)},$$

we see that the batch learning uses the whole training dataset in a single update of parameters.

Next, we will discuss about minibatch learning. Suppose that a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ is given. First, we take $M \in \mathbb{N}$ so that M divides N and then we divide $\mathcal{D}$ into M subsets $\mathcal{B}_t$'s as follows:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} = \mathcal{B}_1 \cup \mathcal{B}_2 \cup \cdots \cup \mathcal{B}_M,$$

where $\mathcal{B}_t = \{(x_{(t-1)\frac{N}{M}+i}, y_{(t-1)\frac{N}{M}+i})\}_{i=1}^{\frac{N}{M}}$ for $t = 1, 2, \ldots, M$. Each $\mathcal{B}_t$ is a minibatch. We have $|\mathcal{B}_1| = |\mathcal{B}_2| = \cdots = |\mathcal{B}_M| = \frac{N}{M}$ and each $|\mathcal{B}_t|$ is called the *minibatch size* of $\mathcal{B}_t$, which represents the number of data in the minibatch. When using stochastic gradient descent, we sort randomly the order of the training data before preparing minibatch $\mathcal{B}_t$ from the training dataset $\mathcal{D}$.

The loss function $\mathcal{L}_{\mathcal{B}_t}(w)$ associated with the minibatch $\mathcal{B}_t$ is defined by the following:

$$\mathcal{L}_{\mathcal{B}_t}(w) = \frac{1}{|\mathcal{B}_t|}\sum_{i:\ (x_i,y_i)\,\in\,\mathcal{B}_t}\mathcal{L}_i(w).$$

The parameters are updated by using the following equation.

$$w(t+1) = w(t) - \eta\,\nabla\mathcal{L}_{\mathcal{B}_t}(w)\big|_{w=w(t)}.$$

Each minibatch $\mathcal{B}_t$ is used to update the parameters. An *epoch* is the collection of all these updates. The epoch is said to be completed when these updates are done. In the next epoch, we take a new random division of $\mathcal{D}$ and repeat the same procedures. Those processes are continued until the values of the loss function are regarded as convergent.

3.3.4 Backpropagation

Deep neural networks have a large number of hidden layers and so a huge number of parameters. Then the calculation of gradient, which is needed for gradient descent, becomes very complicated and computationally expensive. *Backpropagation* is a method to speed up the update of parameters in each layer. It divides the calculation of the whole derivatives into that of each layer, and the computation goes backwards, i.e., from the output layer as we shall see below.

Recall that by using input to the respective layer,

$$\begin{aligned} u^{(l)} &= W^{(l)} z^{(l-1)} \\ &= W^{(l)} h^{(l-1)}(u^{(l-1)}), \end{aligned}$$

the output $\hat{y}$ of the neural network is expressed as follows:

$$\begin{aligned} \hat{y} &= h^{(L)}(u^{(L)}) \\ &= h^{(L)}(W^{(L)} z^{(L-1)}) \\ &= h^{(L)}\left(W^{(L)} h^{(L-1)}(u^{(L-1)})\right) \\ &\quad \vdots \\ &= h^{(L)}\Big(W^{(L)} h^{(L-1)}(W^{(L-1)} h^{(L-2)} \\ &\qquad (\cdots W^{(l+1)} h^{(l)}(W^{(l)} h^{(l-1)}(\cdots h^{(1)}(x))) \cdots))\Big). \end{aligned}$$

From this expression, we see that the output of the neural network is composed of many nonlinear functions. Therefore one naturally expects that the calculation of partial derivatives of $\hat{y}$, which is needed for that of the gradient $\nabla \mathcal{L}(w)$, will be a heavy task.

In order to explain the method of error backpropagation, we first consider the calculation of the gradient $\frac{\partial \mathcal{L}(w)}{\partial W^{(l)}}$ with respect to the parameter $W^{(l)}$ of l-th layer. Note that the output $u^{(l)}$ from the l-th layer can be written as $u^{(l)} = W^{(l)} z^{(l-1)}$. Then, by the chain rule for derivatives of the composite functions, the gradient $\frac{\partial \mathcal{L}(w)}{\partial W^{(l)}}$ can be calculated as follows.

$$\frac{\partial \mathcal{L}(w)}{\partial W^{(l)}} = \frac{\partial u^{(l)}}{\partial W^{(l)}} \frac{\partial \mathcal{L}(w)}{\partial u^{(l)}}.$$

Then, by using the relation $u^{(l)} = W^{(l)} z^{(l-1)}$, $\frac{\partial u^{(l)}}{\partial W^{(l)}}$ can be calculated as

$$\begin{aligned}\frac{\partial u^{(l)}}{\partial W^{(l)}} &= \frac{\partial}{\partial W^{(l)}} \left(W^{(l)} z^{(l-1)} \right) \\ &= z^{(l-1)}.\end{aligned}$$

