The basics of neural networks

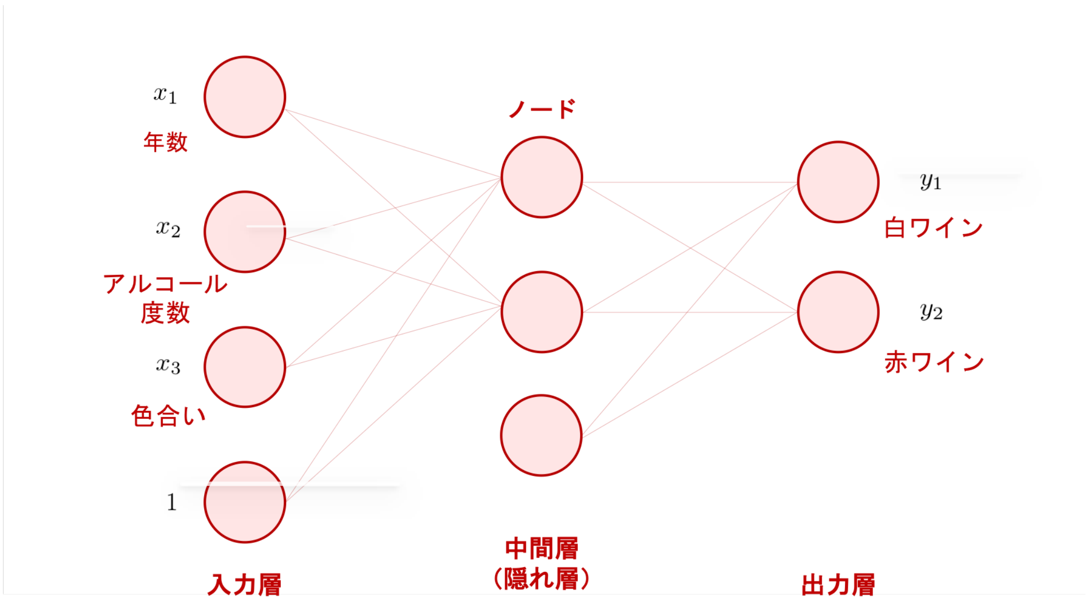
Chapters that need to be read in advance

- Chapter 2 Introduction to Python
- Chapter 4 Basics of Differentiation
- Chapter 5 Fundamentals of Linear Algebra
- Chapter 6 Basics of Probability and Statistics
- Chapter 7 Single regression analysis and double regression analysis
- Chapter 8 Introduction to NumPy

What is a neural network?

A neural network is a **computational graph made by connecting differentiable transformations**. In this chapter, first of all, as shown in the figure below, we will consider that the value is contained in the **node** represented by a circle, and the node and the node are connected by **edge**.

The vertical collection of nodes in this figure is called a **layer**. And **deep learning** refers to machine learning methods using neural networks with a large number of layers and the surrounding research areas.



Layer (layer)

The figure above shows an example of using a neural network to solve a classification problem that predicts the category of whether the wine is "white wine" or "red wine" from some information about wine.

From the left, the first layer is called the **input layer**, the last layer is called the **output layer**, and the **intermediate layer** or the **hidden layer** is called the intermediate layer**. den layer)**. The above neural network has a total of **three layers of structure (architecture; architecture)** because there are input layers, intermediate layers, and output layers. If you have multiple intermediate layers, you can create a more multi-layer neural network.

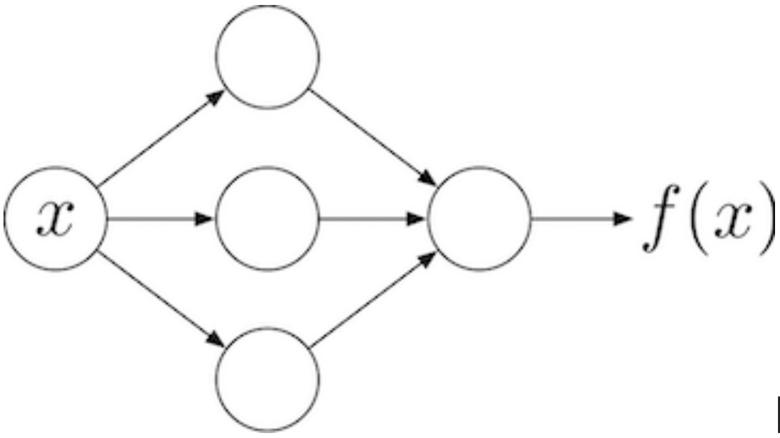
Neural networks convert values from layer to layer. Therefore, it can be thought of as a neural network as **a large function** that can be made by this conversion in a row. Therefore, basically, it receives input and returns some output. And the number of nodes in the input and output layers is determined by what kind of data you want to enter and what kind of output you want to make.

In the above example, the input variableX_1 Number of years,X_2The alcohol content,X_3 Quantitative information for a certain wine, such as color, is given. Therefore, the number of nodes in the input layer is the number of input variables M(In the figure above,$M = 3$) is determined by. Here, the combination between layers and layers of nodes has weights one by one, and **in the case of** a **fully coupled neural network** like the above**, their weights are combined and expressed in a single matrix.** In this case, just like the example of double regression analysis explained in the chapters of single regression analysis and double regression analysis, the bias is always at the end of the input layer because it is treated by including the weight matrix.1Please note that the node with has been added. For this reason, the number of nodes in the input layer in the figure is$M + 1 = 3 + 1 = 4$It is an individual.

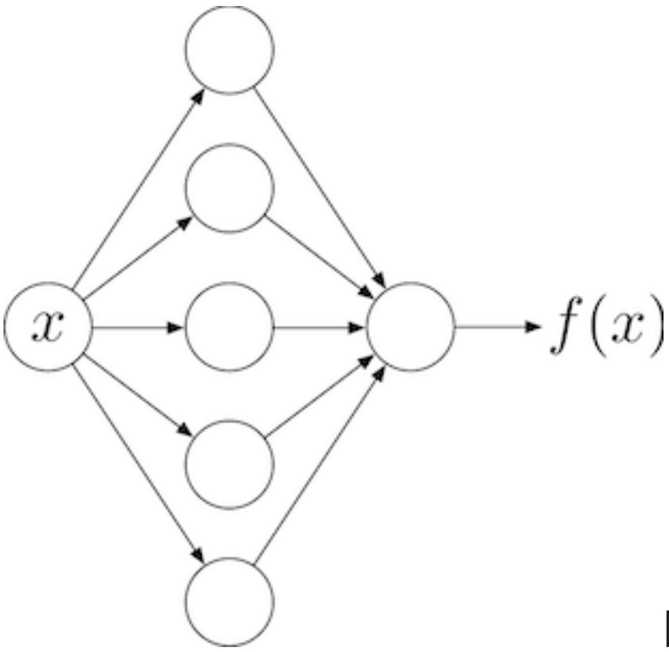
Also, this time, it represents a case where you **enter the information of wine and predict whether it is "red wine" or "white wine",** so it can be said that it is a classification problem with a number of categories of 2. Therefore, the number of nodes in the output layer is 2. If you change the number of nodes in this power layer, you can also respond to the classification problem of 3 or 4 categories. In addition, in the case of regression problems that predict continuous real values rather than discrete categories, the **number of nodes in the output layer** is determined **according to the type of target value**.

Structure (architecture)

As shown in the figure below, even if the number of layers is the same, it is optional whether the number of nodes in the intermediate layer is 3 or 5. How to determine these values is left to the person who designs the neural network. In addition, you need to decide how many layers of the middle layer itself and the number of them. These parameters that you decide by yourself are called **hyperparameters**.



