UNIT-I INTRODUCTION TO DEEP LEARNING

Perceptrons to Neural Networks - Activation Function - Calculating Multidimensional Arrays - Implementing a Three-Layer Neural Network - Designing the Output Layer - Identity Function and Softmax Function - Handwritten Digit Recognition. Neural Network Training: Learning from Data – Loss Function. CHAPTER – 3 & 4 (T1)

What Is a Perceptron?

A perceptron receives multiple signals as inputs and outputs one signal.

The signal in a perceptron is binary: "Flow (1) or Do not flow (0)."

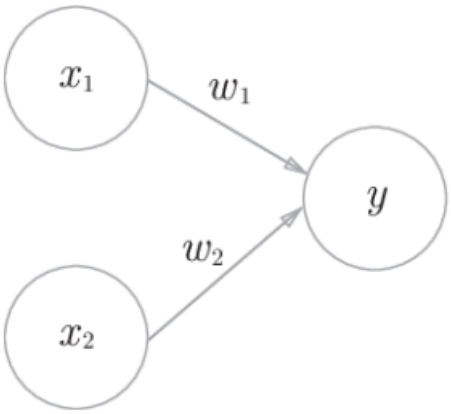


Fig 1.1 Perceptron with two inputs

In Fig 1.1

x1 and x2 are input signals

y is an output signal, and

w1 and w2 are weights

The circle in the preceding diagram is called a "neuron" or a "node."

When input signals are sent to a neuron, each of them is multiplied by its own weight (w1 x1 and w2 x2). The neuron sums the signals that it receives and outputs 1 when the sum exceeds a certain limit value. This is sometimes called "firing a neuron." Here, the limit value is called a **threshold** and is represented by the θ symbol. Below Eq:1.1 describes the same

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases}$$

A perceptron has a specific weight for each of multiple inputs, while the weight controls the importance of each signal. The larger the weight, the more important the signal for the weight.

From Perceptrons to Neural Networks

A neural network is similar to the perceptron.

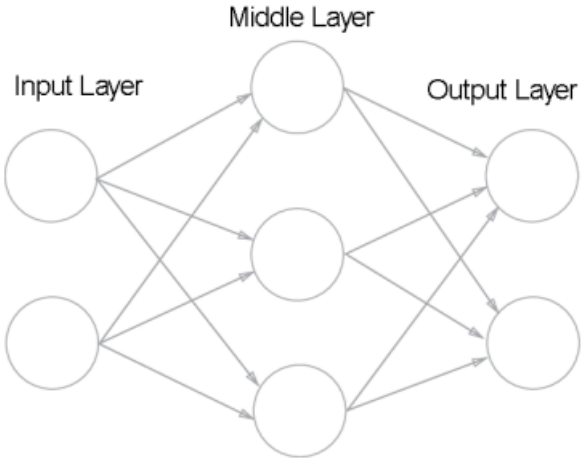


Fig 1.2 Neural network example

The left column is called an input layer, the right column is called an output layer, and the center column is called the middle layer. The middle layer is also known as a hidden layer. "Hidden" means that the neurons in the hidden layer are invisible.

How are signals transmitted in a neural network?

First need to review the perceptron.

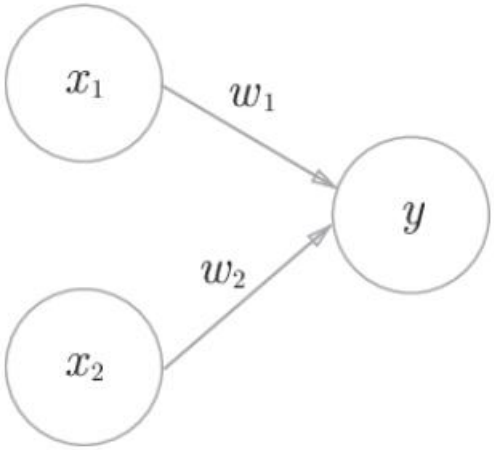


Fig 3.3 Reviewing the perceptron

$$y = \begin{cases} 0 & (b + w_1 x_1 + w_2 x_2 \leq 0) \\ 1 & (b + w_1 x_1 + w_2 x_2 > 0) \end{cases}$$

Eq: 1.2

Here b is a parameter called "bias" and controls how easily the neuron fires.

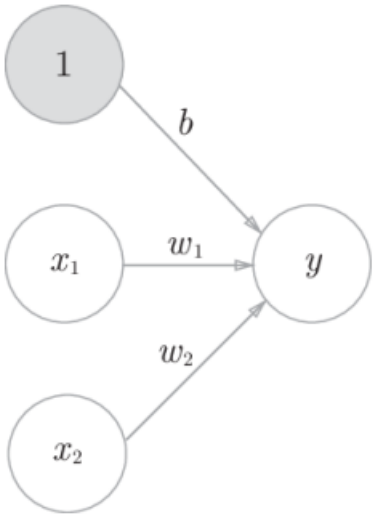


Fig 1.4 Showing the bias explicitly

A signal of weight b and input 1 has been added.

This perceptron receives three signals (x1, x2, and 1) as the inputs to the neuron, and multiplies the signals by each weight before transmitting them to the next neuron. The next neuron sums the weighted signals and then outputs 1 if the sum exceeds 0. It outputs 0 if it doesn't.

Simplifying Eq 1.2

$$y = h(b + w_1x_1 + w_2x_2)$$

Eq 1.3

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

Eq 1.4

Equation (1.3) indicates that the h(x) function converts the sum of input signals into the output, y. The h(x) function represented by equation (1.4) returns 1 if the input exceeds 0 and returns 0 if it does not.