For the term $\frac{\partial \mathcal{L}(w)}{\partial u^{(l)}}$, we again apply the chain rule and we get

$$\frac{\partial \mathcal{L}(w)}{\partial u^{(l)}} = \frac{\partial u^{(l+1)}}{\partial u^{(l)}} \frac{\partial \mathcal{L}(w)}{\partial u^{(l+1)}}.$$

By using the relation $u^{(l+1)} = W^{(l+1)} z^{(l)} = W^{(l+1)} h^{(l)}(u^{(l)})$, we have

$$\begin{aligned}\frac{\partial u^{(l+1)}}{\partial u^{(l)}} &= \frac{\partial}{\partial u^{(l)}} \left(W^{(l+1)} h^{(l)}(u^{(l)}) \right) \\ &= h^{(l)\prime}(u^{(l)}) W^{(l+1)\top}.\end{aligned}$$

Therefore, if we put $\delta^{(l)} = \frac{\partial \mathcal{L}(w)}{\partial u^{(l)}}$, we see that $\delta^{(l)}$ has the following recursive relation:

$$\begin{aligned}\delta^{(l)} &= \frac{\partial u^{(l+1)}}{\partial u^{(l)}}\, \delta^{(l+1)} \\ &= h^{(l)\prime}(u^{(l)}) W^{(l+1)\top}\, \delta^{(l+1)}.\end{aligned}$$

When data are input to the neural network, the computation proceeds from the input layer to the output layer via the hidden layers, but on the contrary, the derivatives are calculated by the above recursive formula and the values are determined from the output layer. Furthermore, the vector involved in the above formula is the derivative of the transformation between adjacent layers, which can be computed in advance and can be implemented easily on a computer when we use the gradient descent method. Hence we can easily solve the above recurrence formula on a computer.

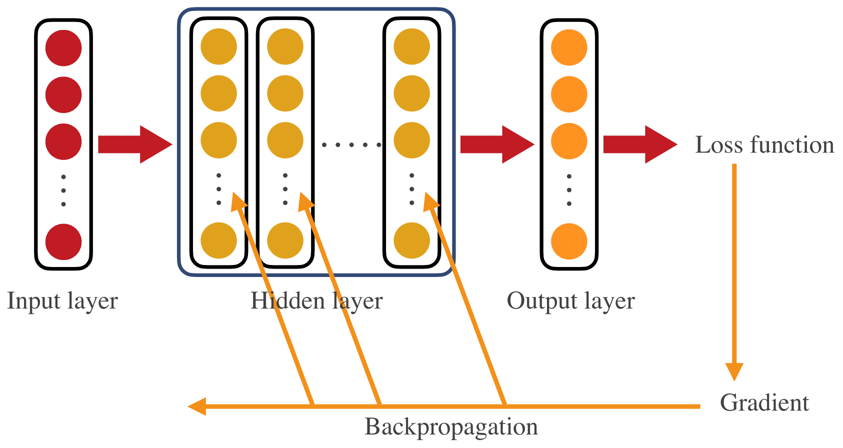

Figure 3.18: Updating parameters by backpropagation.

Chapter 4

Output augmentation

In this chapter, we propose a novel method named *output augmentation* (OA), which is the main contribution of the thesis. First, we explain data augmentation, which is a traditional method to increase the number of data. Next, we propose the OA, and explain the difference between OA and data augmentation.

4.1 Data augmentation

To obtain a well-trained neural network, we generally need large amounts of pairs of input data and their target outputs to avoid overfitting of the network. When the number of input data which we have is not sufficient, we need to increase the number of them. The method of data augmentation is applied to this situation. Data augmentation increases the number of input data by duplicating the existing data with slight changes. For example, when we use image data as input data, we require the target output of some input data to be invariant under some kind of geometric deformation of image such as rotations, reflections, and scalings. On the other hand, we can increase the number of input image data by applying geometric deformations to images which we have. However, geometric deformations have some kind of limitation. For example, when we rotate the digit "6" by 180 degrees, the digit no longer represents the target output "six"; it rather represents "nine". It means that a big deformation of image may affect its target output, therefore when we apply geometric deformations to images, we have to be careful not to change the target outputs. In some sense, the input data should be changed within some "suitable" range. Such kind of knowledge of the limitation is called *domain knowledge*, and is very important to use data augmentation.

For a pair (x, t) of the input and its target output, data augmentation is a method to generate new data $(\tilde{x}, t)$'s from a pair (x, t) where $\tilde{x}$'s are slightly modified copies of the input x (Figure 4.1-4.2).

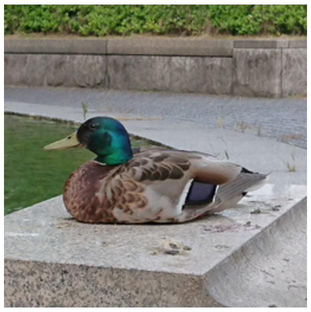

Figure 4.1: Example of image data x.