If the number of nodes in the middle layer is 3



If the number of nodes in the middle layer is 5

Various types (type)

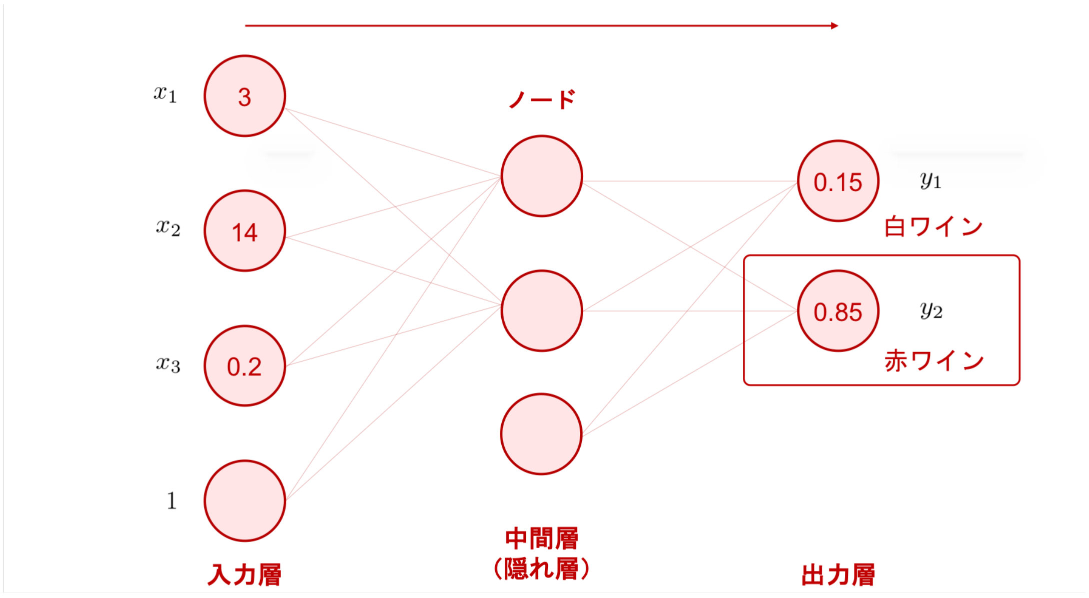
In a neural network, there are not **only fully-connected types** with nodes between layers as shown in the figure above, but also **convolution types** that are often used in image processing, etc. There are many types, such as **al)** and **recurrent,** which are often used for dealing with series data. Also, basically, the difference between them lies in the way they combine between layers, and different types of combination methods can be used between separate layers. In other words, **multiple types of bonds may be mixed in a single neural network.** For example, it is often done to use a convolutional combination near the input and a fully coupled combination near the output.

In this chapter, **we will first explain by all only the most basic all-coined type.** Other types will be explained in a later chapter.

Calculation of neural networks

Flow of basic calculation

Then, when an input in the neural network is given on the subject of "wine classification", how to obtain the calculation results and how to use the results to classify, first of all, without using the formula, understand by looking at the following figure. Sho.



For example, suppose there is a wine with a **year of 3 years, an alcohol content of 14 degrees, and a color of 0.2**.

When the information of this wine was given to the input layer of the neural network, **the calculation proceeded in order** from there to the intermediate layer and the output layer, and finally$Y_1 = 0.15, Y_2 = 0.85$Suppose you have obtained two numbers.

These **two numbers** represent the probability that the entered wine belongs to each of the **two categories**: "red wine" and "white wine". In the classification problem, the **prediction of the neural network** is determined by **which node has the largest value among the output numbers**. This time, out of the two outputs $Y_2 = 0.85$Because it is the largest value,Y_2The corresponding "red wine" is a predicted category.

In this way, when an input is given, each layer of the neural network is calculated in order, and the output is calculated, which is called **forward propagation**.

Now, let's take a closer look at how these calculations are done inside neural networks. The point is that in each layer of the neural network, **"linear transformation" and "nonlinear transformation" are applied to the output value of the previous layer in order**. Now, I will explain what **linear transformation** is and what is **nonlinear transformation** in order.

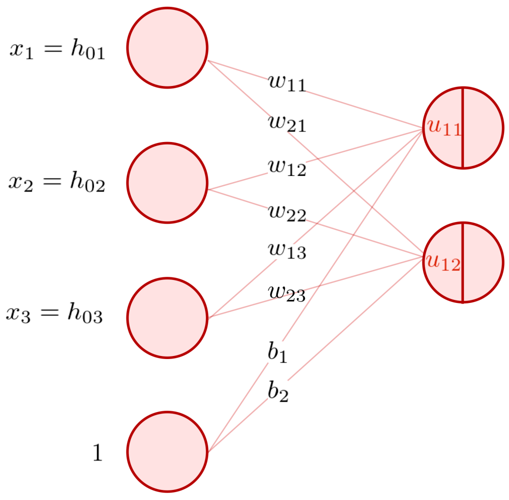
Linear transformation

Linear transformation (Note 1) is the input vector

$\mathbf{H}_0$When ([Note 2]), the weight matrix$\mathbf{W}_{10}$and bias vector$\mathbf{B}_1$Use two of the following conversions.

$$\mathbf{U}_1 = \mathbf{W}_{10}\mathbf{H}_0 + \mathbf{B}_1$$

Based on this, let's take a look at the following two-layer all**-connected neural network** diagram.



This neural network **performs the same calculation as** the **above formula.** Between the two layers, there is a line (edge) connecting nodes. And on the edge,$W_{11}, W_{21}, \ldots$The following characters are written. These represent the **connection weight** between the nodes at both ends.

The value of the node in the input layer is multiplied by the coupling weight and transmitted to the node of the output layer. The calculation results from multiple nodes are transmitted to one node of the output layer, so add all of these together.

Specifically, the following calculations are make.

$$U_{11} = W_{11}H_{01} + W_{12}H_{02} + W_{13}H_{03} + B_1 \quad (1)$$
$$U_{12} = W_{21}H_{01} + W_{22}H_{02} + W_{23}H_{03} + B_2 \quad (2)$$

This formula is the input vector$\mathbf{H}_0$, weight matrix $\mathbf{W}_{10}$, bias vector$\mathbf{B}_1$, and output$\mathbf{U}_1$to

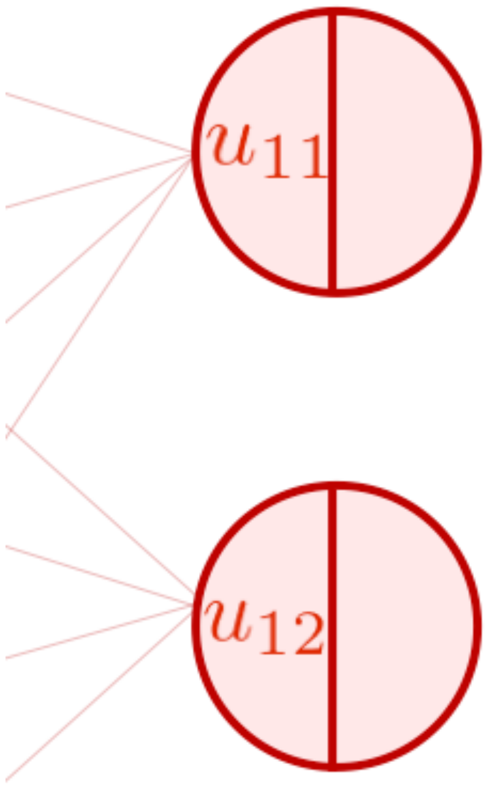
$$\mathbf{H}_0 = \begin{bmatrix} H_{01} \\ H_{02} \\ H_{03} \end{bmatrix} \quad (3)$$
$$\mathbf{W}_{10} = \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \end{bmatrix} \quad (4)$$
$$\mathbf{B}_1 = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \quad (5)$$
$$\mathbf{U}_1 = \begin{bmatrix} U_{11} \\ U_{12} \end{bmatrix} \quad (6)$$

If you define it,

$$\mathbf{U}_1 = \mathbf{W}_{10}\mathbf{H}_0 + \mathbf{B}_1$$

You can write that it is consistent with the formula shown at the beginning of this section.

From the above, it has been found that linear transformations are carried out between layers in fully coupled neural networks. Now, let's look at the nodes of the output layer again. The node is divided into left and right with a line drawn in the middle, and the output value calculated earlier $U_{11}, U_{12}$is written in the semicircle on the left side.



