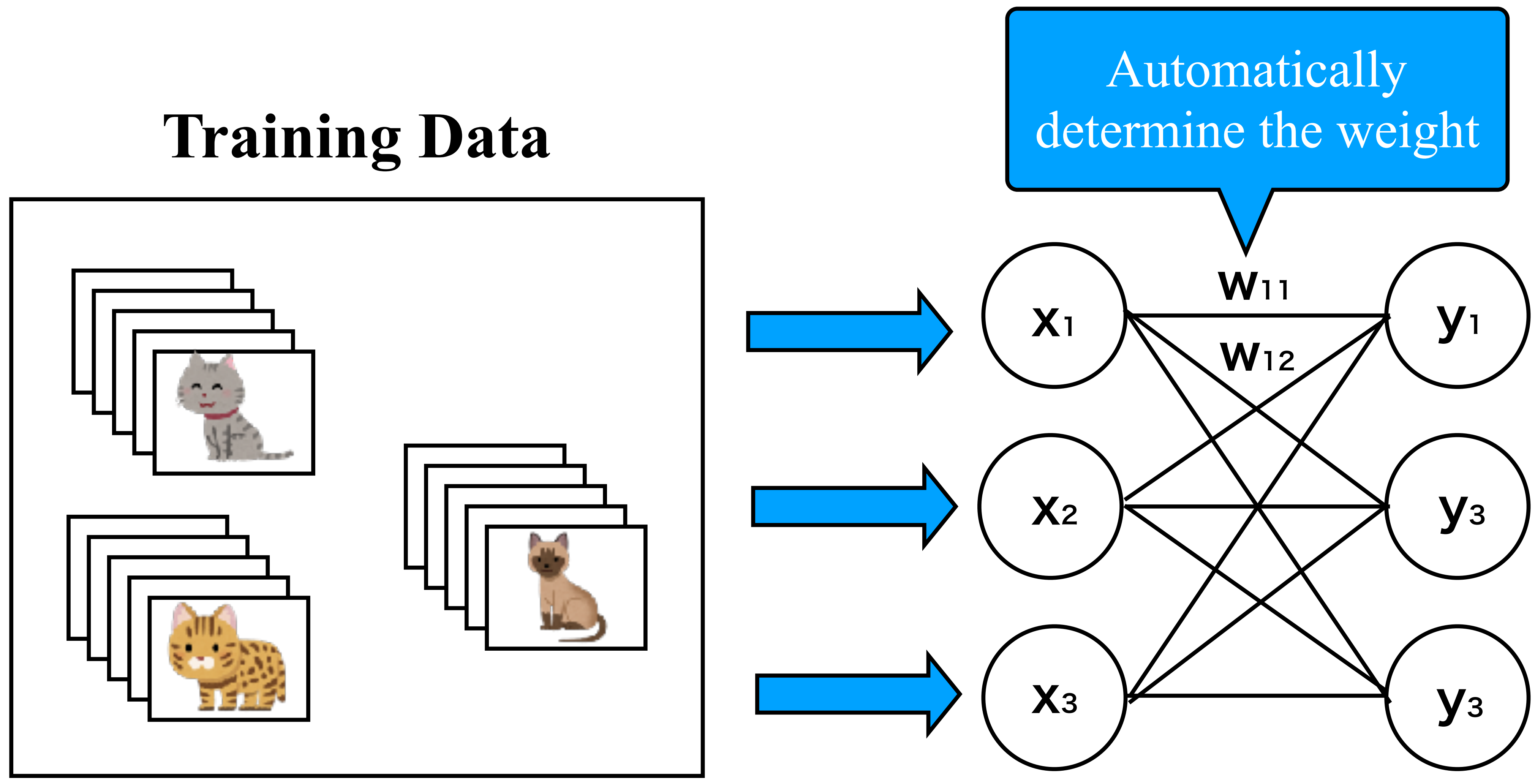
Reading Circle #4

Chapter4 Learning in Neural Network
4.1 Learning from data
4.2 Loss function

06/11/2018
M1 Shunki Kitsunai

What is Learning in neural network

- Refers to automatically acquiring the optimal weight parameter from the training data.



Using the Loss Function to determine the optimal weight in Chapter4

What is the purpose of the Learning

・Find the weight parameter that value of the loss function become smallest based on the loss function.

To find the smallest possible value

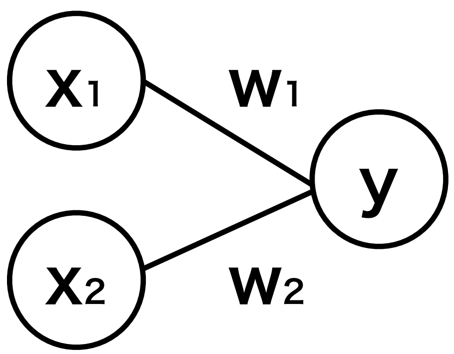
A method using a slope of a function
Gradient method

Outline

- **Learning from data**
- **Training data & Test data**
- **Loss Function**
- **Mini-batch learning**