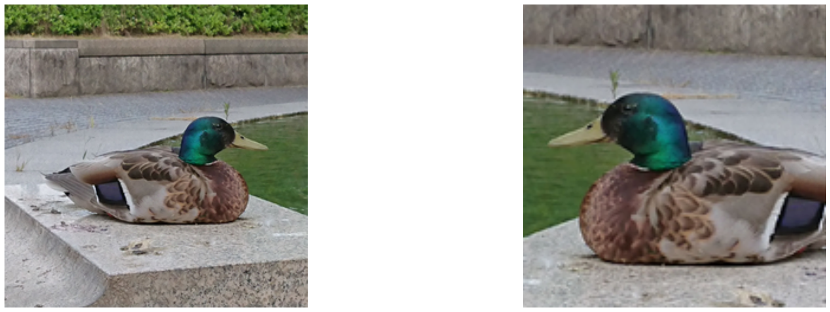

Figure 4.2: Examples of image data $\tilde{x}$'s generated by applying data augmentation to the image data x in Figure 4.1.

4.2 Concept of output augmentation

Let $y = \hat{y}(x; w)$ be the output of the neural network for input data x, and $\tilde{y} = \hat{y}(\tilde{x}; w)$ be the output for $\tilde{x}$. It is important to control how much $\tilde{x}$ deviates from the original x. It depends on the modulus of continuity of the network $\hat{y}(x; w)$. We naturally expect that $\tilde{y}$ places around y up to the distance ε in the output space: $|\tilde{y} - y| < \varepsilon$. For this, we have to control a radius parameter r which plays the role of how much $\tilde{x}$ deviates from x, i.e., $|\tilde{x} - x| < r$. It is r that is what we explained "suitable" range above, which is derived from domain knowledge. In later chapters, this will be controlled by a hyperparameter of a model.

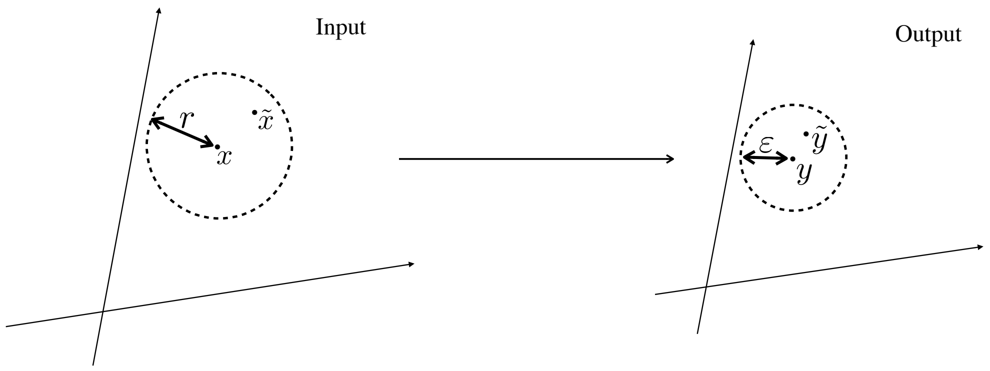

Figure 4.3: Points x and $\tilde{x}$ in the input space and the corresponding points y and $\tilde{y}$ in the output space.

Let (y, t) be a pair of the network-output and its target. OA is a method to generate virtual outputs $(\tilde{y}, t)$'s from the pair (y, t), where $\tilde{y}$'s are slightly modified copies of output y. Along with this, advantages of OA can be contrasted with data augmentation as follows.

- For data augmentation, we must take into account the *domain knowledge* of each input data, where the domain knowledge refers to the characteristics of the data. On the other hand, for OA, we only need to adjust the range of the augmented output without knowing the domain knowledge of the input data. In particular, we don't need to take care of the domain knowledge about the input training data, and this is a great advantage when we deal with image data.

- To update parameters many times until the parameters are regarded as convergent, training with a dataset obtained by a traditional data augmentation requires generating a lot of pairs of input data and output data. On the other hand, we need less input data if we adopt OA.

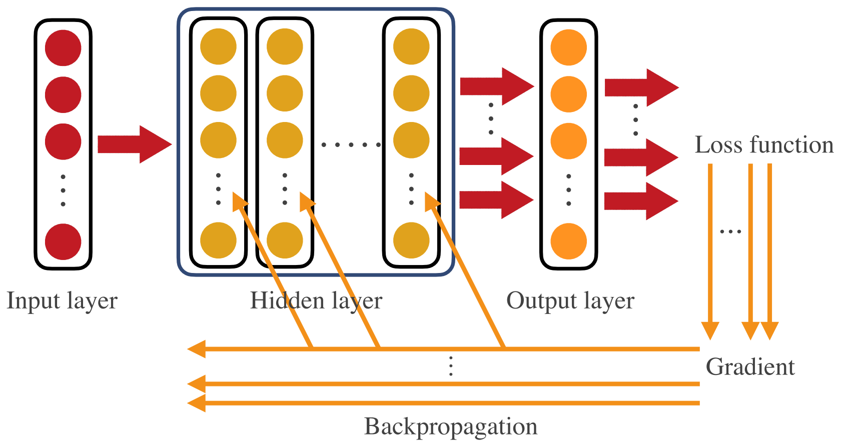

Figure 4.4: Training a neural network using OA.