The hidden layer of the neural network **receives**

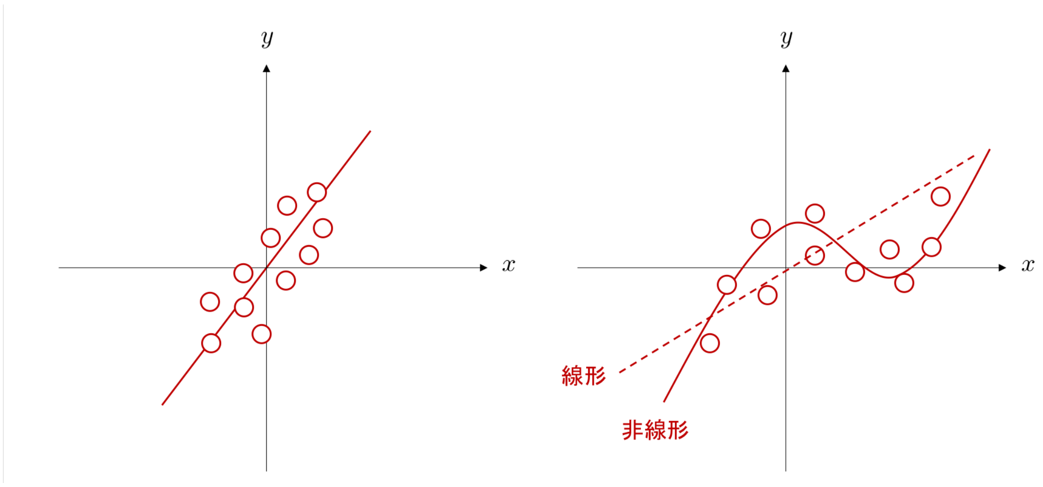
the results of applying a linear transformation to the previous layer, and then outputs the applied nonlinear transformation to it. Write the result in the semicircle on this right.

Next, I will explain what the nonlinear transformation is and why it is necessary.

Nonlinear transformation

Even if **the linear relationship between input and output** as shown in the left of the figure below can be approximated well with the linear transformation introduced in the previous section alone, if the **relationship between input and output** as shown in the right of the figure below **is nonlinear**, the observation data can be successfully approximated. No.

For example, from the observation data represented by the white circle on the right side of the figure below with only linear transformation $X$and$Y$Suppose that if the relationship is approximated, a straight line like a dotted line is obtained. This shows that some data does not apply very well. However, if$XY$If you can express a curve like the solid line on the right side of the figure below on a plane, you may be able to approximate it better.



In order to respond to such cases, in the neural network, nonlinear transformation is applied to each layer following linear transformation, so that the neural network as a whole, which is created by stacking layers, can have nonlinearity. I am.

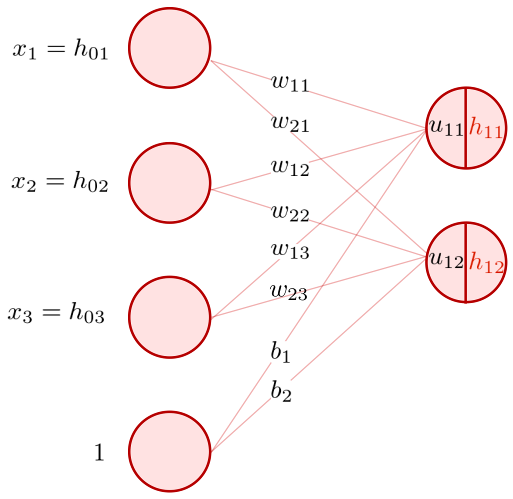
And the function that performs this nonlinear transformation is called an **activation function** in the context of neural networks.

Let's go back to the example used in the linear transformation section and proceed with the explanation. The result of linear transformation U_{11}, U_{12}Activation function for each$A$Perform a non-linear transformation using, and the result

H_{11}, H_{12}I'll leave it. In other words,

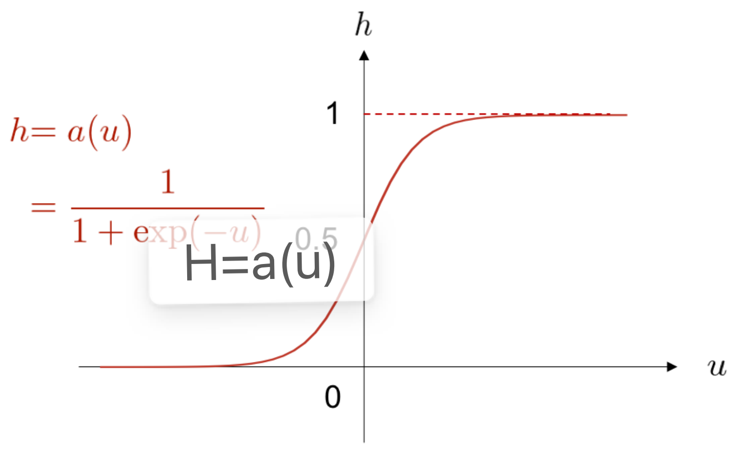
$$H_{11} = A(U_{11}) \tag{7}$$
$$H_{12} = A(U_{12}) \tag{8}$$

It is. These are called **activation values**, and if another layer is connected, they are the input values passed to the next layer.



Logistic sigmoid function

As a specific example of an activation function, the **logistic sigmoid function** shown in the figure below has been often used in the past.

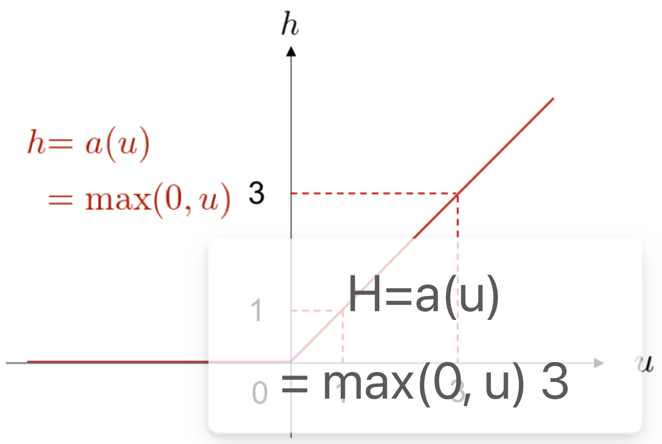


Hereinafter, it is called the **sigmoid function**.

However, in recent years, in neural networks with a large number of layers, sigmoid functions are rarely used as activation functions. One of the reasons for this is that if the sigmaid function is adopted as an activation function, the phenomenon of disappearing gradient **(vanishing gradient)** makes it more likely to cause problems that learning does not progress. The details of this problem will be introduced at the end of this chapter.

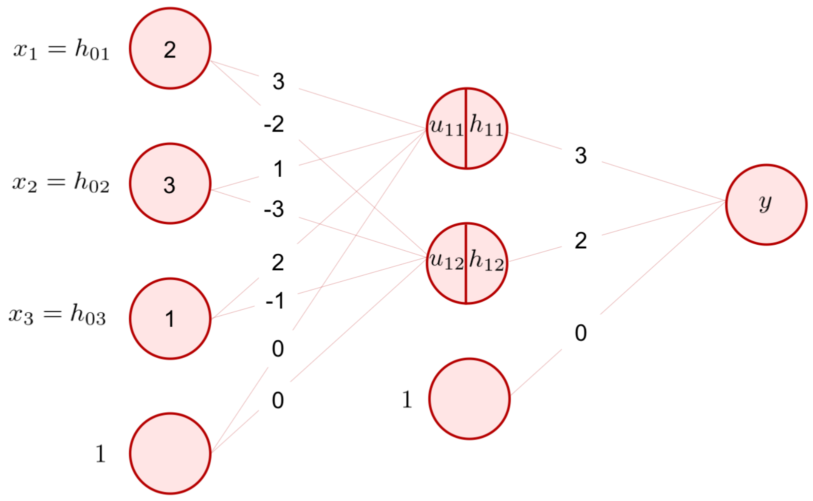
ReLU

To avoid this problem, **normalized linear functions (ReLU: rectified linear units) are** often used these days. This is the following function.



$\mathrm{Max}(0, U)$is,$0$and$U$It is a function that **returns the larger one** by comparing. In other words, **ReLU is** a function that **output is constant at 0 when the input is negative**, and **outputs the input as it is in the case of a positive value**. In the sigmoid function, the input is0If you take a value away from, you can see from the previous figure that the slope (slope) of the curve will gradually become smaller and flatter. On the other hand, if the input value of the ReLU function is positive, no matter how large it is, the slope (slope) is constant. This works effectively for the problem of slope loss that will be introduced later.

Check the flow of the calculation while looking at the numbers



Now, let's consider a three-layer all-joined neural network that solves the regression problem as shown in the figure above, **and let's check the series of calculations from the input is given to the output obtained while actually calculating the numbers by hand.**

As an input,

$$\mathbf{X} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

Let's say it's given. At this time, the output$Y$How is it calculated?