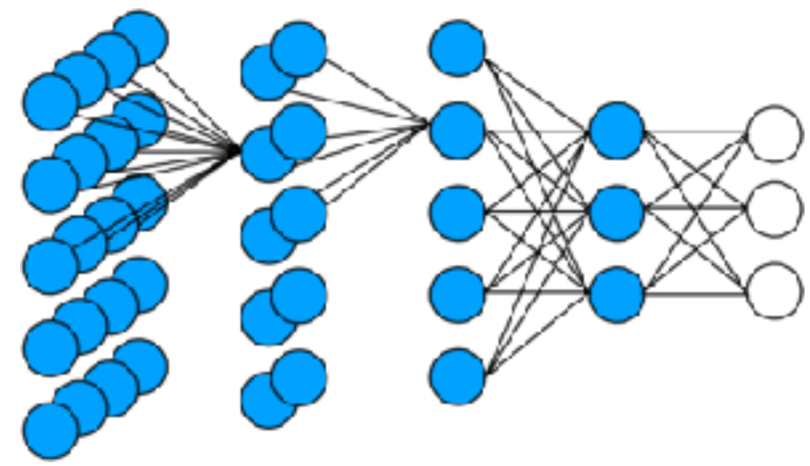
Learning from data

Perceptron in chapter2



Two, three of parameters

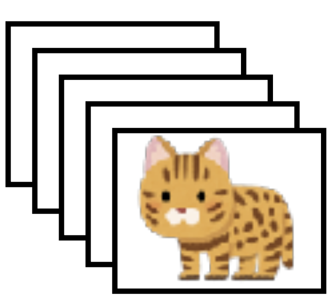
Actually in neural network



Thousands, millions of parameters

Artificially

It is impossible to decide these parameters manually
In neural network.



Automatically

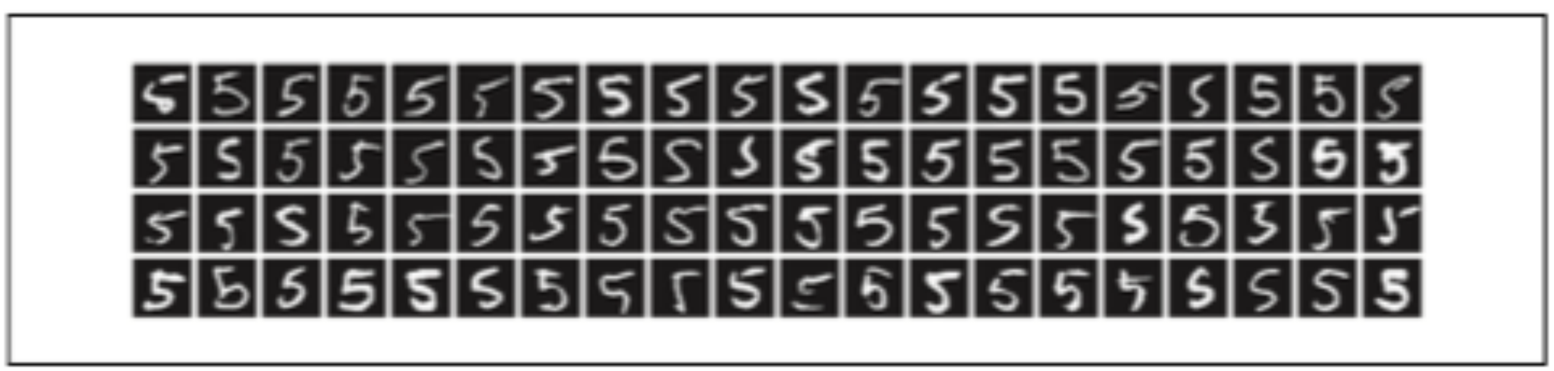
It is possible to decide these parameters automatically
by learning from the data.

Data Driven

Data is central to machine learning.

Namely

Removing from a person-centered approach.



How to recognize 5

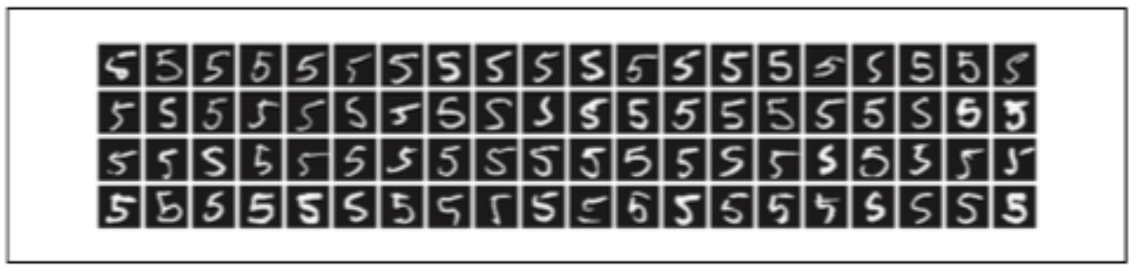
Example of Handwritten Numeral 5

<u>As one method</u>

A feature quantity is extracted from the image,
learn patterns of feature quantities using machine learning techniques

Learning of feature quantity patterns

Machine finds regularity from collected data.



Reduce burden on people

However, even if images are learned as they are, the discrimination accuracy is bad.

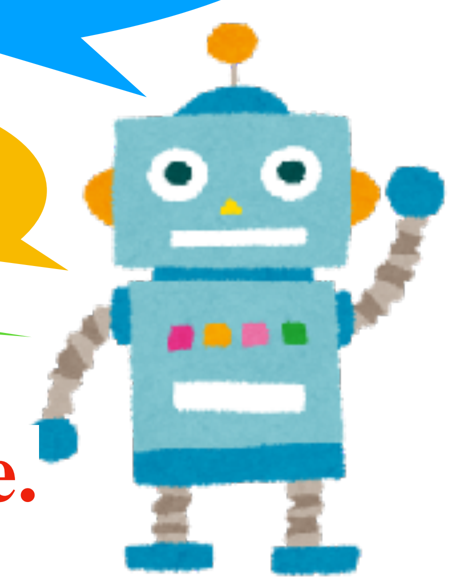
Example of apple

Example of grape

What is an important feature to classify them?

Color?

Shape?



It is necessary to extract the feature quantity from the image.

