

# 深層学習にふれてみよう

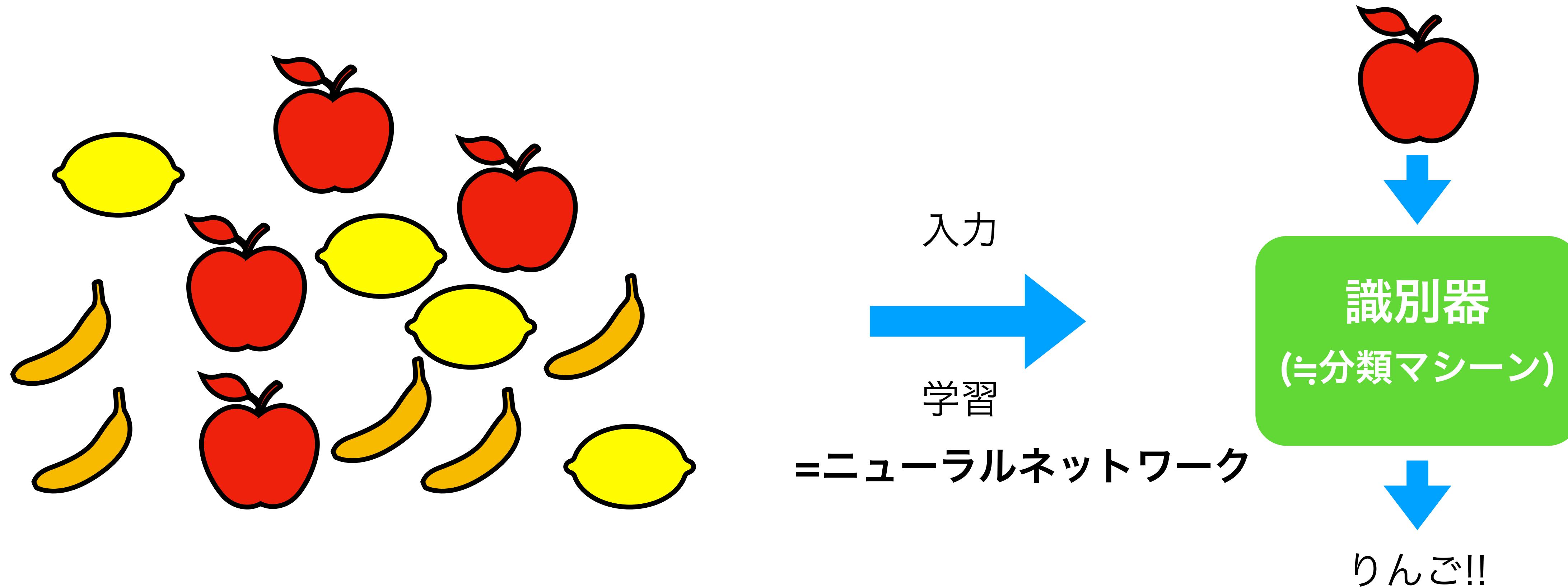
本教材を使用した際にはお手数ですが、  
下記アンケートフォームにご協力下さい。