4.3 OA and the associated rule of parameter update

In this section, we give a concrete description of output augmentation (OA) and then determine how to update the parameters using OA.

4.3.1 Description of OA

Suppose that we have a neural network of depth L. We denote the input variable by x. In the network, let $u^{(l)}$ be the input to the l-th layer, and let $z^{(l)}$ be the output from the l-th layer. We denote by $W^{(l)}$ and $h^{(l)}$ the parameters and the activation function in the l-th layer, respectively. Note that $u^{(l)}$ has the following recurrence equation.

$$\begin{aligned} u^{(l)} &= W^{(l)} z^{(l-1)} \\ &= W^{(l)} h^{(l-1)}(u^{(l-1)}). \end{aligned}$$

To describe OA, we first consider an augmentation of the neural network, in which the input variable is denoted by x. Let $\varepsilon = [\varepsilon^1, \varepsilon^2, \dots, \varepsilon^{n_L}]^\top$ be an n_L-dimensional random vector distributed as $\mathcal{N}(\mathbf{0}, \sigma^2 I_{n_L})$, where I_{n_L} is the n_L-order identity matrix and σ^2 is a hyperparameter. Then we set $\hat{y}_{\text{aug}}$ as follows:

$$
\begin{aligned}
\hat{y}_{\text{aug}} &= h^{(L)}(u^{(L)} + \varepsilon) \\
&= h^{(L)}(W^{(L)} z^{(L-1)} + \varepsilon) \\
&= h^{(L)}\left(W^{(L)} h^{(L-1)}(u^{(L-1)}) + \varepsilon\right) \\
&\ \ \vdots \\
&= h^{(L)}\Big(W^{(L)} h^{(L-1)}(W^{(L-1)} h^{(L-2)} \\
&\qquad (\cdots W^{(l+1)} h^{(l)}(W^{l} h^{(l-1)}(\cdots h^{(1)}(x))) \cdots)) + \varepsilon\Big).
\end{aligned}
$$

Compare this with the output of the neural network, i.e., $\hat{y}(x; w) = h^{(L)}(u^{(L)}(x; w))$. We sometimes write $\hat{y}_{\text{aug}}$ as $\hat{y}_{\text{aug}}(x; w)$ or $\hat{y}_{\text{aug},\varepsilon}(x; w)$ to emphasize the input variable x, the parameter $w = (W^{(l)})_{l=2}^{L}$ of the network, and even the noise ε we attached. We regard $\hat{y}_{\text{aug},\varepsilon}(x; w)$ as an augmentation of the output $\hat{y}(x; w)$ (or we might regard as an augmentation of the neural network itself), and in this sense, this procedure of generation of $\hat{y}_{\text{aug}}$ is what we call the *output augmentation* (OA). Let us emphasize that $\hat{y}(x; w)$ is augmented but not for x and y.

Typical forms of $\hat{y}_{\text{aug}}$ are varied according to choices of the last activation function $h^{(L)}$, some of which are presented in the following.

- If $h^{(L)}$ is a sigmoid function, we have

$$
\begin{aligned}
\hat{y}_{\text{aug}} &= h^{(L)}(u^{(L)} + \varepsilon) \\
&= \frac{1}{1 + \exp(-u^{(L)} + \varepsilon)}.
\end{aligned}
$$

- If $h^{(L)}$ is a softmax function for J-class classification, we have

$$
\hat{y}^{j}_{\text{aug}} = h^{(L)}_{j}(u^{(L)} + \varepsilon) = \frac{\exp(u^{(L)}_{j} + \varepsilon^{j})}{\sum_{m=1}^{J} \exp(u^{(L)}_{m} + \varepsilon^{m})}, \quad j = 1, 2, \ldots, J,
$$

where $n_L = J$, $\varepsilon = \left[\varepsilon^1, \varepsilon^2, \ldots, \varepsilon^J\right]^\top \sim \mathcal{N}(0, \sigma^2 I_J)$ and

$$
\hat{y}_{\text{aug}} = \left[\hat{y}^{1}_{\text{aug}}, \hat{y}^{2}_{\text{aug}}, \ldots, \hat{y}^{J}_{\text{aug}}\right]^\top .
$$

4.3.2 Rule of parameter update

Suppose that we are given a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$. Let $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N$ be N independent and identically distributed n_L-dimensional random vectors distributed as $\mathcal{N}(\mathbf{0}, \sigma^2 I_{n_L})$. We put $\varepsilon_* = [\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N]$. By using the augmentations $\hat{y}_{\mathrm{aug},\varepsilon_i}(x_i; w)$'s, we introduce below two loss functions $\mathcal{L}_{\mathrm{aug},\varepsilon_*}$ adapted to the framework of OA and tasks under consideration.