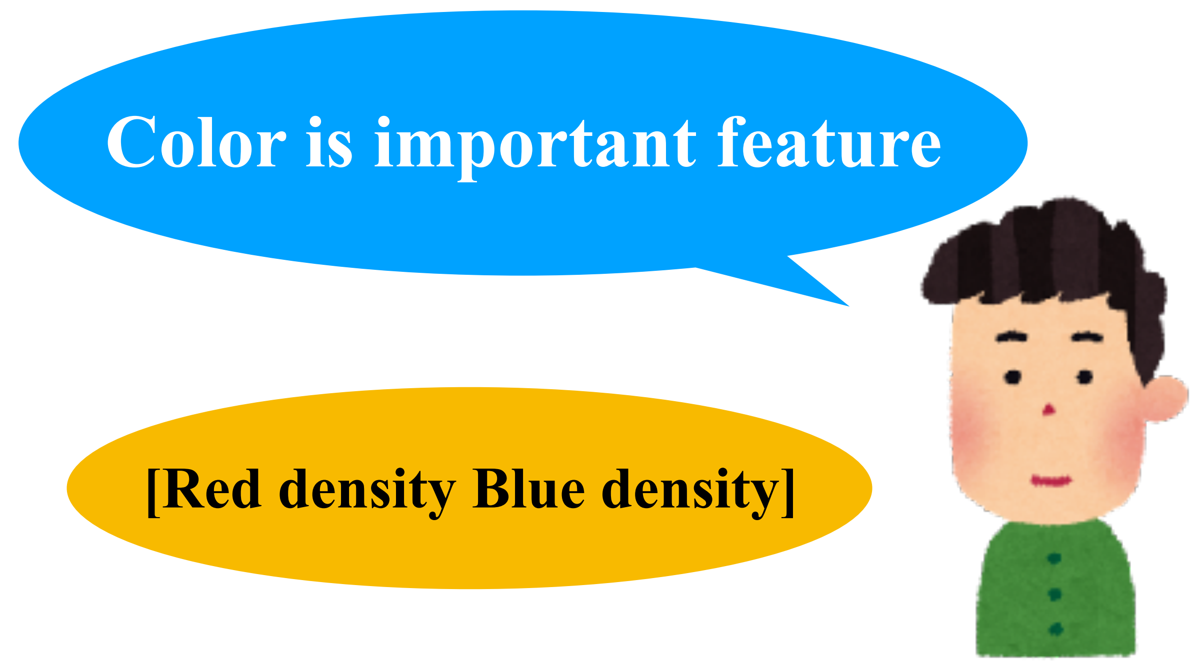
Feature quantity extraction

Vectorization of images

This means convert to one-dimensional data based on feature quantity

Example of apple

Example of grape



It is necessary for a person to decide the feature quantity.

In addition
It is necessary to change the feature quantity for each problem.