統合教育機構  
須藤 肇顕

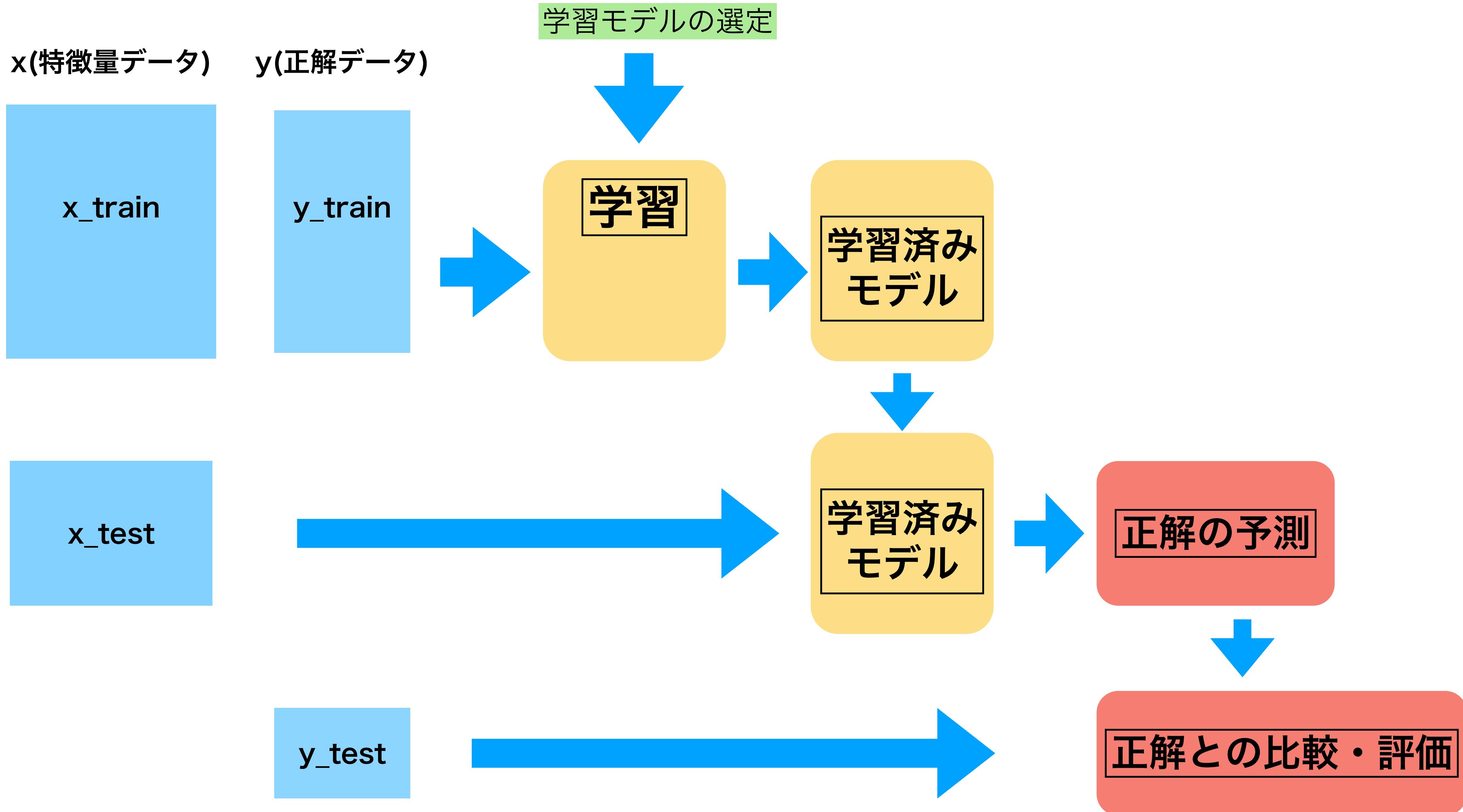
# 深層学習を実践してみよう！！

基本的な全体の流れは前回の機械学習と同様

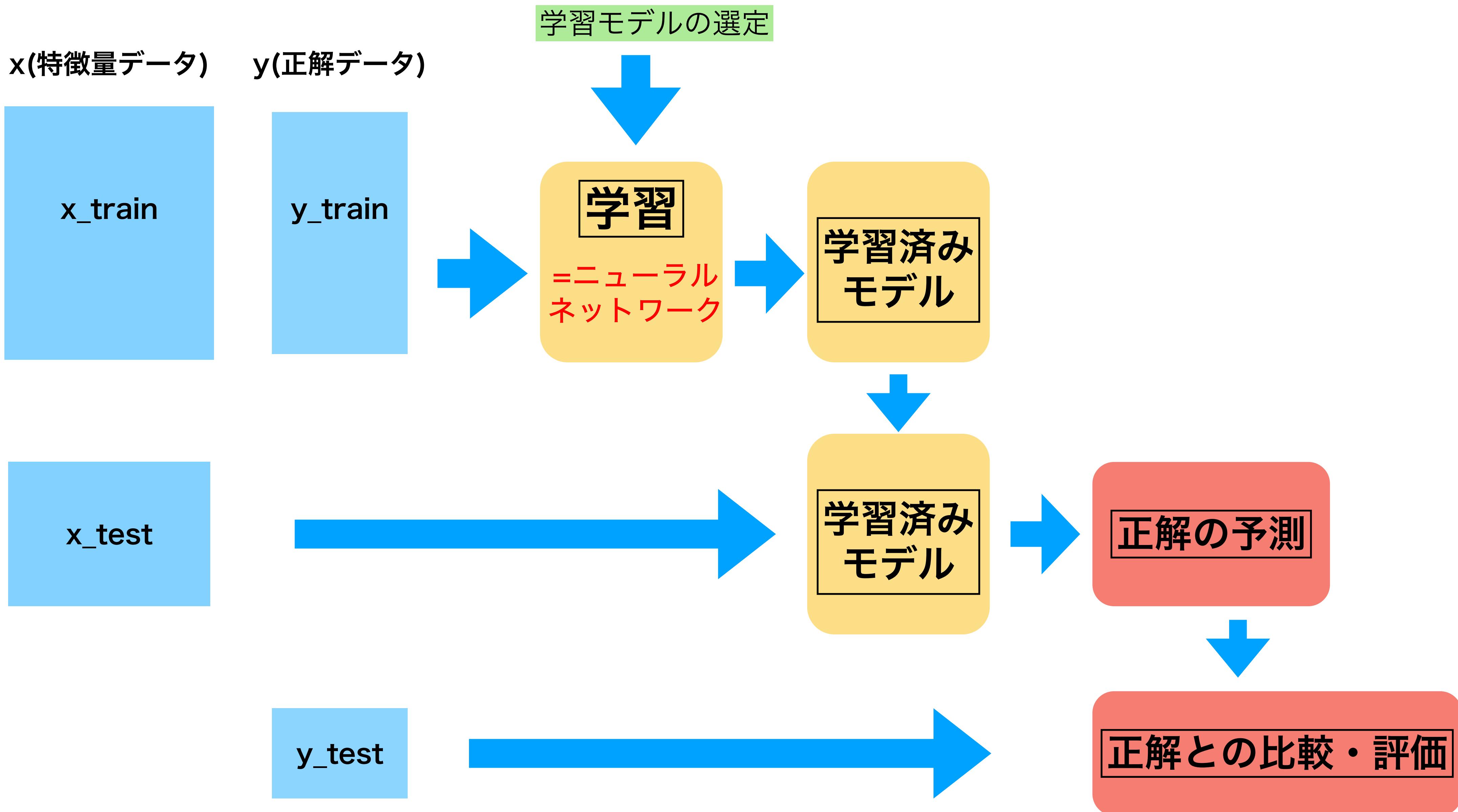