- Mean squared error:

$$\begin{aligned}\mathcal{L}_{\mathrm{aug},\varepsilon_*}(w) &= \frac{1}{N}\sum_{i=1}^N \{\hat{y}_{\mathrm{aug},\varepsilon_i}(x_i; w) - y_i\}^2 \\ &= \frac{1}{N}\sum_{i=1}^N \left\{h^{(L)}\left(u^{(L)}(x_i; w) + \varepsilon_i\right) - y_i\right\}^2,\end{aligned}$$

  where $u^{(L)}(x_i; w)$ stands for the input to the L-th layer corresponding to the input x_i and the parameter w.

- Cross-entropy for J-class classification: With noting that now $n_L = J$,

$$\begin{aligned}\mathcal{L}_{\mathrm{aug},\varepsilon_*}(w) &= \frac{1}{N}\sum_{i=1}^N \left\{-\sum_{j=1}^J y_{ij}\log \hat{y}^j_{\mathrm{aug},\varepsilon_i}(x_i; w)\right\} \\ &= \frac{1}{N}\sum_{i=1}^N \left\{-\sum_{j=1}^J y_{ij}\log h_j^{(L)}\left(u^{(L)}(x_i; w) + \varepsilon_i\right)\right\},\end{aligned}$$

  where $u^{(L)}(x_i; w)$ stands for the input to the L-th layer corresponding to the input x_i and the parameter w.

Note that the above loss functions $\mathcal{L}_{\mathrm{aug},\varepsilon_*}(w)$ determine penalties from $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ to the common parameters w of the augmented networks $\hat{y}_{\mathrm{aug},\varepsilon_i}(x, w)$, $i = 1, 2, \ldots, N$ rather than that of $\hat{y}(x, w)$.

For a given parameter $w(t)$ of the network, we update this by the following equation:

$$w(t+1) = w(t) - \eta \nabla_w \mathcal{L}_{\mathrm{aug},\varepsilon_*}(w)|_{w=w(t)}.$$

The gradient $\nabla_w \mathcal{L}_{\mathrm{aug},\varepsilon_*}(w) = \left[\frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial W^{(l)}}(w)\right]_{l=2}^{L}$ can be calculated as follows:

$$\begin{aligned}\frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial W^{(l)}} &= \frac{\partial u^{(l)}}{\partial W^{(l)}} \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial u^{(l)}} \\ &= z^{(l-1)} \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial u^{(l)}}.\end{aligned}$$

For $l = 1, 2, \ldots, L-1$, we define $\delta_{\mathrm{aug}}^{(l)}$ by

$$\begin{aligned}\delta_{\mathrm{aug}}^{(l)} :&= \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial u^{(l)}} \\ &= \frac{\partial u^{(l+1)}}{\partial u^{(l)}} \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial u^{(l+1)}} \\ &= h^{(l)\prime}(u^{(l)}) W^{(l+1)\top}\ \delta_{\mathrm{aug}}^{(l+1)}.\end{aligned}$$

For $l = L$, we define $\delta_{\mathrm{aug}}^{(l)} = \delta_{\mathrm{aug}}^{(L)} = \left[\delta_{\mathrm{aug},1}^{(L)}, \delta_{\mathrm{aug},2}^{(L)}, \ldots, \delta_{\mathrm{aug},J}^{(L)}\right]^\top$ by the following.

$$\begin{aligned}\delta_{\mathrm{aug},j}^{(L)} &= \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial u_j^{(L)}} \\ &= \sum_{k=1}^{J} \frac{\partial h_k^{(L)}(u^{(L)} + \varepsilon_*)}{\partial u_j^{(L)}} \frac{\partial \mathcal{L}_{\mathrm{aug},\varepsilon_*}}{\partial \hat{y}_k}, \quad j = 1, 2, \ldots, J.\end{aligned}$$

4.4 Algorithms for parameter update

In this section, we present algorithms to update parameters of a neural network by using OA. To compare with traditional data augmentation, we first explain the algorithm to update parameters by using data augmentation, and after that, we explain the algorithm to update parameters by using OA.