Introducing an Activation Function

The h(x) function that appears here is generally called an activation function. It converts the sum of input signals into an output signal. As the name "activation" indicates, the activation function determines how the sum of the input signals activates. The weighted input signals are summed, and the sum is converted by the activation function which is shown in below Eq.1.5 and 1.6.

$$a = b + w_1x_1 + w_2x_2$$

$$y = h(a)$$

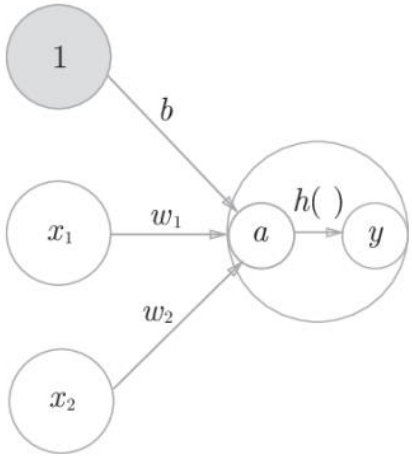


Fig 1.5 Showing the process performed by the activation function explicitly

In the above fig , circles a and y are called "nodes," which are used in the same sense as "neurons"

Fig 1.6 The left-hand image is an ordinary image that shows a neuron, while the right-hand image explicitly shows the process of activation in a neuron (a is the sum of input signals, h() is the activation function, and y is the output)

The activation function, which serves as the bridge from a perceptron to a neural network.

Activation Function

The activation function represented by eq:1.4 changes output values at a threshold and is called a "step function" or a "staircase function." Therefore, we can say, "a perceptron uses a step function as the activation function."

Sigmoid Function

One of the activation functions often used in neural networks is the **sigmoid function**, represented by below equation

$$h(x) = \frac{1}{1 + \exp(-x)}$$

exp(-*x*) in the above equation indicates e^{-X}. The real number, *e*, is Napier's number, 2.7182... The sigmoid function represented seems complicated, but it is only a "function." A function is a converter that returns output when input is provided.

For example, when a value such as 1.0 and 2.0 is provided to the sigmoid function, values such as *h*(1.0) = 0.731… and *h*(2.0) = 0.880… are returned.

Activation is used to convert signals, and the converted signals are transmitted to the next neuron. In fact, the main difference between the perceptron and the neural network is the activation function.

Implementing a Step Function

```
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

But the above function only takes a real number (a floating-point number) as argument **x**. Therefore, **step_function(3.0)** is allowed.

However, the function cannot take a NumPy array as the argument. Thus, **step_ function(np.array([1.0, 2.0]))** is not allowed.

To take a NumPy array the following code is used

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)
```

The above code is little difficult to understand. The code can be rewritten as

```
>>> import numpy as np
>>> x = np.array([-1.0, 1.0, 2.0])
>>> x
array([-1., 1., 2.])
>>> y = x > 0
>>> y
array([False, True, True], dtype=bool)
```

The **y** array is Boolean, and the desired step function must return **0** or **1** of the **int** type. Therefore, convert the type of elements of array **y** from Boolean into **int**:

```
>>> y = y.astype(np.int)
>>> y
array([0, 1, 1])
```

astype() method is used to convert the type of the NumPy array. **True** is converted into **1**, and **False** is converted into **0** by converting the Boolean type into the int type.

Step Function Graph

```
import numpy as np
import matplotlib.pylab as plt

def step_function(x):
    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # Specify the range of the y-axis
plt.show()
```

np.arange(-5.0, 5.0, 0.1) generates a NumPy array containing values from **-5.0** to **5.0** in **0.1** steps, **([-5.0, -4.9, …, 4.9]).**

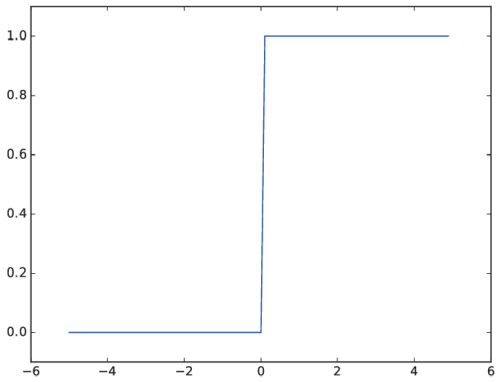


Fig 1.7 Step function graph

Implementing a Sigmoid Function

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Here np.exp(-x) corresponds to **exp(−x)** in the equation. This implementation is not very difficult. The correct results are returned even when a NumPy array is provided as the **x** argument. When this sigmoid function receives a NumPy array, it calculates correctly, as shown below

```
>>> x = np.array([-1.0, 1.0, 2.0])
>>> sigmoid(x)
array([0.26894142,  0.73105858,  0.88079708])
```

The implementation of the sigmoid function supports a NumPy array due to NumPy's broadcasting When an operation is performed on a scalar and a NumPy array, the operation is performed between the scalar and each element of the NumPy array.

```
>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([2., 3., 4.])
>>> 1.0 / t
array([1.  ,  0.5  ,  0.33333333])
```