First, the weight matrix$\mathbf{W}_{10}$and bias vector$\mathbf{B}_1$ But the value is as follows from the figure.

$$\mathbf{W}_{10} = \begin{bmatrix} 3 & 1 & 2 \\ \text{-2} & \text{-3} & \text{-1} \end{bmatrix} \tag{9}$$

$$\mathbf{B}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{10}$$

Using this, first perform a linear conversion of the input.

$$\begin{aligned} \mathbf{U}_1 = \mathbf{W}_{10}\mathbf{X} + \mathbf{B}_1 &= \begin{bmatrix} 3 & 1 & 2 \\ \text{-2} & \text{-3} & \text{-1} \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} + \\ &= \begin{bmatrix} 11 \\ \text{-14} \end{bmatrix} \end{aligned}$$

Next, apply a nonlinear transformation. This time, let's adopt the ReLU function as the activation function. Even if the vector is received, the activation function is basically applied to each element.

$$\begin{aligned} \mathbf{H}_1 &= \mathrm{RELU}(\mathbf{U}_1) && (13) \\ &= \begin{bmatrix} \mathrm{Max}(0, U_{11}) \\ \mathrm{Max}(0, U_{12}) \end{bmatrix} && (14) \\ &= \begin{bmatrix} \mathrm{Max}(0, 11) \\ \mathrm{Max}(0, \text{-14}) \end{bmatrix} && (15) \\ &= \begin{bmatrix} 11 \\ 0 \end{bmatrix} && (16) \end{aligned}$$

Similarly, the output layer'sYCalculate up to the value of. Weight matrix between the 2nd and 3rd layers$\mathbf{W}_{21}$The dimension of the output side is1 So,

$$\mathbf{W}_{21} = \begin{bmatrix} 3 & 2 \end{bmatrix}$$

called1×2It's a queue. Bias vector$\mathbf{B}_2$From the figure

$$\mathbf{B}_2 = \begin{bmatrix} 0 \end{bmatrix}$$

I can see that it is. Using these, the output value of the hidden layer$\mathbf{H}_1$If you perform a linear conversion to,

$$Y = \mathbf{W}_{21}\mathbf{H}_1 + \mathbf{B}_2 = 3 \times 11 + 2 \times 0 + 0 \times 1 = 33$$

was sought. Here, we do not use the activation function for the final output layer to explain when to solve regression problems using neural networks (Note 3).

So far, we have looked at the contents of the propagation calculation of neural networks in detail.

Such a calculation process, also known **as inference**, generally refers to making **predictions about new data using a model**

that has been trained. In this example, already $\mathbf{w}_{10}$Ya$\mathbf{w}_{21}$The value of the parameter is given, and we used it to calculate what kind of value the neural network outputs for the input.

In the next section, this$\mathbf{w}_{10}$Ya$\mathbf{w}_{21}$I will explain the **training** of neural networks that determine parameters such as.

Training of neural networks

In the double regression analysis, the expression that comes out by differentiating the objective function with the parameters of the model$= 0$In other hand, leave the variable **without using the actual number** ($\mathbf{X}$Ya$\mathbf{T}$I was able to find the solution (optimal parameter) while it was left. In this way, finding a solution in the state of a variable is called **an analytical solution**, and the answer is called **an analytical solution**.

However, in the case of complex functions such as those expressed in neural networks, it is difficult to solve the optimal solution analytically in most cases. Therefore, you need to think of another way.

For the analytical solution method, it is said to **solve numerically** by performing repeated numerical calculations using a computer to find a solution, and the solution found is called a **numerical solution**.

In neural networks, we basically find the optimal parameters using numerical methods.

And in the typical numerical solution that we are going to introduce, it is necessary to determine the **initial value** of the parameter you want to find in advance. This is because we will update the value little by little from this initial value and bring it closer to the desired solution.

Such a method is a type of thing called the **optimization method**.

1. Design a model
2. Set the objective function
3. Find a parameter that optimizes the objective function

This procedure has been repeated in the chapters of single regression analysis and double regression analysis. And **this procedure often appears in many other machine learning methods.** The neural networks covered in this chapter also follow this procedure.

This time, the model uses the 3-layer neural network used in the previous section to explain. For this reason, step 1 has already been completed. From the next section, we will explain step 2 and beyond.

Objective function

For neural network training, you can use various objective functions according to the task you want to solve as long as it is differentiable. Here,

- **Mean squared error (mean squared error)** often used in **regression problems**
- **Cross entropy,** which is often used in **classification problems**

I will introduce two representative objective functions.

Average square error

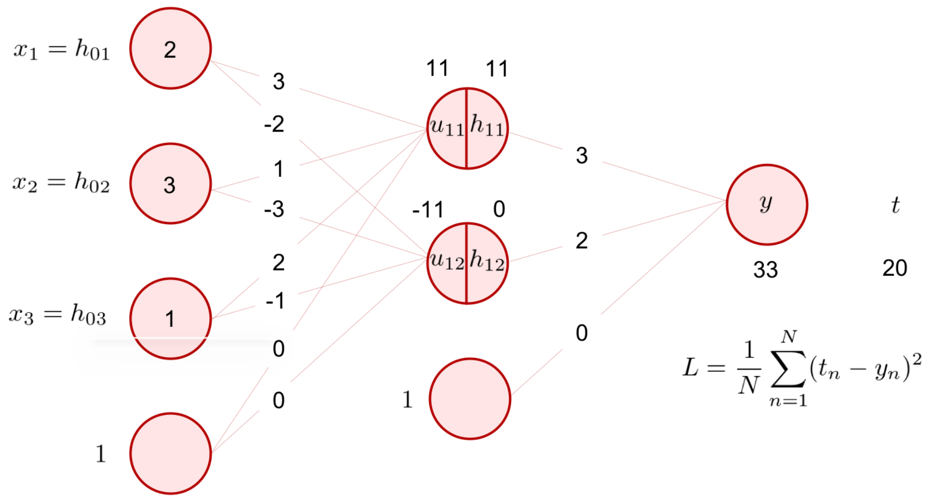
Mean squared error is **an objective function**

that is often used when you want to solve a **regression problem.** It is similar to the square sum error introduced in the explanation of heavy regression analysis, but the difference is that it not only takes the sum of the errors at each data point, but also divides it by the number of data to calculate the average value of the error. If you express it with an expression, it will be as follows.

$$L = \frac{1}{N} \sum_{N=1}^{N} (T_N \text{ - } Y_N)^2$$

Here,$N$is a sample size,$Y_N$is$N$The output value of the neural network for the first data,$T_N$is$N$It is the desired correct answer value for the first data.

Now, let's consider an example shown in the figure below.



Now, an input$\mathbf{X}$The output of this neural network when given$Y$is,33It was. Here, if the target value corresponding to this input is$T = 20$If so, if the overall sample size is 1, the average square error is

$$L = \frac{1}{1}(20 - 33)^2 = 169$$

It can be calculated.

Cross entropy

Cross entropy (cross entropy) is a **purpose function that is often used when you want to solve classification problems.** As an example, KLet's consider the classification problem of the class. A certain input$X$When given, the output layer of the neural network$K$There are nodes, and each of them has this input$K$The probability of belonging to the second class

$$Y_K = P(Y = K|X)$$

Let's say it represents. This is input$X$**Under** the **condition** that is given, **it** means a prediction class$Y$but$K$It is **a conditional probability** that represents a probability such as.

Here,XThe correct answer for the class to which is

$$\mathbf{T} = \begin{bmatrix} T_1 & T_2 & \dots & T_K \end{bmatrix}^{\mathrm{T}}$$

Let's say it's given by a vector called. However, this vector is$T_K \ (K = 1, 2, \dots, K)$ Let's say that **only one of the following is 1, and the other is 0**. This is called **one hot vector (1-hot vector).** And this one element whose value is 1 means that the class corresponding to the index of that element is the correct answer. For example, $T_3 = 1$If so, the third class (class corresponding to the index called 3) is the correct answer.