4.4.1 The case of traditional data augmentation

Suppose that we have a neural network of depth L and a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ is given. For each $(x, y) \in \mathcal{D}$, we apply traditional data augmentation to x to generate K augmentations $\tilde{x}^1, \tilde{x}^2, \ldots, \tilde{x}^K$. We put conventionally $\tilde{x}^0 = x$. By collecting these data, we obtain an enlarged dataset

$$\widetilde{\mathcal{D}} = \left\{(\tilde{x}^k, y) : (x, y) \in \mathcal{D},\ k = 0, 1, \ldots, K\right\}.$$

We randomly chose a division of $\widetilde{\mathcal{D}}$ into $(K+1)$ subsets (minibatches) $\mathcal{B}_t$'s, $\widetilde{\mathcal{D}} = \cup_{t=0}^{K} \mathcal{B}_t$. Note that all sizes of minibatches are the same, namely it holds that $|\mathcal{B}_t| \equiv |\widetilde{\mathcal{D}}|/(K+1) = |\mathcal{D}|$. Denote $\mathcal{B}$ as representative of one of minibatches $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_K$. Then we define a loss function $\mathcal{L}_{\mathcal{B}}$ associated with the minibatch $\mathcal{B}$ by

$$\mathcal{L}_{\mathcal{B}}(w) = \frac{1}{|\mathcal{B}|} \sum_{(\tilde{x}, y) \in \mathcal{B}} \mathcal{L}_{(\tilde{x}, y)}(w),$$

where $\mathcal{L}_{(\tilde{x},y)}(w)$ is a penalty from $(\tilde{x}, y)$ to the parameter w of the network $\hat{y}(x; w)$ determined depending on the task under consideration (see e.g., equations (3.2) and (3.3)). Then the traditional algorithm to update parameters is described in the following table. To make it simpler, we have renamed the elements in $\mathcal{B}_k$ as $\tilde{x}_1^k, \tilde{x}_2^k, \ldots, \tilde{x}_N^k$ (and now it may not hold that $\tilde{x}_i^0 = x_i$).

Table 4.1: Parameter update algorithm using traditional data augmentation, in each epoch; $w(t)$ is the initial parameter in the epoch. Totally $(K+1)$ updates of parameters are performed in an epoch.

Process	Minibatch	Input	Output
1	$\mathcal{B}_0$	$\tilde{x}_i^0$'s	$\hat{y}(\tilde{x}_i^0; w(t))$'s
2	$\mathcal{B}_1$	$\tilde{x}_i^1$'s	$\hat{y}(\tilde{x}_i^1; w(t+1))$'s
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$1+K$	$\mathcal{B}_K$	$\tilde{x}_i^K$'s	$\hat{y}(\tilde{x}_i^K; w(t+K))$'s

Process	Parameter update
1	$w(t+1) = w(t) - \eta \nabla_w \mathcal{L}_{\mathcal{B}_0}\|_{w=w(t)}$
2	$w(t+1+1) = w(t+1) - \eta \nabla_w \mathcal{L}_{\mathcal{B}_1}\|_{w=w(t+1)}$
$\vdots$	$\vdots$
$1+K$	$w(t+1+K) = w(t+K) - \eta \nabla_w \mathcal{L}_{\mathcal{B}_K}\|_{w=w(t+K)}$

4.4.2 The case of output augmentation

Suppose that a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ is given. We take independent and identically distributed random vectors $\varepsilon(1)_*, \varepsilon(2)_*, \ldots, \varepsilon(K)_* \sim \mathcal{N}(\mathbf{0}, \sigma^2 I_{n_L})^{\otimes N}$, where we recall that n_L is the width of the output layer. Note that for each $k = 1, 2, \ldots, K$, the random vector $\varepsilon(k)$ is of the form

$$\varepsilon(k)_* = [\varepsilon(k)_1, \varepsilon(k)_2, \ldots, \varepsilon(k)_N] = \begin{bmatrix} \varepsilon(k)_1^1 & \varepsilon(k)_2^1 & \ldots & \varepsilon(k)_N^1 \\ \varepsilon(k)_1^2 & \varepsilon(k)_2^2 & \ldots & \varepsilon(k)_N^2 \\ \vdots & \vdots & \vdots & \vdots \\ \varepsilon(k)_1^{n_L} & \varepsilon(k)_2^{n_L} & \ldots & \varepsilon(k)_N^{n_L} \end{bmatrix}.$$

For each $k = 1, 2, \ldots, K$, we denote by

$$\hat{y}_{\mathrm{aug},\varepsilon(k)_i}(x_i; w) = h^{(L)}\left(u^{(L)}(x_i; w) + \varepsilon(k)_i\right), \quad i = 1, 2, \ldots, N,$$

output augmentations of the output $\hat{y}(x_i; w)$'s. The associated loss function will be denoted by $\mathcal{L}_{\mathrm{aug},\varepsilon(k)_*}$. Then we determine an algorithm to update parameters as shown in the following table.

Table 4.2: Parameter update algorithm using OA, in each epoch; $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ is the training dataset; $w(t)$ is the initial parameter in the epoch. Totally $(K+1)$-updates of parameters are performed in an epoch, here, note that the parameters of outputs are fixed to $w(t)$ during all processes in an epoch.