Sigmoid Function Graph

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # Specify the range of the y-axis
plt.show()
```

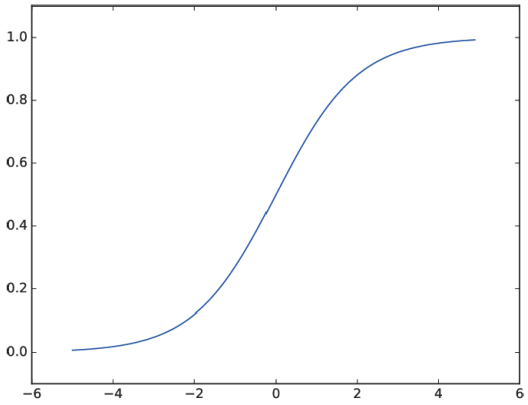


Fig 1.8 Graph of the sigmoid function

Comparing the Sigmoid Function and the Step Function

The sigmoid function is a smooth curve, where the output changes continuously based on the input. On the other hand, the output of the step function changes suddenly at **0**. This smoothness of the sigmoid function has an important meaning when training neural networks:

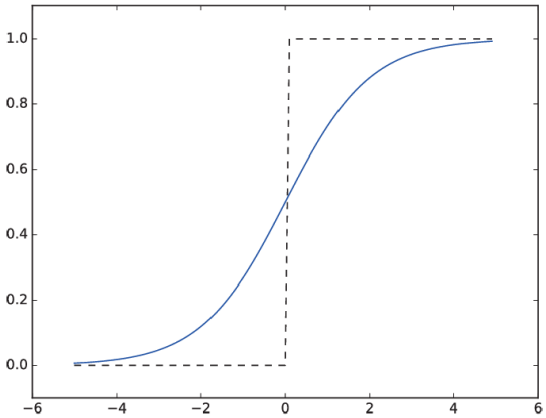


Fig 1.9 Step function and sigmoid function (the dashed line shows the step function)

The step function returns only 0 or 1, while the sigmoid function returns real numbers such as 0.731... and 0.880... That is, binary signals of 0 and 1 flow among neurons in a perceptron, while signals of continuous real numbers flow in a neural network.

Similarities:
Both of them output a value near/ of 0 when the input is small, and, as the input becomes larger, the output approaches/ reaches 1. The step and sigmoid functions output a large value when the input signal contains important information and output a small value when it don't. They are also similar in that they output a value between 0 and 1, no matter how small or large the value of the input signal is.

Nonlinear functions. The sigmoid function is represented by a curve, while the step function is represented by straight lines that look like stairs. They are both classified as nonlinear functions.

Why may a linear function not be used? The reason is that increasing the number of layers in a neural network becomes useless if a linear function is used.

The problem with a linear function is caused by the fact that a "network without a hidden layer" that does the same task always exists, no matter how many layers are added.
let's consider a simple example. Here, a linear function, $h(x) = cx$, is used as the activation function and the calculation of $y(x) = h(h(h(x)))$ is performed as in a three-layer network. It contains multiplications of y(x) = c×c×c×x, and the same operation can be represented by one multiplication of $y(x) = ax$ (where $a = c3$). Thus, it can be represented by a network without a hidden layer. As this example shows, using a linear function offsets the advantage of multiple layers. Therefore, to take advantage of multiple layers, a nonlinear function must be used as the activation function.

Rectified Linear Unit (ReLU) Function
If the input exceeds 0, the ReLU function outputs the input as it is. If the input is equal to or smaller than 0, it outputs 0.

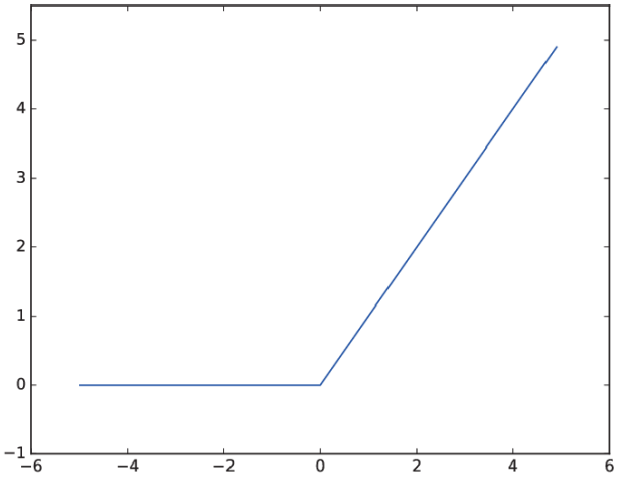


Fig 1.10 **ReLU function**

Below is the equation for ReLU

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

```
def relu(x):
    return np.maximum(0, x)
```

Here, NumPy's maximum function is used. It outputs the larger of the input values.

Calculating Multidimensional Arrays

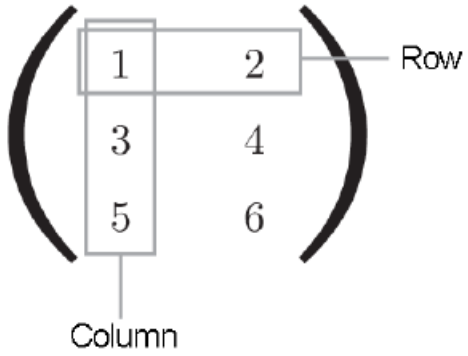
Multidimensional Arrays
A multidimensional array is "a set of numbers" arranged in a line, in a rectangle, in three dimensions, or (more generally) in N dimensions, called a multidimensional array.

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4])
>>> print(A)
[1 2 3 4]
>>> np.ndim(A)
1
>>> A.shape
(4,)
>>> A.shape[0]
4
```

np.ndim() function to obtain the number of dimensions of an array.
Use the instance variable, **shape**, to obtain the shape of the array.

Let's create a two-dimensional array:

```
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> print(B)
[[1 2]
[3 4]
[5 6]]
>>> np.ndim(B)
2
>>> B.shape
(3, 2)
```



Matrix Multiplication

```
>>> A = np.array([[1,2], [3,4]])
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

For example, the product of 2x3 and 3x2 matrices can be implemented in Python as follows:

```
>>> A = np.array([[1,2,3], [4,5,6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> B.shape
(3, 2)
>>> np.dot(A, B)
array([[22, 28],
       [49, 64]])
```

Note: careful about the "shapes of matrices."

```
>>> C = np.array([[1,2],  [3,4]])
>>> C.shape
(2, 2)
>>> A.shape
(2, 3)
>>> np.dot(A, C)
```

ERROR: Traceback (most recent call last):

```
    File "<stdin>", line 1, in <module>
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

The number of elements in the corresponding dimensions must be the same, even when A is a two-dimensional matrix and B is a one-dimensional array. Below code works for this.

```
>>> A = np.array([[1,2], [3, 4], [5,6]])
>>> A.shape
(3, 2)
>>> B = np.array([7,8])
>>> B.shape
(2,)
>>> np.dot(A,  B)
array([23, 53, 83])
```