Using the above, cross-entropy is defined as follows.

$$-\sum_{K=1}^{K} T_K \operatorname{Log} Y_K$$

This is$T_K$but$K = 1, \ldots, K$One of the correct classes$K$Only with the value of1So, it's like the correct class$K$in$\operatorname{Log} Y_K$Take it out-1It's the same as hanging. Also,NConsidering all samples, the cross-entropy is

$$L = -\sum_{N=1}^{N} \sum_{K=1}^{K} T_{N,K} \operatorname{Log} Y_{N,K}$$

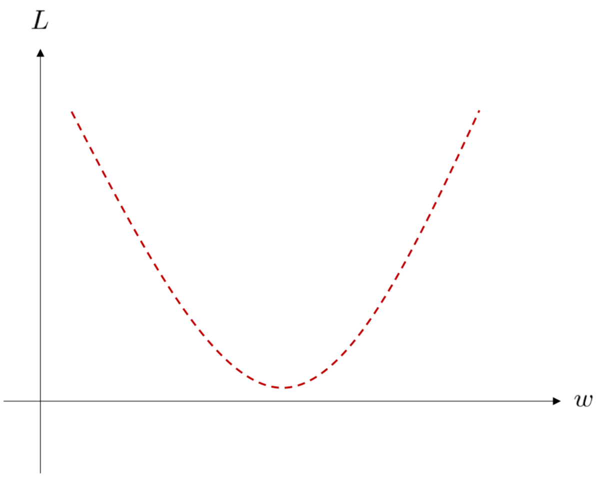
It will be.

Optimization of objective functions

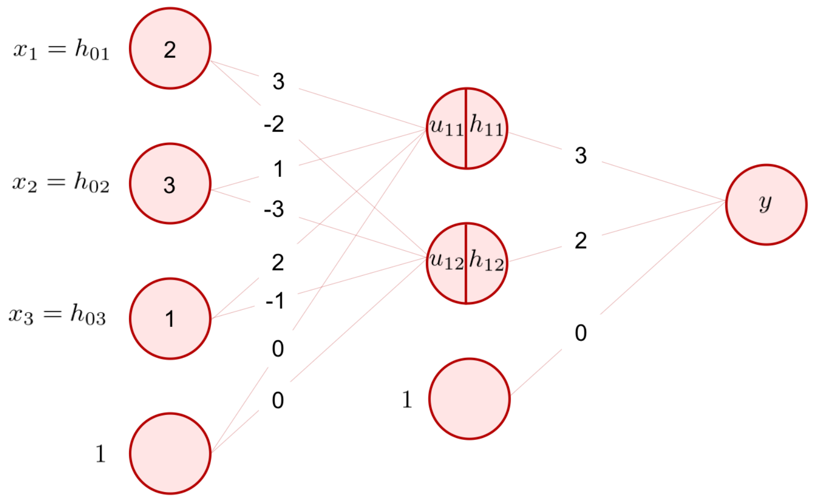
Train a neural network by finding the value of a parameter that minimizes the value of the objective function**.** In this section, we will introduce **the gradient descent method,** one of the optimization algorithms.

Sloping method

The dotted line in the figure is a parameter$W$The objective function when changing$L$It represents the value of. In this diagram, it is represented by a quadratic function for simplicity. Most of the objective functions of actual neural networks are multidimensional and more complex. Now, this objective function gives a minimum value$W$How is it required?



As explained in the previous section, it is necessary to determine the initial value of the parameters of the neural network. In general, **neural network parameters are initialized with random numbers.** Here, as an example $W = 4$Let's think that initialization has been done.



Then,$W = 4$in$L$The slope of the tagent$\frac{\partial L}{\partial W}$is sought. **The slope of** this **tangent** is generally called **gradient**.

Well, here, let's sak$W = 4$in$\frac{\partial L}{\partial W}but5$Let's say it was. about this

$$\left.\frac{\partial L}{\partial W}\right|_{W=4} = 5$$

I will write. This$5$The value is$W = 4$in$L(W)$It represents the gradient of the function.

What is a slope?W **When increasing** L means the **direction of increase**. Now$L$I want to reduce the value of, so **in the opposite direction of** this **slope** W **Change**, that is$W$From$\frac{\partial L}{\partial W}$**If you draw**, it will be **fine**.

This is the basic idea when **updating neural network parameters using the gradient of the objective function**. at this time$W$It is common to multiply the gradient by a value called **learning rate** to adjust the width of the amount of one update. The learning rateηAnd,WFrom$\eta \frac{\partial L}{\partial W}$By subtracting, the value of the gradient itselfηOnly the amount I made$W$It will be updated. After the update$W$is,

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

It will be. Here,$\leftarrow$means replacing the value on the left with the value on the right, that is, **updating**.

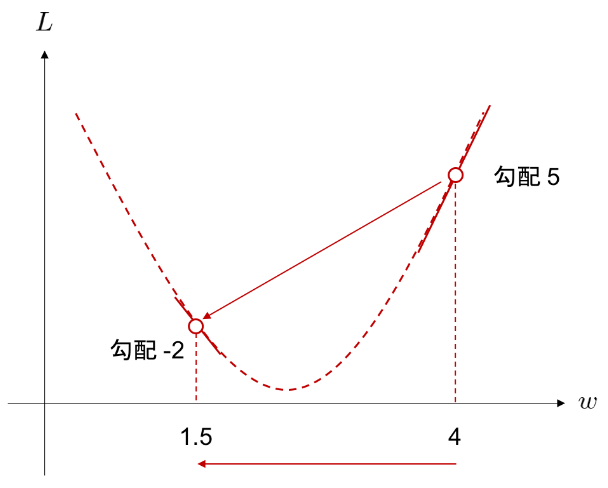
For example, the learning rate$\eta = 0.5$Set it to, after updating$W$Let's actually seek it.WThe amount of updates is the **learning rate** $\times$ Because it is determined by the **slope**,

$$0.5 \times 5 = 2.5$$

It will be. Present$W = 4$So, **subtract this price**

$$W \leftarrow 4 - 2.5$$

After updating,$W = 1.5$It is.



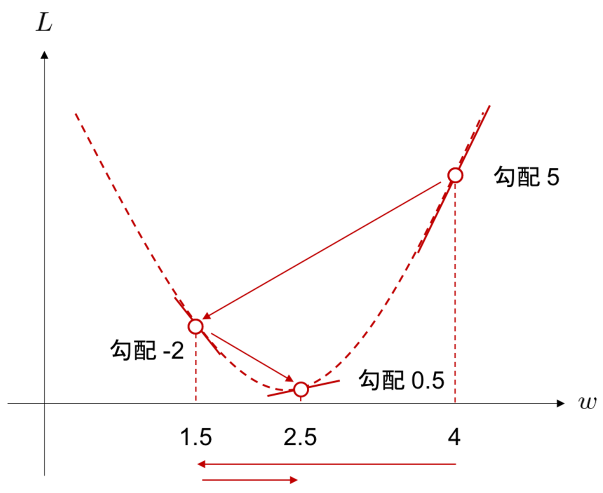
Update for the first time,$W$but$W = 1.5$Moved to the location of. So, let's look for the slope in this point again. This time the slope is-2Let's say it was. Then the **learning rate** $\times$ **The slope** is

$$0.5 \times (\text{-}2) = \text{-}1$$

It will be. Use this again,

$$W \leftarrow 1.5 \text{ - } (\text{-}1)$$

And if you update for the second time, this time $W = 2.5$Come to the position of. After updating twice in this way, the parameter is moved to the position of the figure below.



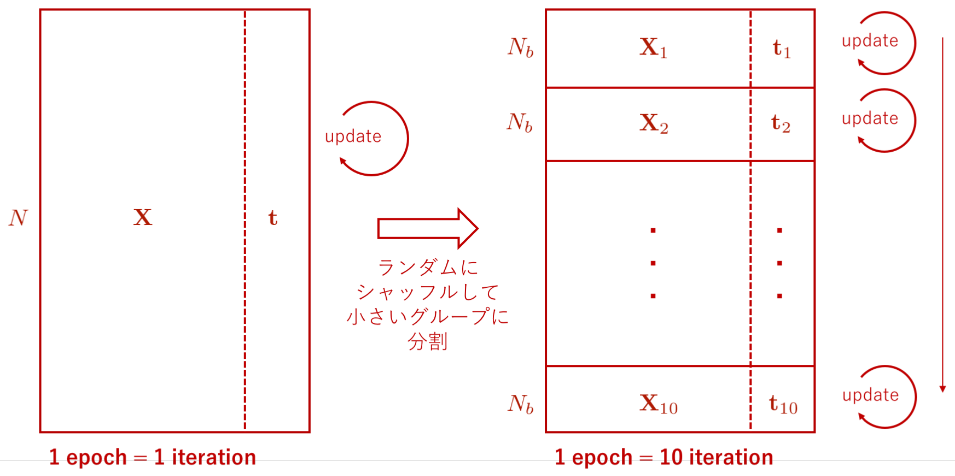
gradually L When the minimum value is W You can see that it is approaching the value of.