Process	Batch	Input	Output
1	$\mathcal{D}$	x_i's	$\hat{y}(x_i; w(t))$'s
2	$\mathcal{D}$	x_i's	$\hat{y}_{\mathrm{aug},\varepsilon(1)_i}(x_i; w(t))$'s
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1+K	$\mathcal{D}$	x_i's	$\hat{y}_{\mathrm{aug},\varepsilon(K)_i}(x_i; w(t))$'s

Process	Parameter update
1	$w(t+1) = w(t) - \eta \nabla_w \mathcal{L}\vert_{w=w(t)}$
2	$w(t+2) = w(t+1) - \eta \nabla_w \mathcal{L}_{\mathrm{aug},\varepsilon(1)_*}\vert_{w=w(t)}$
$\vdots$	$\vdots$
$1+K$	$w(t+K+1) = w(t+K) - \eta \nabla_w \mathcal{L}_{\mathrm{aug},\varepsilon(K)_*}\vert_{w=w(t)}$

Note that the parameter $w(t)$ used in the network during whole processes in an epoch doesn't change. Therefore, if we stock calculations of the output from the layer just before the last in the first process, we can recycle them in the whole process of calculating outputs (in fourth column in Table 4.2) and gradients (fifth column in Table 4.2) in the sequel. This is a crucial feature of the OA method making the computational cost economic when compared with training using traditional data augmentation described in Section 4.4.1.

In Table 4.2, each process can be replaced by the minibatch learning by dividing $\mathcal{D}$ into minibatches. In the experiments reported in Chapter 5, we have used this algorithm in which each process is replaced by minibatch learning.

Chapter 5

Experiments

In this chapter, the effectiveness of output augmentation (OA) is shown by the experiments designed as follows. We perform image classification for the datasets CIFAR-10 and CIFAR-100 ([20]) by using ResNet-18 ([21]), which is a neural network of depth $L = 18$. CIFAR-10 and CIFAR-100 contain images of 10 and 100 different objects, respectively. We compare totally six models: One is plain and we denote it by "Basic"; One is trained using AutoAugment (AA), which is known as one of the most effective methods among traditional ones of data augmentation, and we express it as "AA"; Two of them are trained using OA where the noises to be attached have different distributions. We denote these by "OA" in both cases; The other two are trained using both AA and OA where the noises to be attached are the same as the previous models OA's, respectively. We denote these by "AA+OA" in both cases. For the models using OA below, the learning rates are fixed to $\eta = 0.01$ and ε denotes representative of noises we attach to each unit in the output layer.

- Basic
- AA
- OA ($K = 2, 5, 10$, $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, $\sigma^2 = 1, 3$)
- OA ($K = 2, 5, 10$, $\varepsilon \sim \mathcal{U}[-a, a)$, $a = 1, 3$)
- AA+OA ($K = 2, 5, 10$, $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, $\sigma^2 = 1, 3$)
- AA+OA ($K = 2, 5, 10$, $\varepsilon \sim \mathcal{U}[-a, a)$, $a = 1, 3$)

Regarding the description of parameters in the list above, $\mathcal{N}(\cdot)$ denotes a normal distribution, and $\mathcal{U}[-a, a)$ is the uniform distribution on the interval $[-a, a)$.

To make the total number of parameter-updates the same among these different models, we arrange the total number of epochs for each model. Then we compute the accuracy on test data, which is by definition the ratio between the number of correctly-classified data and that of total data, to observe the generalization performance of trained models.

Table 5.1: Classification accuracy for CIFAR-10 test data.

Method	CIFAR-10	
	Basic	AA
SGD (300 epochs)	0.7765	0.8848
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.8214	0.8789
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{N}(0, 3)$	**0.841**1	0.8828
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.8147	0.8880
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{U}[-3, 3)$	0.8311	**0.8922**
SGD (600 epochs)	0.7765	0.8921
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.8643	0.8891
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{N}(0, 3)$	**0.8750**	**0.8965**
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.8604	0.8933
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{U}[-3, 3)$	0.8110	0.8858
SGD (1100 epochs)	0.7765	**0.8936**
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.8009	0.8769
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{N}(0, 3)$	**0.8621**	0.8929
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.8010	0.8694
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{U}[-3, 3)$	0.8110	0.8858

Table 5.2: Classification accuracy for CIFAR-100 test data.