Matrix Multiplication in a Neural Network

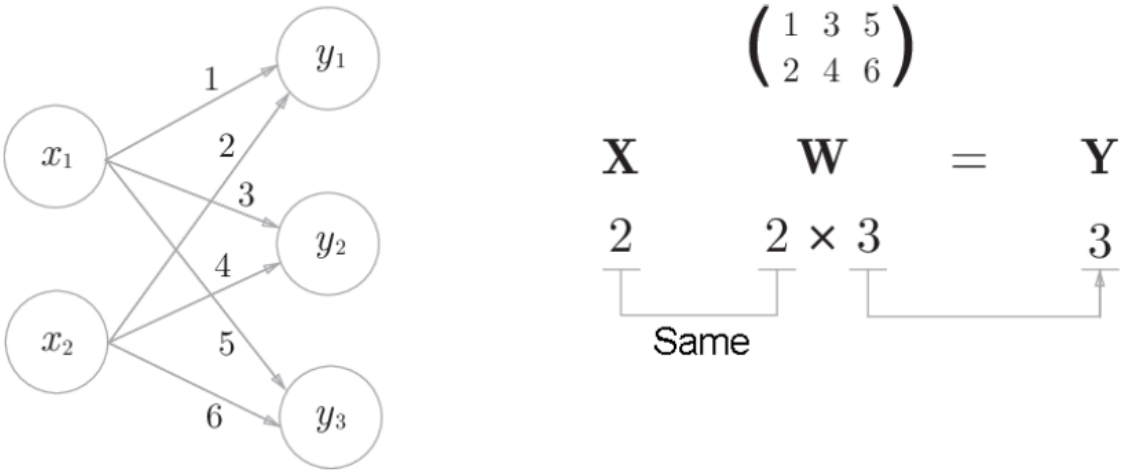


Fig 1.11 Using matrix multiplication to calculate a neural network

careful about the shapes of **X**, **W**, and **Y**. It is very important that the number of elements in the corresponding dimensions of **X** and **W** are the same:

```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
[2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

Implementing a Three-Layer Neural Network

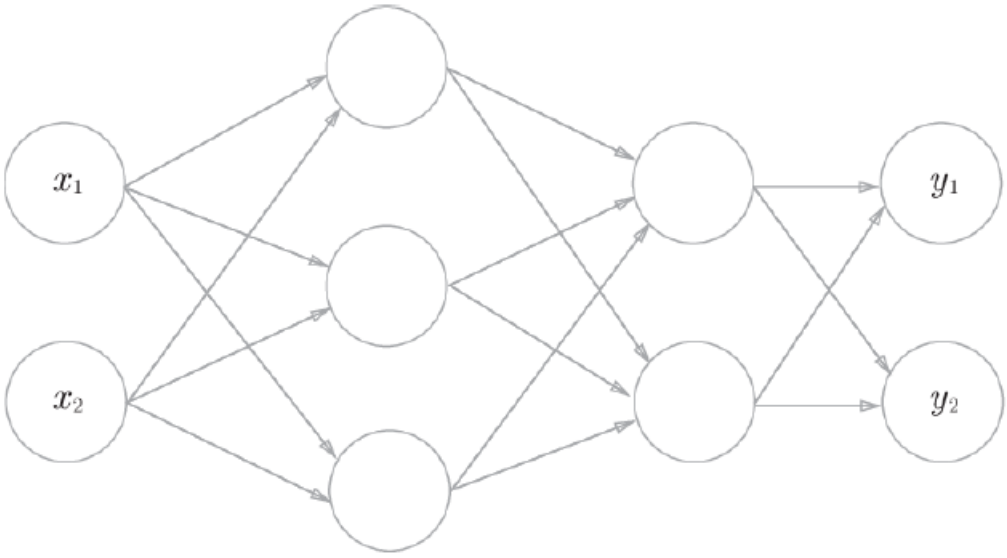


Fig 1.12 A three-layer neural network consisting of two neurons in the input layer (layer 0), three neurons in the first hidden layer (layer 1), two neurons in the second hidden layer (layer 2), and two neurons in the output layer (layer 3)

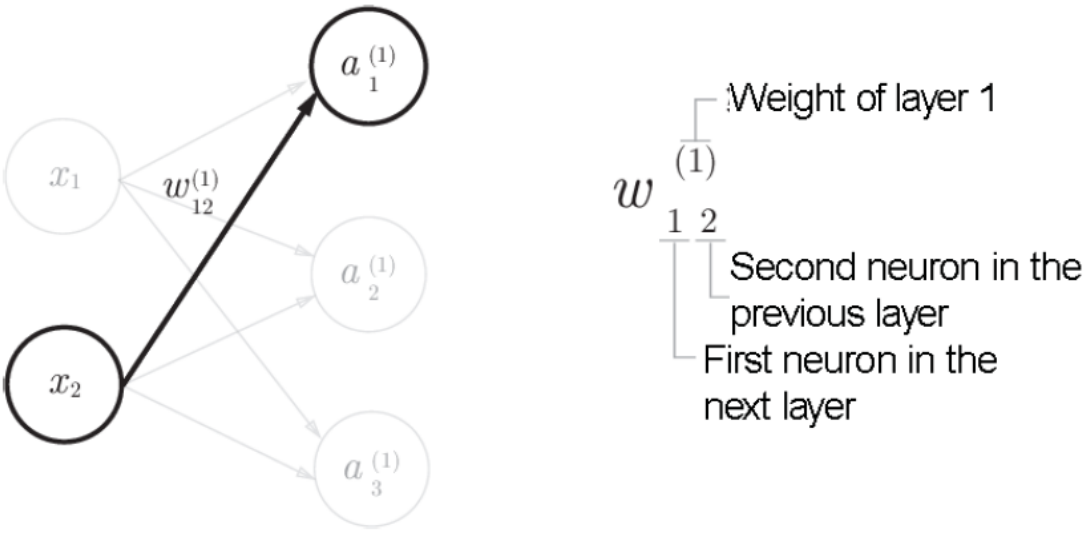


Fig 1.13 **Weight symbols**

Implementing Signal Transmission in Each Layer

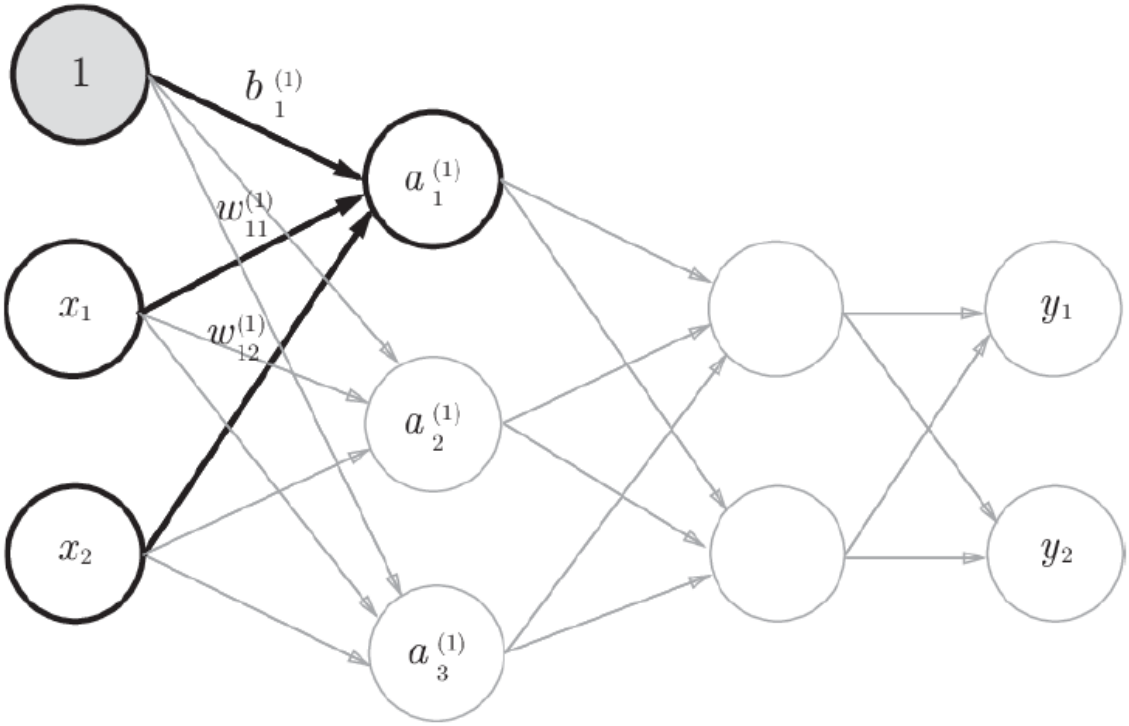


Fig 1.14 Transmitting signals from the input layer to layer 1

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

By using matrix multiplication, you can express "the weighted sum" of layer 1 collectively as follows:

$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

Here, $A^{(1)}$, X, $B^{(1)}$, and $W^{(1)}$ are as follows:

$$\mathbf{A}^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}, \ \mathbf{X} = \begin{pmatrix} x_1 & x_2 \end{pmatrix}, \ \mathbf{B}^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

print(W1.shape) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

A1 = np.dot(X, W1) + B1
```