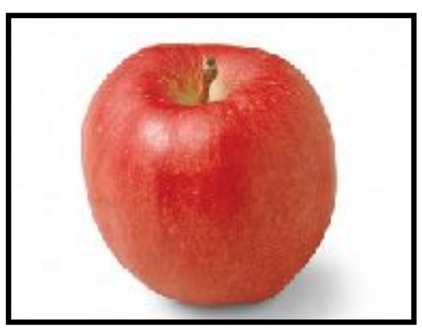
Neural network

The image is learned as it is in the neural network

Feature quantity is learned by machine

Advantage

The point that all problems can be solved in the same flow



Example of apple

Example of grape

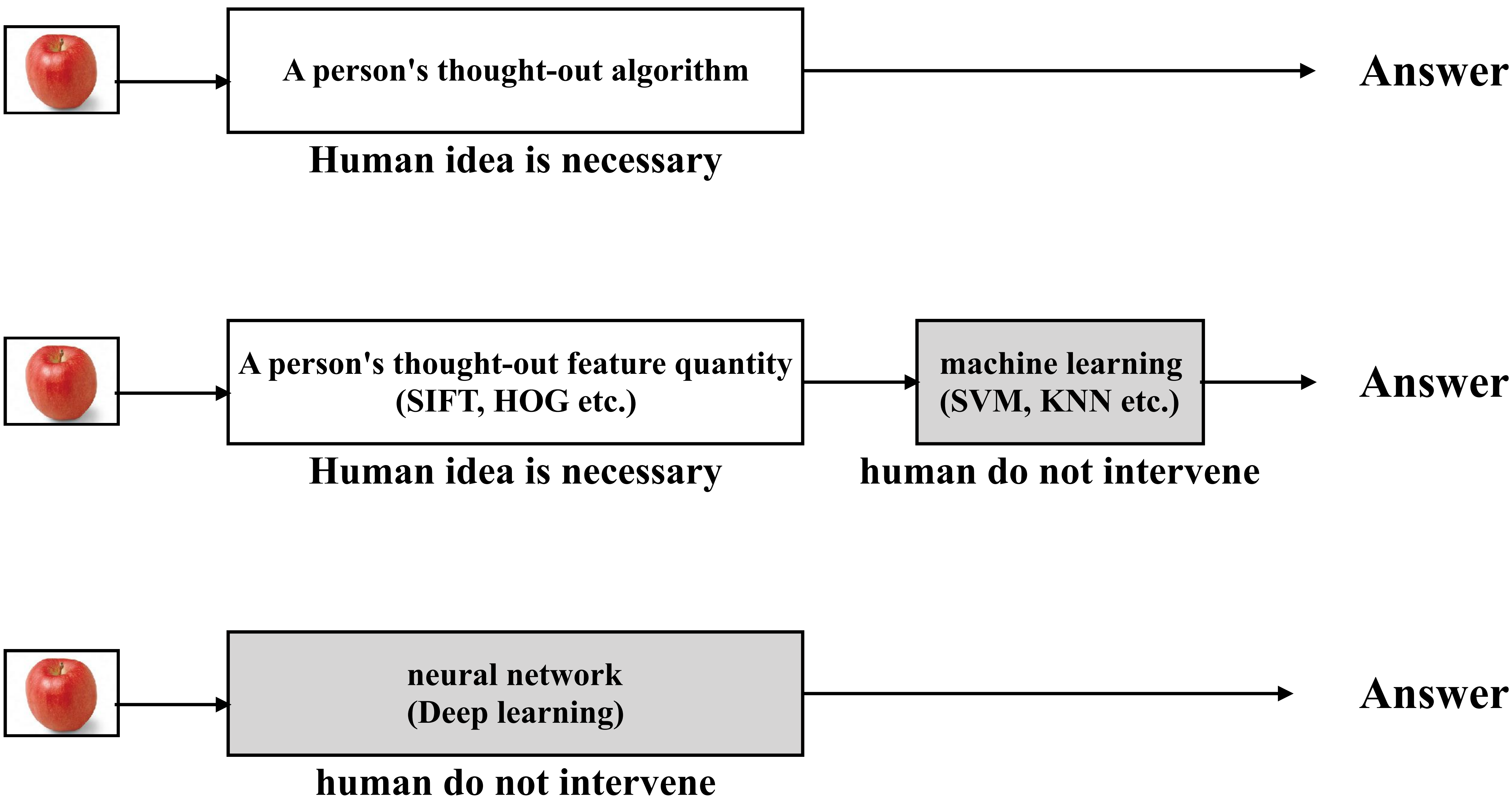
Example of two

Example of five

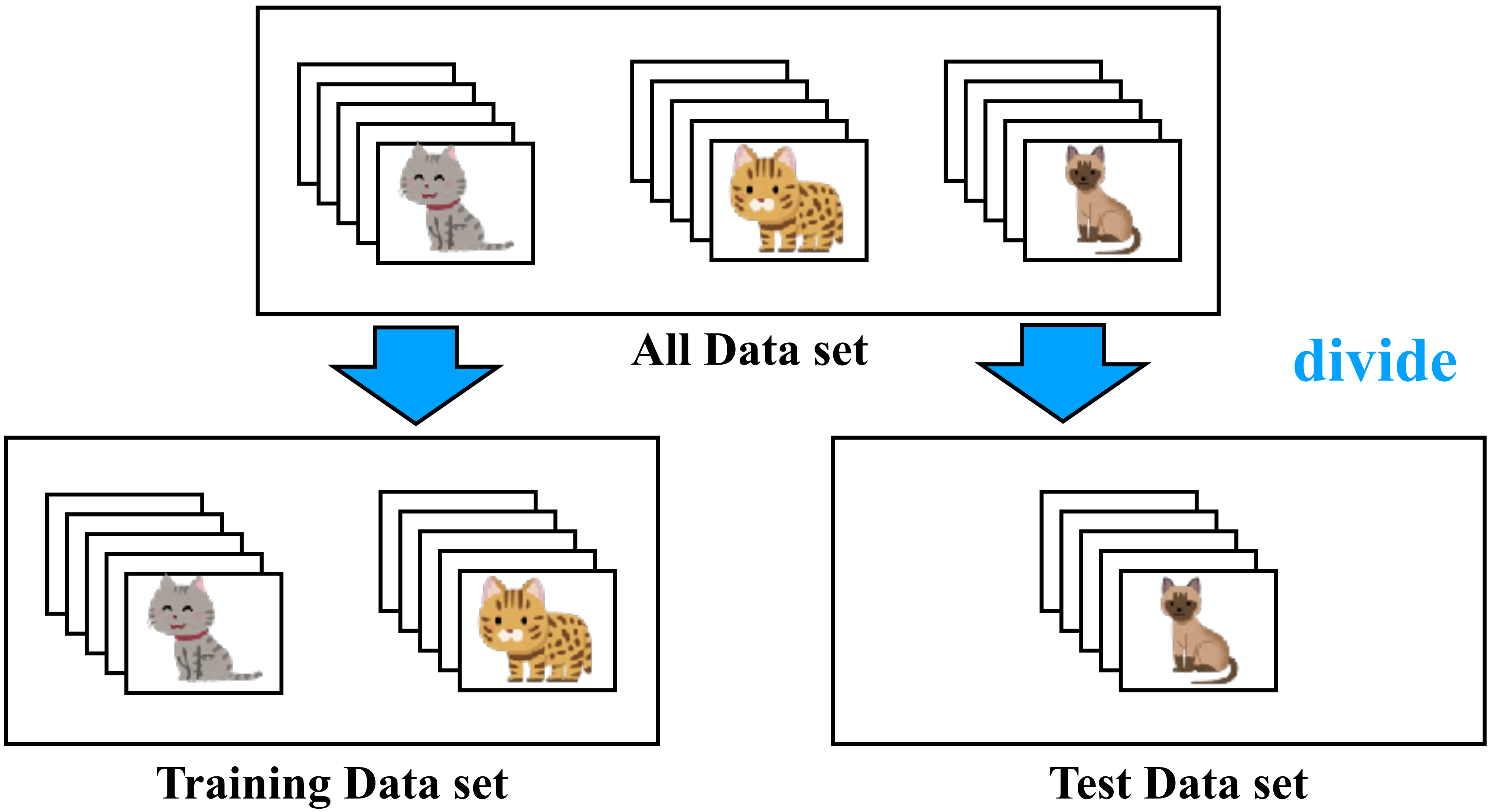
Same flow

The neural network finds a problem pattern from the learning of data

Summary of Learning from data



Training data & Test data

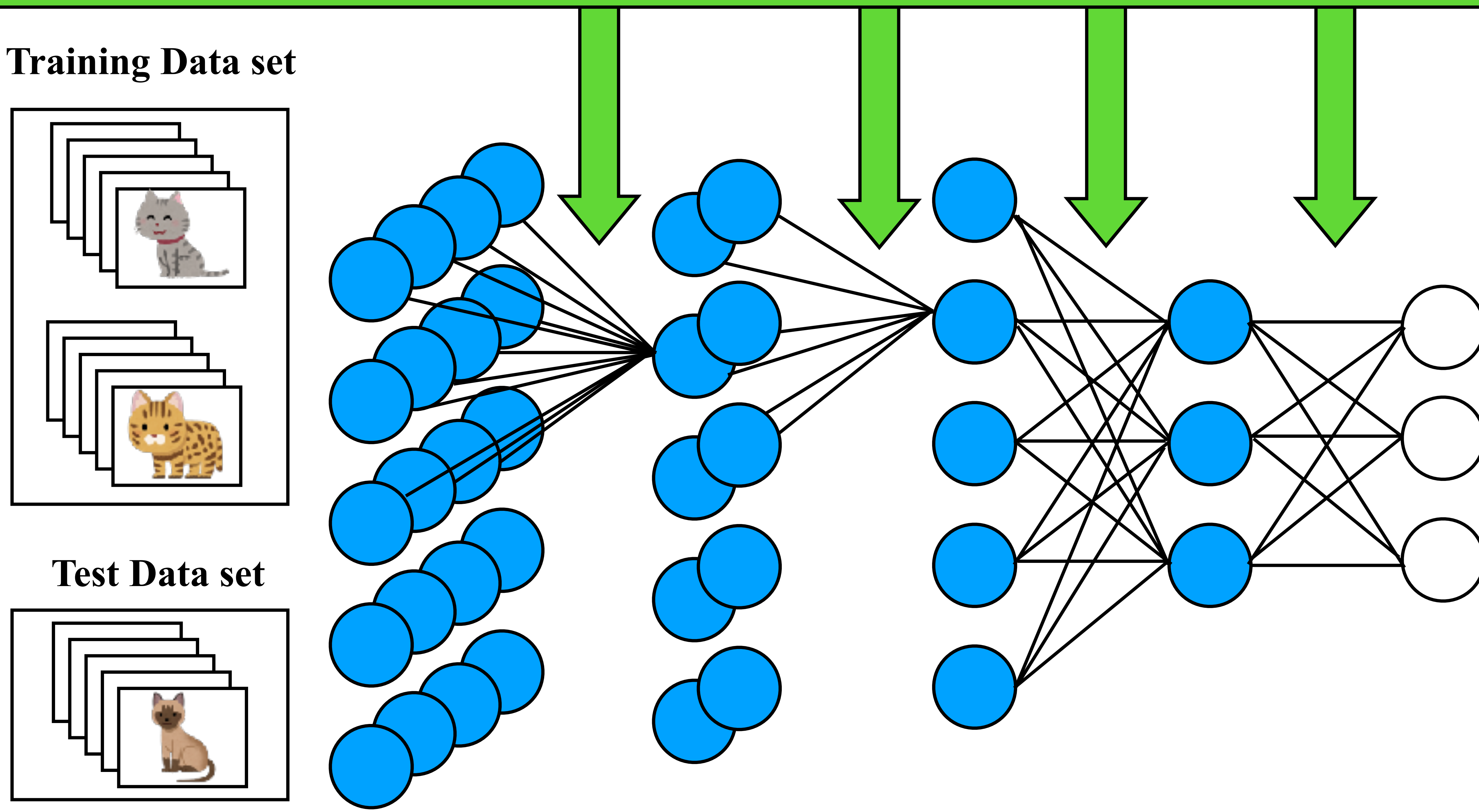


Learn with training data and evaluate generalization ability with test data.

The condition which excessively corresponds to only one data set is called over feeting.

Updating weights with training data

The training data is given to the neural network, and the update of weights is repeated so that the error between the correct label and the output result disappears.



Loss function

Indicators for searching for optimum weight parameters

Namely

It shows bad performance of the neural network.

Loss function × **Minus** = **Not bad** = **Good**

Loss function

- **mean squared error**
- **cross entropy error**

mean squared error

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

y_k is output of neural network.
t_k is training data
k is number of dimensions

In the example of handwritten digit recognition

y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]   Probability of corresponding numbers

t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]   Label: correct is 1, incorrect is 0

「0」 「1」 「2」 … 「9」   Number 0-9

one-hot expression

Implementation of mean squared error

Program

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

# mean squared error
def mean_squared_error(y, t):
    return 0.5 * numpy.sum((y-t)**2)

# 「2」is correct
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# eg.1: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print (str(mean_squared_error(numpy.array(y), numpy.array(t))))

# eg. 2: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print (str(mean_squared_error(numpy.array(y), numpy.array(t))))
```

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

Output