Method	CIFAR-100	
	Basic	AA
SGD (300 epochs)	0.4697	0.6360
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.5287	0.6289
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{N}(0, 3)$	0.5509	0.6330
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.5328	0.6277
OA (100 epochs): $K = 2$, $\varepsilon \sim \mathcal{U}[-3, 3)$	**0.5799**	**0.6391**
SGD (600 epochs)	0.4697	0.6490
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.5990	0.6372
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{N}(0, 3)$	**0.6209**	**0.6512**
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.5980	0.6391
OA (100 epochs): $K = 5$, $\varepsilon \sim \mathcal{U}[-3, 3)$	0.6013	0.6400
SGD (1100 epochs)	0.4697	**0.6501**
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{N}(0, 1)$	0.5416	0.6338
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{N}(0, 3)$	**0.5946**	0.6454
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{U}[-1, 1)$	0.4842	0.6167
OA (100 epochs): $K = 10$, $\varepsilon \sim \mathcal{U}[-3, 3)$	0.5106	0.6266

Table 5.1 and Table 5.2 show that the proposed method, OA, improves test accuracy significantly for both CIFAR-10 and CIFAR-100 datasets. On the other hand, in the case where both OA and traditional data augmentation are used, we find that the test accuracy is slightly improved. It is interesting that use of only OA improves the test accuracy significantly but not so much for use of both AA and OA. In the use of both AA and OA, input data is augmented and so "shifted" from the original one, and then the corresponding network-output is further "shifted", which might give too much deviance from the true target.

It is also worth mentioning that the test accuracy for Basic with OA is not so bad compared with that of AA with SGD in spite of the less computational cost. This suggests that OA can be used as a substitute for the traditional data augmentation if the hyperparameters, the distributions of ε's, of OA are well chosen.

Chapter 6

Conclusions

In this thesis, we proposed a new method of data augmentation, which is named output augmentation, to improve the generalization performance. Data augmentation is generally used on training data to compensate for the lack of the data or adjust the balance if the data are biased. For a given training data, our new method augments the output corresponding to the data, namely, it generates an arbitrary number of new data as if they are output from the neural network, while the traditional method of data augmentation does for the input data.

We have seen at least two main advantages of output augmentation: Traditional data augmentation for images requires the domain knowledge of the training data, and requires some laborious processing before adding the augmented data to the training dataset and feeding them to the neural network. On the other hand, output augmentation and the algorithms for parameter update with it don't touch the input data. Therefore we are free from such bothersome processing and it doesn't increase the amount of input data to feed to the neural network, which keeps the computational cost economical.

Finally, we showed that the method of output augmentation improves the accuracy for test data in the case of image classification for the datasets CIFAR-10 and CIFAR-100. It is naturally expected that output augmentation is an alternative to the traditional methods of data augmentation.

Reference

[1] D. O. Hebb. *The organization of behavior: A neuropsychological theory.* Wiley, New York, June 1949.

[2] W. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 127–147, 1943.

[3] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. pp. 318–362. MIT Press, 1986.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, Vol. 86, pp. 2278–2324, 1998.

[6] S. Eguchi, R. Nakamura, and M. Tanaka. Output augmentation works well without any domain knowledge. In *2021 17th International Conference on Machine Vision and Applications (MVA)*, pp. 1–5, 2021.

[7] A. Kwasigroch, A. Mikołajczyk, and M. Grochowski. Deep convolutional neural networks as a decision support tool in medical problems-malignant melanoma case study. In *W. Mitkowski, J. Kacprzyk, K. Oprzędkiewicz, P. Skruch (eds) Trends in Advanced Intelligent Control, Optimization and Automation. KKA 2017. Advances in Intelligent Systems and Computing, Springer, Cham*, pp. 848–856.

[8] A. Kwasigroch, A. Mikołajczyk, and M. Grochowski. Deep neural networks approach to skin lesions classification a comparative analysis. In

2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR), pp. 1069–1074, 2017.

[9] A. Galdran, A. A. Gila, M. I. Meyer, C. L. Saratxaga, T. Araujo, E. Garrote, G. Aresta, P. Costa, A. M. Mendonça, and A. J. C. Campilho. Data-driven color augmentation techniques for deep skin image analysis. In *ArXiv Prepr. ArXiv170303702*, 2017.

[10] Tensorflow. `https://tensorflow.org/`.

[11] Pytorch. `https://pytorch.org/`.

[12] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Aug. 2019.

[13] D. S. Park, W. Chan, Y. Zhang, C. C. Chiu, B. Zoph, E. D. Cubuk, and Q. V. Le. Specaugment: A simple data augmentation method for automatic speech recognition. In *INTERSPEECH*, 2019.

[14] Y. Tokozume and T. Harada. Learning environmental sounds with end-to-end convolutional neural network. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2721–2725, 2017.

[15] Reiwa gan nen ban joho tsushin hakusho (white paper information and communications in japan). Ministry of Internal Affairs and Communications.

[16] Wikimedia commons. `https://commons.wikimedia.org/`.

[17] M. Taki. *Kore nara wakaru shinsou gakushuu nyuumon (Deep Learning - Machine Learning Primer).* Machine learning professional series. Kodansha Ltd., 2017.

[18] T. Okatani. *Shinsou gakushuu (Deep learning).* Machine learning professional series. Kodansha Ltd., 2015.

[19] C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[20] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.