The weighted sums in a hidden layer (the total of the weighted signals and the biases) are shown as *a*'s, and the signals converted with the activation function are shown as *z*'s. Here, the activation function is shown as *h*() using a sigmoid function:

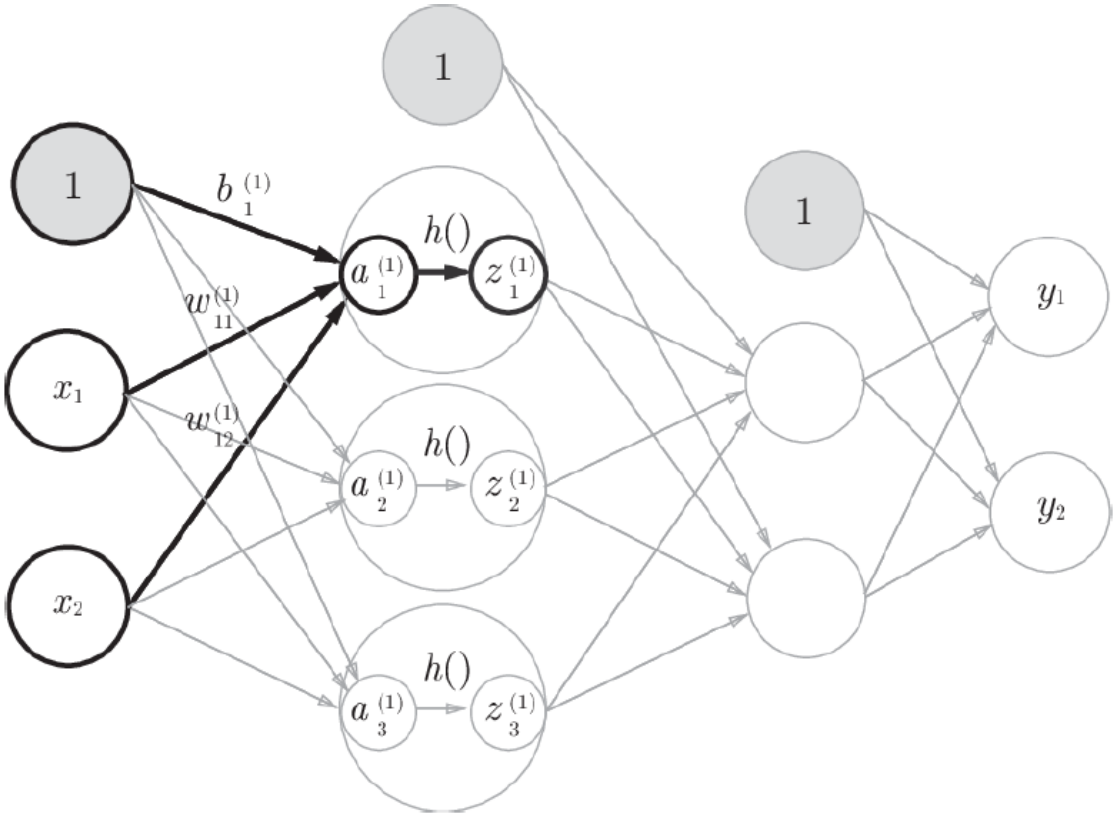


Fig 1.15 Transmitting signals from the input layer to layer 1

This process is implemented in Python as follows:

```
Z1 = sigmoid(A1)
print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [0.57444252, 0.66818777, 0.75026011]
```