```
0.09750000000000003
0.5975
```

eg. 1: When the probability of 「2」 is the highest (0.6)
eg. 2: When the probability of 「7」 is the highest (0.6)

eg.1 has a smaller error than eg.2 between the output result and the training data. The first example is better suited to training data.

Cross entropy error

$$E = -\sum_k \underline{t_k} \log y_k \qquad (2)$$

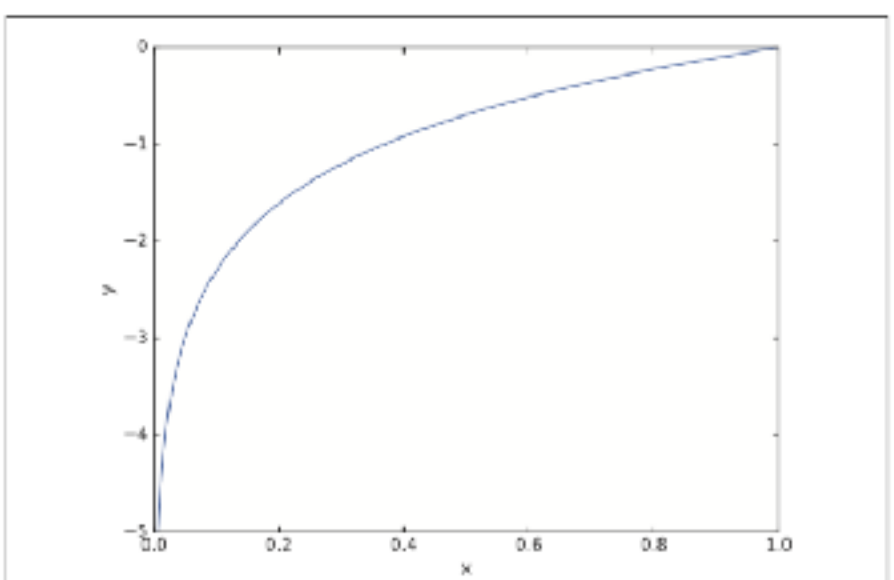
Calculate only when the value of t_k is 1

$E = -t_k \log y_k$

y = [0.1, 0.05, **0.6**, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] Probability of corresponding numbers

t = [0, 0, **1**, 0, 0, 0, 0, 0, 0, 0] Label: correct is 1, incorrect is 0

「0」 「1」 「2」 … 「9」 Number 0-9



A graph of natural logarithm y = log x

In Expression (2)
Output corresponding to the correct label is y_k

0 Value of y_k 1

High Cross entropy error Low

Implementation of cross entropy error

Program

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

# cross entropy error
def cross_entropy_error(y, t):
    delta = 1e-7
    return -numpy.sum(t * numpy.log(y + delta))

# 「2」is correct
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# eg.1: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))

# eg. 2: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))
```

$E = -t_k \log y_k$

If y is 0,
numpy.log(0) = -inf.

Can not calculate any more

So add a small value delta.

Output

```
0.510825457099338
2.302584092994546
```

eg. 1: When the probability of 「2」 is the highest (0.6)
eg. 2: When the probability of 「7」 is the highest (0.6)

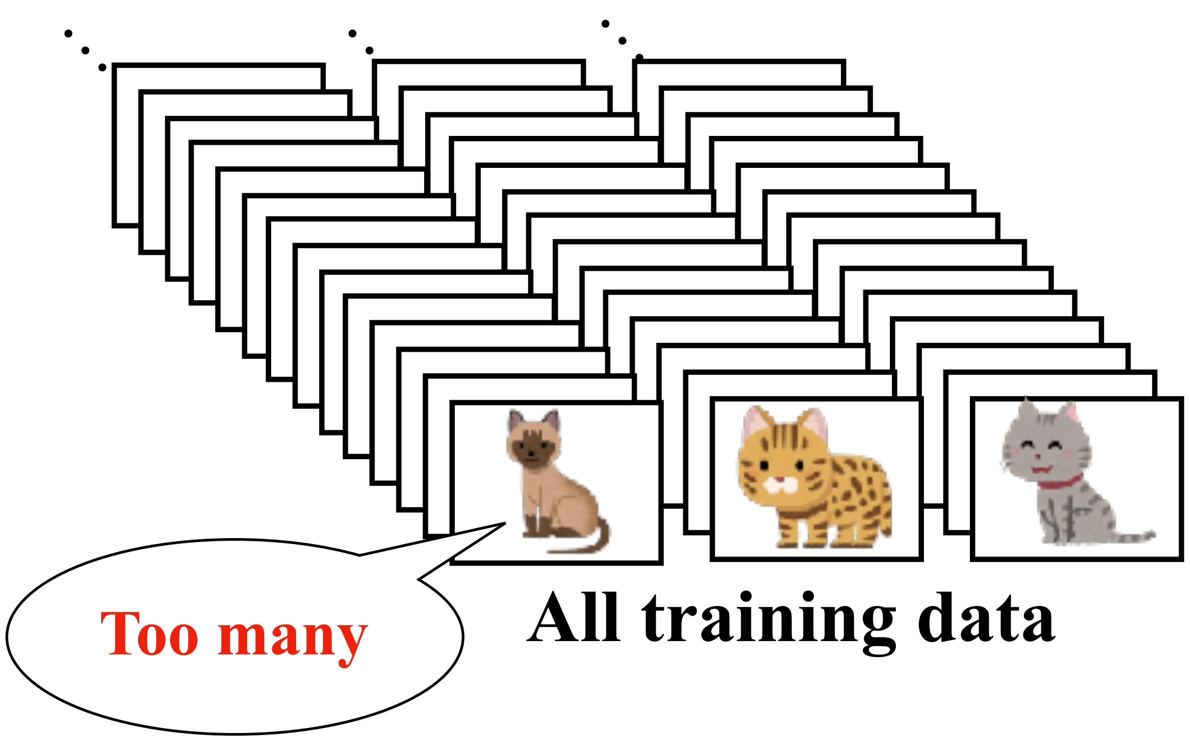
eg.1 has a smaller error than eg.2 between the output result and the training data.
The first example is better suited to training data.

Mini-batch learning

The loss function needs to be calculated for all data.

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk} \log y_{nk} \qquad \textbf{(3)}$$

E is average loss function per piece.
y_{nk} is output of neural network.
t_{nk} is training data.
k is number of dimensions.
N is number of training data.