In this way, the **learning rate** $\times$ If you change the parameters as the renewal amount of **gradient**, the parameters$W$I want to ask$L$Minimize$W$You can gradually get closer to.

The optimization method of the objective function using such a gradient is called **gradient descent**. Since neural networks are basically **made by connecting only functions that can be differentiated** (Note 4), the functions represented by the entire neural network can also be differentiated, and the objective function is divided by the gradient descent method using the training data set. You can find the best parameter to make it smaller (locally).

Mini batch learning

Normally, when optimizing a neural network with the gradient descent method, instead of using the data one by one to update the parameters, **enter several data together, calculate each gradient, and use the average value of the gradient.** The method **of updating parameters** is often done. This is called **mini-batch learning**.



In mini-batch learning, training is carried out by following the steps below.

1. Uniformly random from the training data set $N_B\ (>0)$Extract individual data
2. ThatN_BPut the data together into the neural network and calculate the value of the

objective function for each data.

3. N_BTake the average value of the objective function
4. Find the gradient of each parameter for the value of this average
5. Update the parameters using the desired gradient

And this is different N_B Repeat for the combination of individual data.

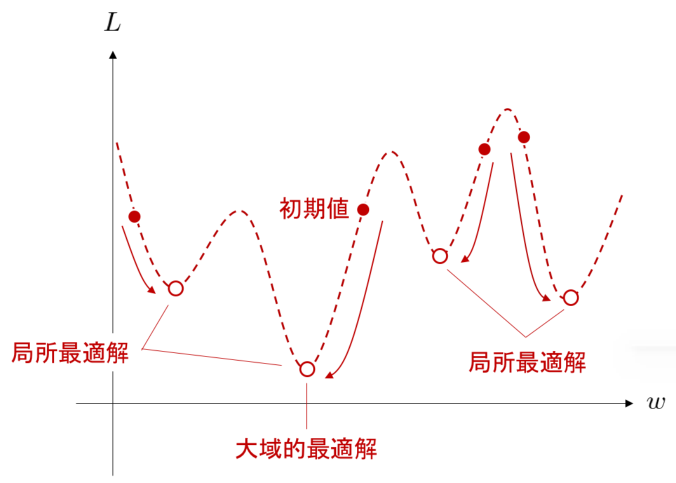
Here, the number of samples used for a single parameter update$N_B$is called **batch size**. As a result, all the data included in the data set will be used, but the data used for a single update isN_B Please note that it is one by one.

From the data setN_BThe diagram above shows the state of mini-batch learning in a schematic way of taking and using data one by one. In this figure, each data set isN_BIt is divided into 10 groups containing data one by one. After that, the parameters will be updated sequentially for each group.

Unlike **mini-batch learning**, the method of using the entire data set at once to update it, that is, not dividing it into groups, is called **batch learning**.

In the gradient descent method, the time until the parameters are updated once using data is called **1 iteration**, and in batch learning, the entire data of the training data set is used for one update at once. **, 1 epoch = 1 iteration**. On the other hand, for example, in the case of mini-batch learning, which data sets are divided into 10 groups and updated for each group, 1 **epoch = 10 iterations** will be.

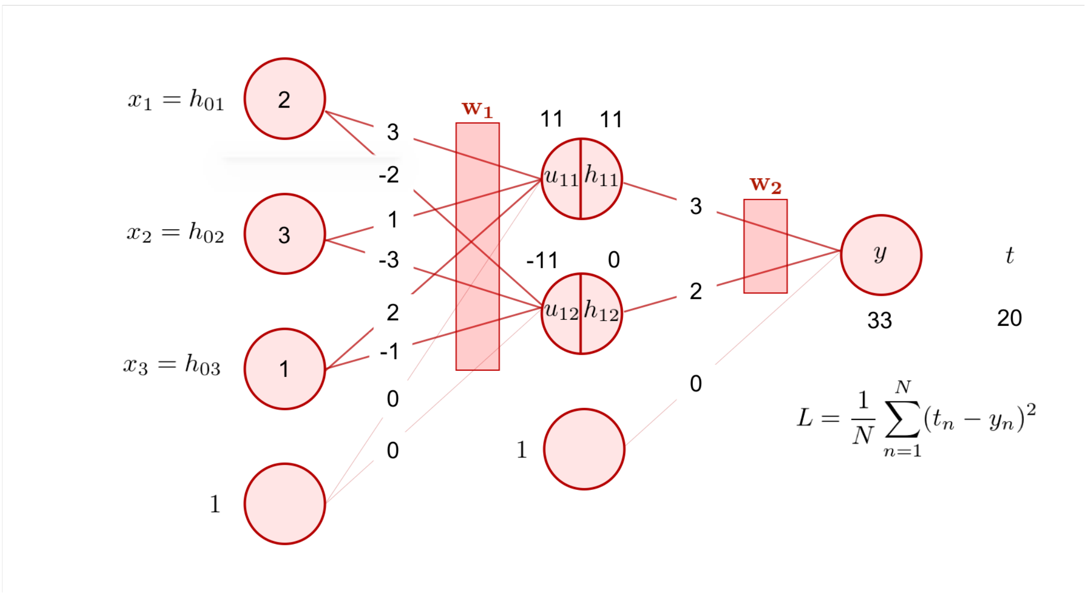
The gradient descent method using such mini-batch learning is especially called the **stocastic gradient descent method (SGD).** Currently, many neural network optimization methods are based on this SGD. Using SGD not only dramatically reduces the overall calculation time, but it is also known that even if the objective function has multiple valleys as shown in the figure below, it will converge to a **local optimal solution** (Note 5) "almost certainly" under appropriate conditions. I'm doing.



Calculation of parameter update amount

Now, let's consider a three-layer all-joined neural network as shown in the figure below, and introduce a formula that updates parameters on the subject of a regression problem that outputs a 1-dimensional value from a 3-dimensional input vector and predicts the correct value. .

First of all, the linear transformation between the first and second layers$\mathbf{W}_1, \mathbf{B}_1$, the linear transformation between the 2nd and 3rd layers $\mathbf{W}_2, \mathbf{B}_2$Let's say it is represented by the parameter. Also, put these together$\boldsymbol{\Theta}$I will express it as. In other words, $\boldsymbol{\Theta} = \{\mathbf{W}_1, \mathbf{B}_1, \mathbf{W}_2, \mathbf{B}_2\}$It is (Note 6).



The input is $\mathbf{X} \in \mathbb{R}^3$, output Y and the target value T I'll leave it. The following mean square error function will be used for the objective function.

$$L = \frac{1}{N}\sum_{N=1}^{N} (T_N - Y_N)^2$$

However, N is the number of data. This time $N = 1$ Therefore, the objective function is easy

$$L = (T - Y)^2$$

I can write.

Now, let's differentiate this objective function by each parameter and calculate the amount of updates for each parameter.

First, calculate the propagation in order. Look at the entire neural network as a function$F$If you decide to write, the output$Y$is

$$\begin{aligned} Y &= F(\mathbf{X}; \mathbf{\Theta}) \\ &= \mathbf{W}_2 A_1(\mathbf{W}_1 \mathbf{X} + \mathbf{B}_1) + \mathbf{B}_2 \end{aligned}$$

You can write. Here,A_1means a nonlinear transformation (the activation function used) performed on the node of the first layer. This time, we will use the **sigmoid function** for this activation function.

Below, for simplicity, the results of the linear conversion performed on the input$\mathbf{U}_1$And the value of the middle layer, that is$\mathbf{U}_1$The result of applying the activation function to$\mathbf{H}_1$I will write. Then, these relationships can be organized as follows.

$$\begin{aligned} \text{([illegible])} \quad \mathbf{U}_1 &= \mathbf{W}_1 \mathbf{X} + \mathbf{B}_1 \\ \text{(1[illegible])} \quad \mathbf{H}_1 &= A_1(\mathbf{U}_1) \\ \text{(1[illegible])} \quad Y &= \mathbf{W}_2 \mathbf{H}_1 + \mathbf{B}_2 \end{aligned}$$

Parameters$\mathbf{W}_2$The amount of updates

First of all, the parameters that are closer to the output layer,$\mathbf{W}_2$about$L$Let's find the slope of. Since this is a partial derivative of a synthetic function, it can be expanded as follows using **the chain rule**.

$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{W}_2}$$

The product of two partial derivatives appeared. Each of these,

$$\frac{\partial L}{\partial Y} = \text{-}2(T\text{-}Y)$$
$$\frac{\partial Y}{\partial \mathbf{W}_2} = \mathbf{H}_1$$

I ask for it.