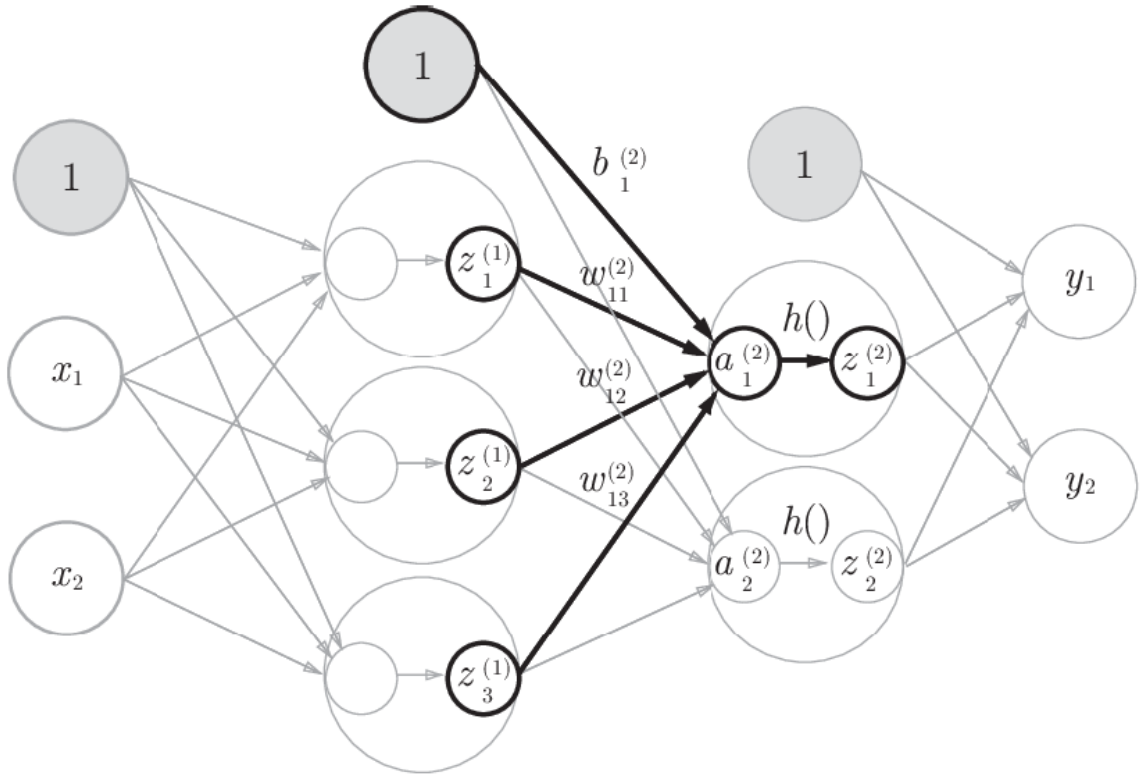


Fig 1.16 Transmitting signals from layer 1 to layer 2

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

print(Z1.shape) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

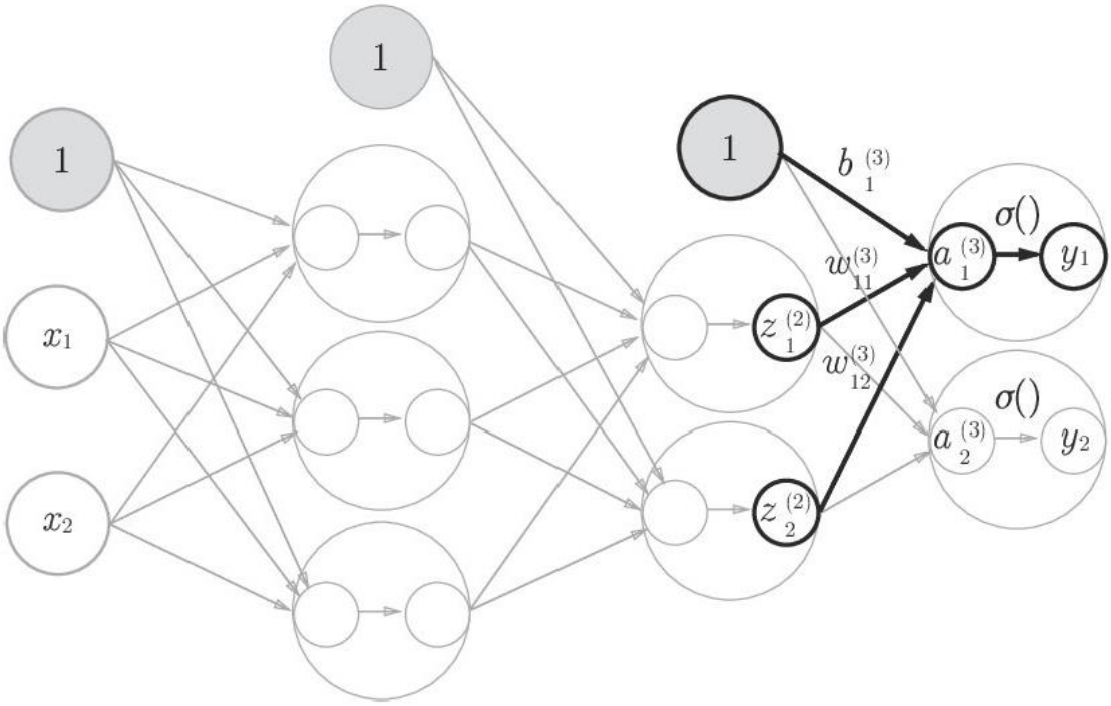


Fig 1.17 Transmitting signals from layer 2 to the output layer

Define a function named **identity_function()** and use it as the activation function for the output layer. An identity function outputs the input as it is. Although you do not need to define **identity_function()** in this example, this implementation is used so that it is consistent with the previous ones.The activation function of the output layer is shown as σ() to indicate that it is different from the activation function, *h*(), of the hidden layers (σ is called **sigma**):

```
def identity_function(x):
    return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3) # or Y = A3
```