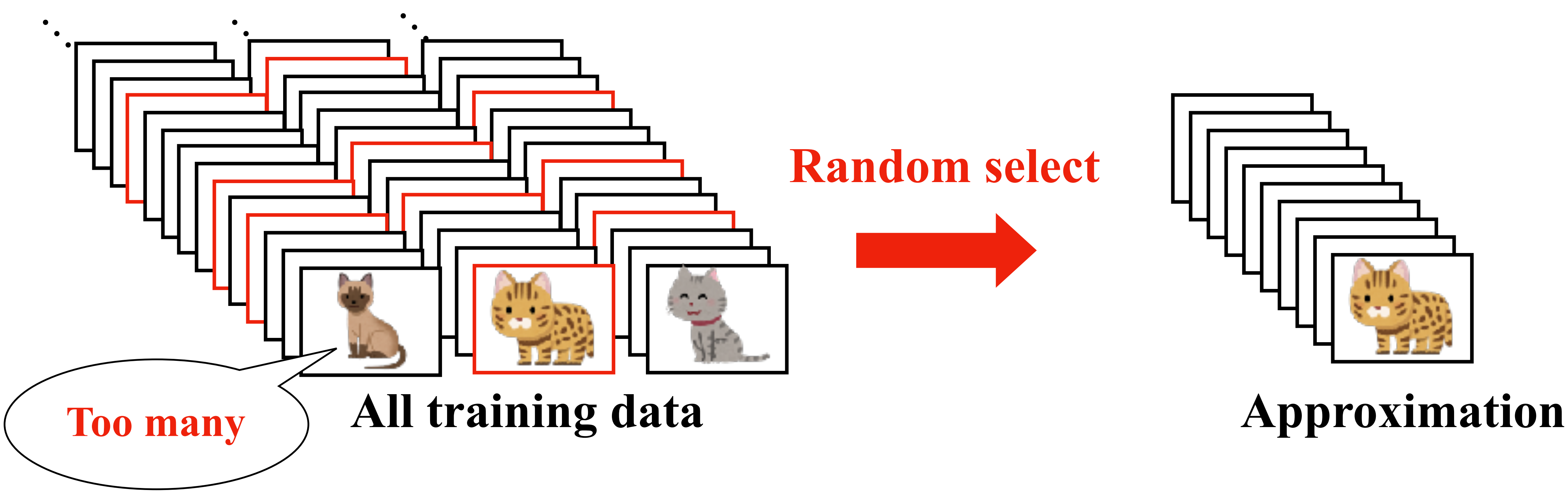


Mini-batch learning

The loss function needs to be calculated for all data.

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \qquad (3)$$

E is average loss function per piece.
y_{nk} is output of neural network.
t_{nk} is training data.
k is number of dimensions.
N is number of training data.

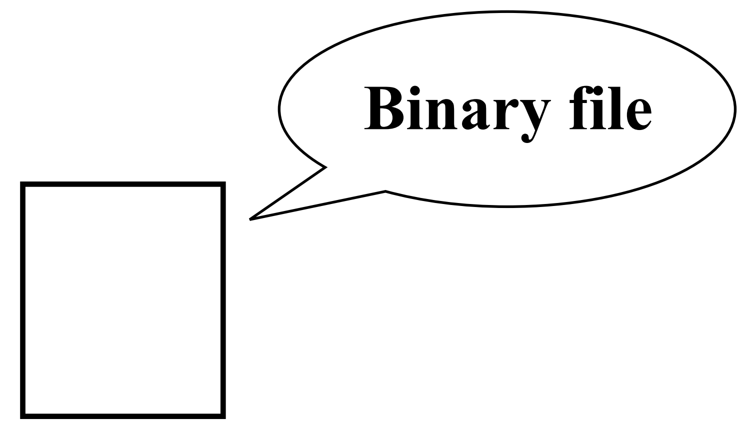


Implementation of mini-batch learning

Load mnist data set (in chapter3)

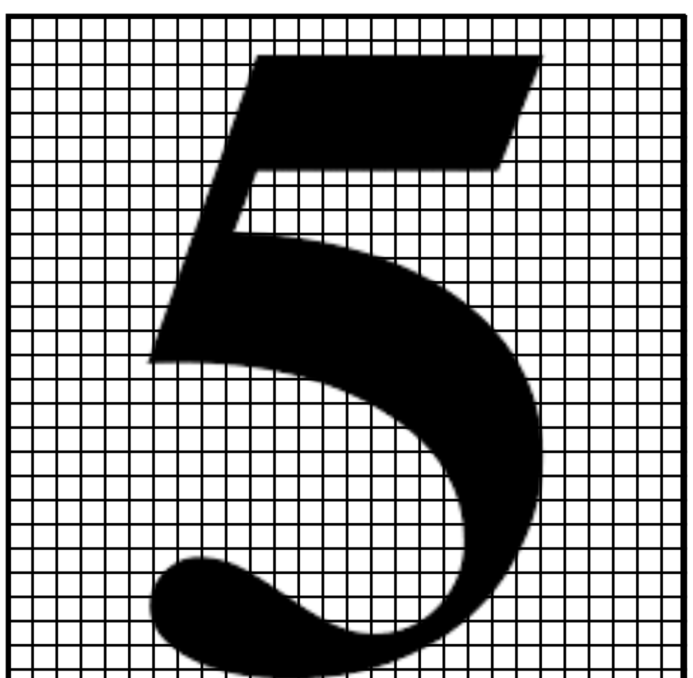
Program

```
import numpy as numpy
import sys, os
sys.path.append(os.pardir)
from dataset.mnist import load_mnist
(x_train, t_train), (x_test, t_test) = \
   load_mnist(normalize=True, one_hot_label=True)
print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000, 10)
```



mnist.pki

28 pixel



28 pixel

Training data: 60,000 pieces, 10 columns
Input data: 28 pixel × 28 pixel = 784 columns

Implementation of mini-batch learning

numpy.random.choice()

Program

```
import numpy as numpy
import sys, os
sys.path.append(os.pardir)
from dataset.mnist import load_mnist
(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)
train_size = x_train.shape[0]
batch_size = 10
batch_mask = numpy.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
print(x_batch.shape) # (10, 784)
print(t_batch.shape) # (10, 10)
print(batch_mask) # [37480 33545 41369  5954 43017 12797 38063 30547 51212 25036]
```

Extract 10 data randomly

>>> numpy.random.choice(60000, 10)
[37480 33545 41369 5954 43017 12797 38063 30547 51212 25036]

0 < batch_mask < 60,000

Implementation of cross entropy error [batch support]