Now, let's actually use NumPy to calculate the value of this gradient. Here, for simplicity, let's say that all bias vectors are initialized at 0.

まず、NumPy モジュールを読み込んでから、入力の配列を定義します。 ここでは、上図と同じになるように `2, 3, 1` の3つの値を持つ3次元ベクトルを定義しています。 また、正解として図と同じように `20` を与えることにしました。 次に、パラメータを定義します。

Here, we have defined the following four parameters.

Parameters of linear transformation between the first layer and the second layer

$\mathbf{W}_1 \in \mathbb{R}^{2 \times 3}$: A matrix that converts a 3-dimensional vector into a 2-dimensional vector

$\mathbf{B}_1 \in \mathbb{R}^{2}$: 2-dimensional bias vector

Parameters for linear transformation between the 2nd and 3rd layers

$\mathbf{W}_2 \in \mathbb{R}^{1 \times 2}$: A matrix that converts a 2-dimensional vector into a 1-dimensional vector

$\mathbf{B}_2 \in \mathbb{R}^{1}$: 1-dimensional bias vector

Now, let's actually perform the calculation of each layer.

```
[2.99995156]
```

The output is2.99995156I asked for it. In other words,$F([2, 3, 1]^T) = 2.99995156$That means. Next, I derived it from above

$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial \mathbf{W}_2}$$

Let's calculate the two partial derivatives on the right side of respectively.

If you multiply these, the parameters you wanted to find$\mathbf{W}_2$You can get the slope of.

```
[-3.39995290e+01 -2.82720335e-05]
```

The slope was sought. This is$\frac{\partial L}{\partial \mathbf{W}_2}$It is the value of (Note 7). If you use this scaled by the learning rate, the parameters$\mathbf{W}_2$You can update. The update formula is specifically as follows.

$$\mathbf{W}_2 \leftarrow \mathbf{W}_2 \text{ - } \eta \frac{\partial L}{\partial \mathbf{W}_2}$$

About the learning rate

If the learning rate is too high, the value of the objective function will oscillate or divergent while updating the parameters repeatedly. On the contrary, if it is too small, it will take time to converge. Therefore, it is very important to properly determine this learning rate in neural network learning. In many cases, it is done to empirically find the largest value that progresses learning properly. In simple image recognition tasks, usually,$0.1$From$0.01$It is relatively common to see that the value of the degree is tested first.

Parameters$\mathbf{W}_1$The amount of updates

Next,$\mathbf{W}_1$Let's also find the amount of updates. To do that,$\mathbf{W}_1$The objective function at$L$A partially differentiated value is required. This can be calculated as follows.

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_1} &= \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{W}_1} \\ &= \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{H}_1}\frac{\partial \mathbf{H}_1}{\partial \mathbf{W}_1} \\ &= \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial \mathbf{H}_1}\frac{\partial \mathbf{H}_1}{\partial \mathbf{U}_1}\frac{\partial \mathbf{U}_1}{\partial \mathbf{W}_1}\end{aligned}$$

It became the product of four partial derivatives. Here, the gradient of the activation function

$$\frac{\partial \mathbf{H}_1}{\partial \mathbf{U}_1}$$

is appearing. This time, the activation functionA_1 Since the sigmoid function is used,

$$\mathbf{H}_1 = A_1(\mathbf{U}_1) = \frac{1}{1 + \mathrm{Exp}(\text{-}\mathbf{U}_1)}$$

to$\mathbf{U}_1$Differentiate with.

$$
\begin{aligned}
\frac{\partial A_1(\mathbf{U}_1)}{\partial \mathbf{U}_1} &= - \frac{-(\mathrm{Exp}(-\mathbf{U}_1))}{(1 + \mathrm{Exp}(-\mathbf{U}_1))^2} \\
&= \frac{1}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \cdot \frac{\mathrm{Exp}(-\mathbf{U}_1)}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \\
&= \frac{1}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \cdot \frac{1 + \mathrm{Exp}(-\mathbf{U}_1) - 1}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \\
&= \frac{1}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \left(1 - \frac{1}{1 + \mathrm{Exp}(-\mathbf{U}_1)} \right) \\
&= A_1(\mathbf{U}_1)\,(1 - A_1(\mathbf{U}_1)) \\
&= \mathbf{H}_1(1 - \mathbf{H}_1)
\end{aligned}
$$

In this way, **the gradient of the sigmoid function can be easily calculated using the output value of the sigmoid function.**

Now, of the four partial derivatives above, the first one and the third one have already been found. The other two, each,

$$
\begin{aligned}
\frac{\partial Y}{\partial \mathbf{H}_1} &= \mathbf{W}_2 \\
\frac{\partial \mathbf{U}_1}{\partial \mathbf{W}_1} &= \mathbf{X}
\end{aligned}
$$

It can be calculated. Now, let's actually use NumPy to perform the calculation right away.

```
[[-3.40704286e-03 -5.11056429e-03 -
1.70352143e-03] [-1.13088040e-04 -1.
69632060e-04 -5.65440200e-05]
```

This is $\frac{\partial L}{\partial \mathbf{W}_1}$It is the value of (Note 7). Using this, $\mathbf{W}_2$Similarly, the parameters are updated with the following formula$\mathbf{W}_1$You can update.

$$\mathbf{W}_1 \leftarrow \mathbf{W}_1 - \eta \frac{\partial L}{\partial \mathbf{W}_1}$$

With this$\mathbf{W}_1$I was able to update.$\mathbf{B}_1$,$\mathbf{B}_2$You can also update it in the same way.

Error reverse propagation method (back propagation)

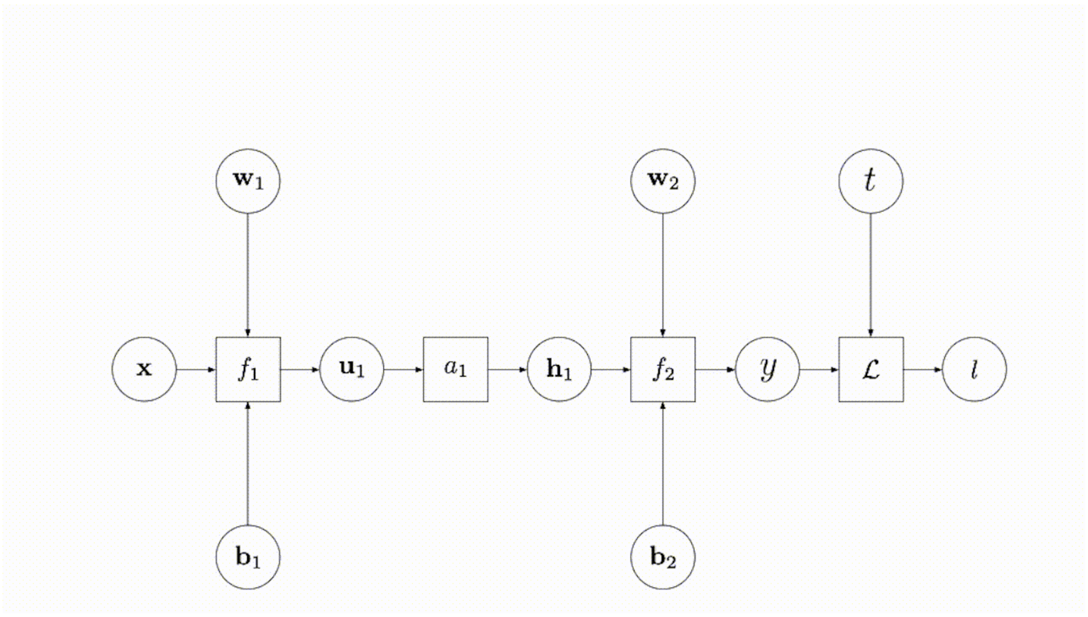
So far, I have experienced that the derivative of the objective function for each parameter is derived by manual calculation and the actual numerical calculation of the slope is performed. So, what happens in the case of neural networks with a larger number of layers?

Similarly, it may not be impossible to find the slope of all parameters by manual calculation, but if the number of layers increases, it will take a lot of time. However, **using the property of applying the differentiable functions of the**

neural network in order, it is possible for the computer to **derive a function that automatically gives a gradient**.

First of all, let's remember that the partial derivative of a synthetic function can be **transformed into the form of the product of multiple partial derivatives by** the **chain law**.