Implementation Summary

```
def init_network():
    network = {}
    network['W1'] = np.array([[0.1,  0.3,  0.5],  [0.2,  0.4,  0.6]])
    network['b1'] = np.array([0.1,  0.2,  0.3])
    network['W2'] = np.array([[0.1,  0.4],  [0.2,  0.5],  [0.3,  0.6]])
    network['b2'] = np.array([0.1,  0.2])
    network['W3'] = np.array([[0.1,  0.3],  [0.2,  0.4]])
    network['b3'] = np.array([0.1,  0.2])

    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2)  +  b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2,  W3)  +  b3
    y = identity_function(a3)

    return y


network = init_network()
x =  np.array([1.0,  0.5])
y = forward(network, x)
print(y) # [ 0.31682708 0.69627909]
```

Designing the Output Layer

Neural network can be used for both a classification problem and for a regression problem. However, you must change the activation function of the output layer, depending on which of the problems you use a neural network for. Usually, an identity function is used for a regression problem, and a softmax function is used for a classification problem.

NOTE: A classification problem is a problem of identifying which class the data belongs to—for example, classifying the person in an image as a man or a woman—while a regression problem is a problem of predicting a (continuous) number from certain input data—for example, predicting the weight of the person in an image.

Identity Function and Softmax Function

An identity function outputs the input as it is. The function that outputs what is entered without doing anything is an identity function. Therefore, when an identity function is used for the output layer, an input signal is returned as-is.

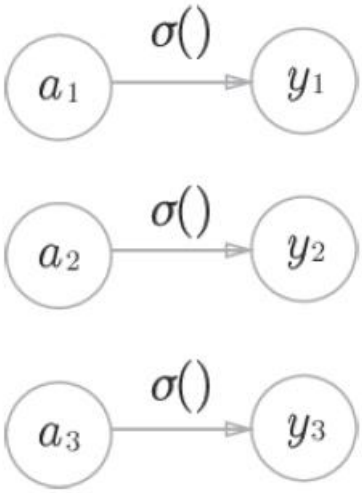


Fig 1.18 **Identity function**

The softmax function, which is used for a classification problem, is expressed by the following equation:

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

exp(x) is an exponential function that indicates ex (e is Napier's constant, 2.7182…). Assuming the total number of output layers is n, the equation provides the k-th output, yk. As shown in equation, the numerator of the softmax function is the exponential function of the input signal, *a*k, and the denominator is the sum of the exponential functions of all the input signals.

The output of the softmax function is connected from all the input signals with arrows. As the denominator of equation indicates, each neuron of the output is affected by all the input signals:

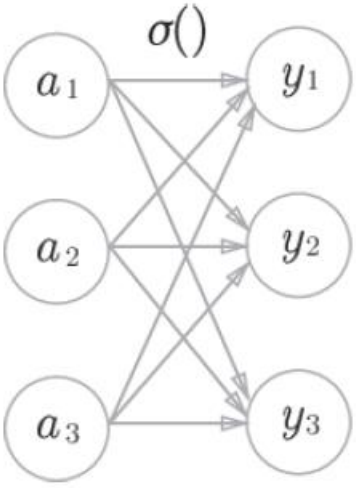


Fig 1.19 Softmax function

```
def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

Issues when Implementing the Softmax Function

The implementation of the softmax function represents equation correctly, but it is defective for computer calculations. This defect is an overflow problem. Implementing the softmax function involves calculating the exponential functions, and the value of an exponential function can be very large. For example, $e10$ is larger than 20,000, and $e100$ is a large value that has more than 40 digits. The result of $e1000$ returns **inf**, which indicates an infinite value. Dividing these large values returns an "unstable" result.
Improved implementation of the softmax function is obtained from the following equation:

$$
\begin{aligned}
y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} &= \frac{C \exp(a_k)}{C \sum_{i=1}^{n} \exp(a_i)} \\
&= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n} \exp(a_i + \log C)} \\
&= \frac{\exp(a_k + C')}{\sum_{i=1}^{n} \exp(a_i + C')}
\end{aligned}
$$

Adding or subtracting a certain constant does not change the result when the exponential functions in the softmax function are calculated. Although you can use any number as C' here, the largest value from the input signals is usually used to prevent an overflow.

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # Calculating the softmax function
array([  nan,   nan,   nan]) # Not calculated correctly
>>>
>>> c = np.max(a) # 1010
>>> a - c
array([ 0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([   9.99954600e-01,    4.53978686e-05,   2.06106005e-09])
```

when the largest value of the input signals (c, here) is subtracted, you can calculate the function properly. Otherwise, nan (not a number: unstable) values are returned. Based on this description, we can implement the softmax function as follows:

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a - c) # Prevent an overflow
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

Characteristics of the Softmax Function

use the **softmax()** function to calculate the output of the neural network, as follows:

```
>>> a = np.array([0.3, 2.9, 4.0])
>>> y = softmax(a)
>>> print(y)
[ 0.01821127 0.24519181 0.73659691]
>>> np.sum(y)
1.0
```

The softmax function outputs a real number between 0 and 1.0. The total of the outputs of the softmax function is 1. The fact that the total is 1 is an important characteristic of the softmax function as it means we can interpret the output of the softmax function as "probability."

The probability of **y[0]** as **0.018** (1.8%), the probability of **y[1]** as **0.245** (24.5%), and the probability of **y[2]** as **0.737** (73.7%). From these probabilities, we can say, "because the second element is the most probable, the answer is the second class."