Program

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

# cross entropy error (Batch support)
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    batch_size = y.shape[0]
    return -numpy.sum(t * numpy.log(y + 1e-7)) / batch_size
                                         delta

# 「2」 is correct
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# eg.1: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))

# eg. 2: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))
```

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk} \log y_{nk}$$

Output

```
0.510825457099338
2.302584092994546
```

eg. 1: When the probability of 「2」 is the highest (0.6)
eg. 2: When the probability of 「7」 is the highest (0.6)

Implementation of cross entropy error [batch support]

Program

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

def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    batch_size = y.shape[0]
    print (str(batch_size))
    return -numpy.sum(t * numpy.log(y + 1e-7)) / batch_size

# 「2」is correct
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

y = [[0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0],
[0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))
```

$$E = -\frac{1}{N}\sum_n \sum_k t_{nk} \log y_{nk}$$

Output

```
2
1.406704775046942
```

batch_size
When y is two dimensional

Implementation of cross entropy error [batch support]

Program

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

def cross_entropy_error(y, t):
        if y.ndim == 1:
                t = t.reshape(1, t.size)
                y = y.reshape(1, y.size)
        batch_size = y.shape[0]
        return -numpy.sum(numpy.log(y[numpy.arange(batch_size), t] + 1e-7)) / batch_size

                    [y[0,0], y[0,4], y[0,1], y[0,2], y[0,2], y[0,3], y[0,4], y[0,1], y[0,0]]

t = [0, 1, 4, 1, 2, 2, 3, 4, 1, 0]      Not one-hot

# eg.1: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))

# eg. 2: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))
```

arange(5)
→[0, 1, 2 , 3 , 4]

Output

```
36.723566118926044
40.30708339071646
```

eg. 1: When the probability of 「2」 is the highest (0.6)
eg. 2: When the probability of 「7」 is the highest (0.6)

Implementation of cross entropy error [batch support]

Program

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

def cross_entropy_error(y, t):
        if y.ndim == 1:
                t = t.reshape(1, t.size)
                y = y.reshape(1, y.size)
        batch_size = y.shape[0]
        return -numpy.sum(numpy.log(y[numpy.arange(batch_size), t] + 1e-7)) / batch_size

# 「2」is correct
t = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

# eg.1: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))

# eg. 2: When the probability of 「2」 is the highest (0.6)
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print (str(cross_entropy_error(numpy.array(y), numpy.array(t))))
```

Output

```
5.108254570993381
23.02584092994546
```

eg. 1: When the probability of 「2」 is the highest (0.6)
eg. 2: When the probability of 「7」 is the highest (0.6)

Why set loss function?

Update parameters gradually

Differentiation is used to find optimal parameters

Calculate derivative of parameter ➡ **Differential value**

Positive: Update in negative direction

0: Update done

Negative: Update in positive direction

<u>In the loss function</u>

Respond to slight changes in parameters

⬇

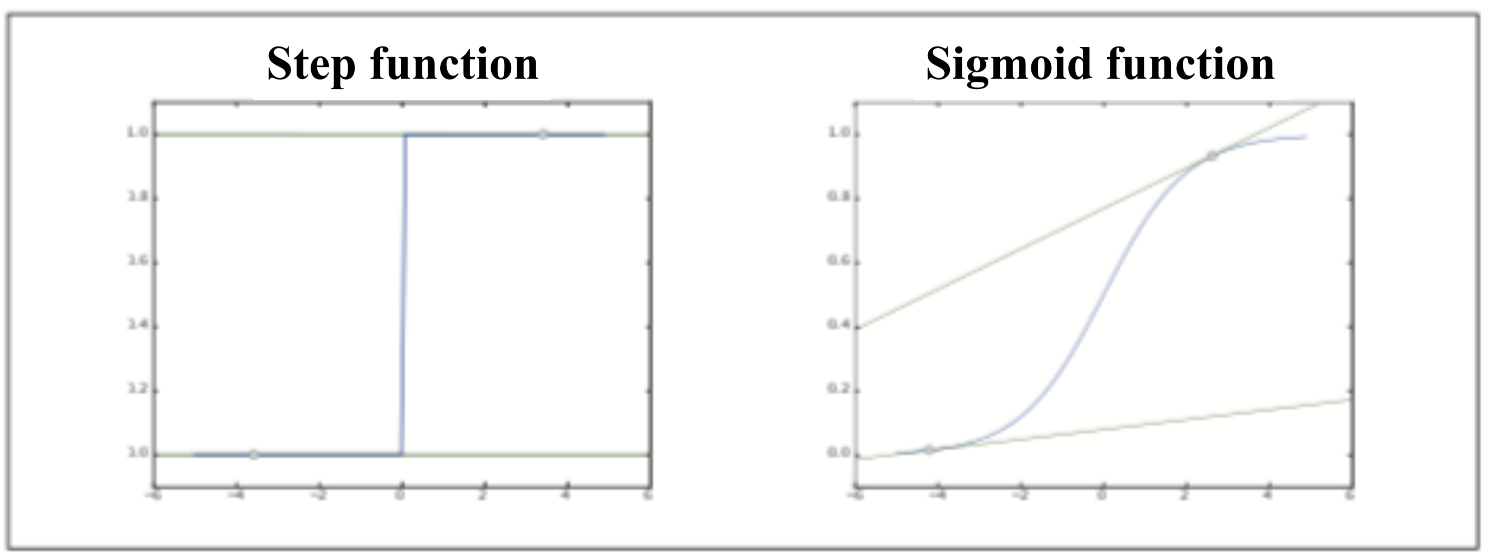
Value of the loss function is continuous.

Why set loss function?

When recognition accuracy is used as an index

It does not respond to slight changes in parameters.

Value of the loss function is discontinuous.



Step function: In most places the slope is 0

Sigmoid function: The gradient does not become 0