The figure below shows the calculation (**propagation**) to obtain the output of the 3-layer all-connected neural network used in the explanation so far, and the process of calculating the value of the objective function using that value with the blue arrow, and **each pa done by manual calculation in the previous section.** This is a video that expresses **the process of calculating the partial derivative of the objective function by the rameter with a red arrow**.



First, the output of the objective function $L = L(Y, T)$I will do it. The round node in this figure represents the variable, and the square node represents the function.

Now, the entire neural network seen as a huge synthetic function$F$It is expressed, and the function used for linear transformation between each layerF_1,F_2, nonlinear transformation in the middle layerA_1It is expressed as.

Now, as shown by the blue arrow in the figure above, a new input$\mathbf{X}$is given to the neural network, which is transmitted to the output side in order, and finally the value of the objective function$L$Let's say the calculation is finished until.

Then, the next step is to find the update amount of each parameter that reduces the output value of the objective function, but the **gradient of the objective function required for** this **is the gradient of the function in the part (output side) before the round node of each parameter.** I can see that you can **calculate by ke**. Specifically, it is a combination of all of them.

In other words, as shown by the red arrow in the figure above, if you look for the gradient of the input in each function **from the output side to the input side**, **in the opposite direction of propagation**, and multiply it, you can calculate the gradient of the objective function for the parameter. It is.

In this way, using the mechanism of the differential chain law, the gradient of the objective function for the parameters of the function that makes up the neural network is calculated by **following the path passed by propagation in** the **opposite direction by** multiplying the slope of the function in the middle. The algorithm is called **backpropagation**.

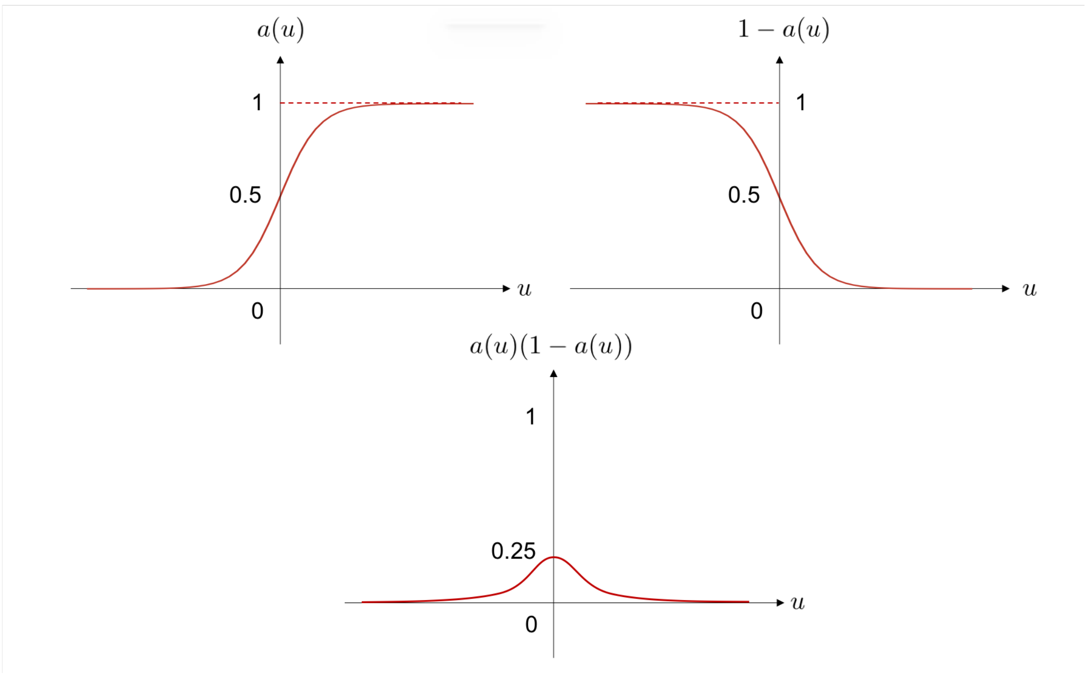
Slope disappearance

When I first mentioned the activation function, I explained that the sigmoid function has a problem that the phenomenon of gradient loss is more likely to occur, and it is not used much at present. Let's take a closer look at the reason.

I will try to recall the derivative of the sigmoid function that I have already calculated above.

$$A\left(U\right) = \frac{1}{1 + \mathrm{Exp}(\text{-}U)}$$
$$A^{'}\left(U\right) = A\left(U\right)\left(1 \text{ - } A\left(U\right)\right)$$

Now, if you try to plot the value of this derivative with respect to the input variable, it will be as follows.



The two above in this figure are the two parts that make up the derivative,$F(U)$and1 - $F(U)$The values are plotted separately, and the figure at the bottom center is the value of the entire derivative. As is clear from the shape of the derivative at the bottom center, it can be seen that the value of the slope becomes smaller and smaller as the input becomes farther from the origin, and it is asymptiate to 0.

To find the update amount for each parameter, it was necessary **to multiply the gradient of all functions ahead of that parameter**, as explained in the previous section. At this time, if the sigmaid function is used for the activation function, the gradient will always be only a **maximum** value of **0.25**. In other words, every time a sigmoid function appears in a neural network, the gradient of the objective function is multiplied by at **most** 0.25. As the number of layers increases, the value that can only be 0.25 at this maximum will be multiplied repeatedly, and the gradient that flows to the layer close to the input will get closer to 0.

Let's look at a specific example. This time, I was explaining using a 3-layer neural network, but I will consider the case of a 4-layer. Then, the gradient of the linear transformation parameter closest to the input is at most the gradient of the objective function.$0.25 \times 0.25 = 0.0625$It means that it is doubled. It is clear that every time the number of layers increases by one, the slope decreases exponentially.

In deep learning, neural networks that accumulate more layers than 4 layers are used. Then, if you use a sigmoid function as an activation function, the **gradient of the objective function will almost completely no longer be transmitted to the parameters of the function that is close to the input**. If only a gradient is too small, the amount of parameter updates will almost be 0, so no matter how large the value of the objective function is, the parameters of the function close to the input layer will not change. In other words, the value will hardly change from the time of initialization, and learning will not be carried out. This was called **gradient disappearance**, and it was one of the factors that made it difficult to learn neural networks that had been deep (over a dozen layers) for a long time.

As a solution, the ReLU function is proposed, and learning for multi-layered neural networks can also be done while avoiding gradient loss.

Note 1
In mathematics, linear transformation is usually $\mathbf{W}$Refers to the operation of multiplying,$\mathbf{B}$The operation of adding is not included.$\mathbf{B}$The operation that involves addition is strictly called "afine transformation (or affine transformation)". However, in the context of deep learning, this transformation is often called linear transformation.

▲Back to the top

Note 2
Here, the node of the input layer H_{01}, H_{02}, H_{03}It is represented by the character, but this$H$comes from h, the initials of **hidden layer in** English, which means **hidden**

layer. The input layer is not a hidden layer, but by using the same characters, the notation is generalized and simplified.

▲Back to the top

Note 3
Also, if you want to solve the classification problem, prepare a node that is as large as the number of classes in the output layer, and represent the probability that the input belongs to the class where each node is located. For this reason, the **total value of all output nodes is normalized to 1.** This requires the use of another function that calculates the activity value for each **layer**, not the activation function applied to each **element**. A typical function used for such purposes is the Softmax function.

▲Back to the top

Note 4
Strictly speaking, there is a possibility that there is a point that cannot be differentiated in

the objective function. For example, ReLU is $X = 0$Because it is indiferiated, there is a point that cannot be differentiated in neural networks containing ReLU. In this case, the idea of **subderivative (subdifferential)** is introduced so that the gradient can be determined at all points.

▲Back to the top

Note 5
The conditions in which this local solution matches the global optimal solution are currently actively researched. Reference: "A Convergence Theory for Deep Learning via Over-Parameterization"

▲Back to the top

Note 6
{}The symbol represents a set and writes with elements lined up inside.

▲Back to the top

Note 7
Now, let's try to calculate the same gradient using Chainer and check if the result matches the code using NumPy.

▲Back to the top

```
variable([[2.9999516]]) [variable([[
-3.3999527e+01, -2.837189191e-05]])]
[variable([[-3.4045703e-03, -5.10685
52e-03, -1.7022851e-03], [-1.1348747
e-04, -1.7023120e-04, -5.6743735e-0
5]])]
```