Number of Neurons in the Output Layer

determine the number of neurons in the output layer as appropriate, depending on the problem to solve. For classification problems, the number of classes to classify is usually used as the number of neurons in the output layer. For example, to predict a number from **0** to **9** from an input image (10-class classification), 10 neurons are placed in the output layer

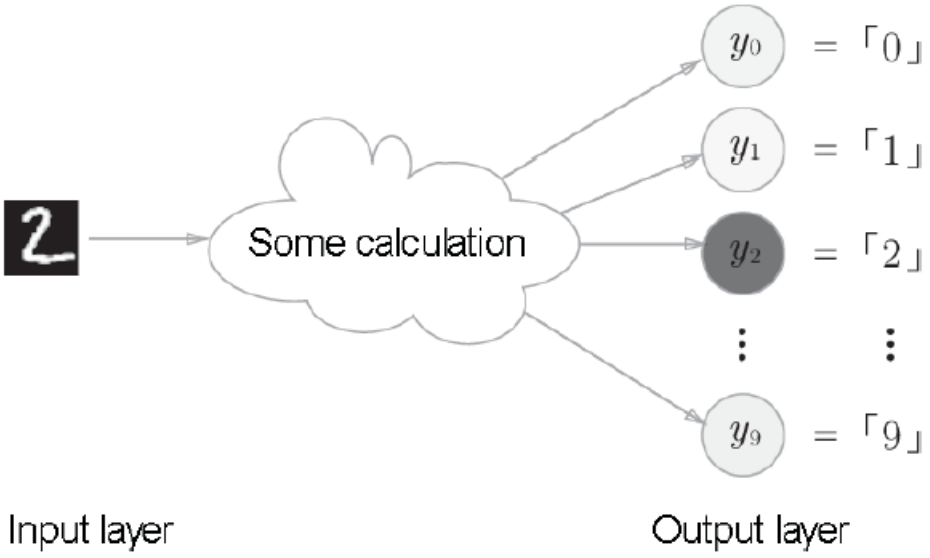


Fig 1.20 The neuron in the output layer corresponds to each number

Handwritten Digit Recognition
classify some handwritten digit images. Assuming that training has already been completed, we will use trained parameters to implement "inference" in the neural network. This inference is also called forward propagation in a neural network.
MNIST Dataset
set of images of handwritten digits.
The MNIST dataset consists of images of numbers from 0 to 9 (*Figure 3. 24*). It contains 60,000 training images and 10,000 test images, and they are used for training and inference.

MNIST's image data is a 28x28 gray image (one channel), and each pixel has a value from 0 to 255. Each image data is labeled, such as "7", "2", and "1."

Neural Network Training
The purpose of training is to discover the weight parameters that lead to the smallest value of the loss function(to learn).

Learning from Data
The essential characteristic of a neural network is its ability to learn from data. Training from data means that weight parameter values can be automatically determined. If you have to determine all the parameters manually, it is quite hard work.

Data-Driven
Data is critical in machine learning. Machine learning looks for an answer in the data, finds a pattern in the data, and tells a story based on it. It can do nothing without data. Therefore, "data" exists at the center of machine learning.

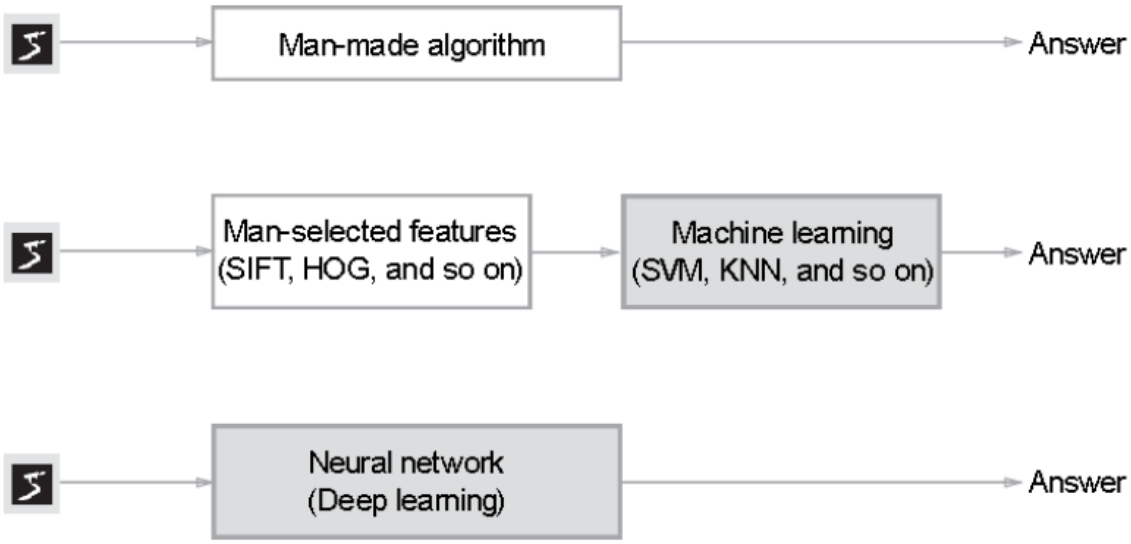


Fig 1.21 A paradigm shift from man-made rules to a "machine" learning from data – a block without human intervention is shown in gray

Training Data and Test Data
we use only training data to find optimal parameters. Then, we use test data to evaluate the ability of the trained model. Why should we divide training data and test data? Because we

want the generalization capability of the model. We must separate the training data and test data because we want to evaluate this **generalization** correctly.
Generalization means the ability of unknown data (data that is not contained in the training data), and the ultimate goal of machine learning is to obtain this generalization.

When a model has become too adapted to only one dataset, **overfitting** occurs. Avoiding overfitting is an important challenge in machine learning.

Loss Function
In neural network training, one "score" is used to indicate the current status. Based on the score, optimal weight parameters are searched for. As this person looks for an "optimal life" based on the "happiness score," a neural network searches for optimal parameters using "one score" as a guide. The score that's used in neural network training is called a **loss function**. Although any function can be used as the loss function, the sum of squared errors or a cross-entropy error is usually used.

Sum of Squared Errors
There are a few functions that are used as loss functions. Probably the most famous one is the **sum of squared errors**. It is expressed by the following equation:

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

Here, *y*k is the output of the neural network, *t*k is labeled data, and *k* is the number of dimensions of the data.

Cross-Entropy Error

$$E = -\sum_{k} t_k \log y_k$$

Here, log indicates the natural logarithm, that is, the logarithm to the base of *e (log*e*)*. *yk* is the output of the neural network and *tk* is the correct label. In *tk*, only the index for the correct label is 1; the other indices are 0

100 pieces of data are selected at random from 60,000 items of training data to be used for training. This training method is called **mini-batch training**.