深層学習では、学習モデルにニューラルネットワークを用いる

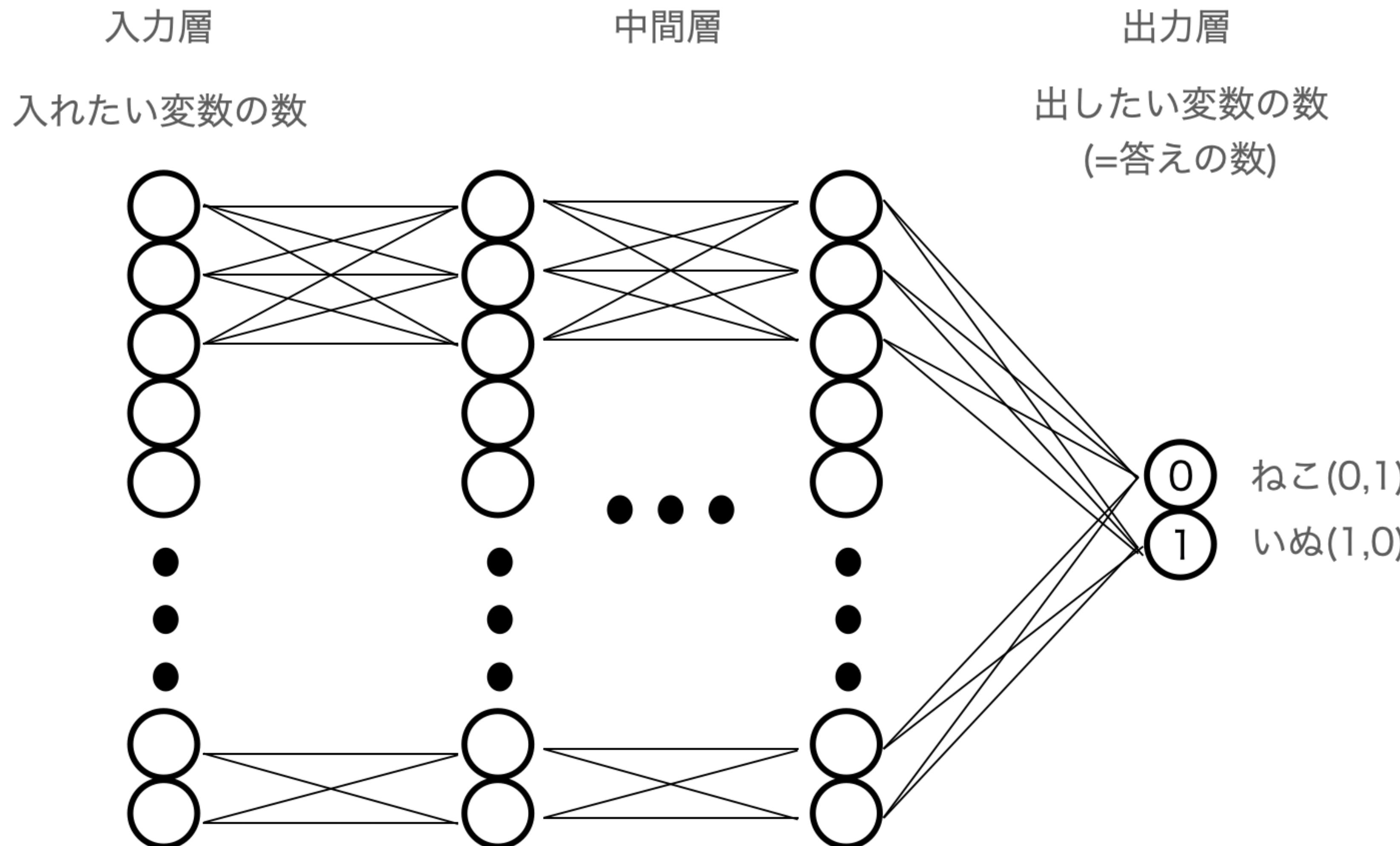
# 深層学習の流れ



# 深層学習の流れ



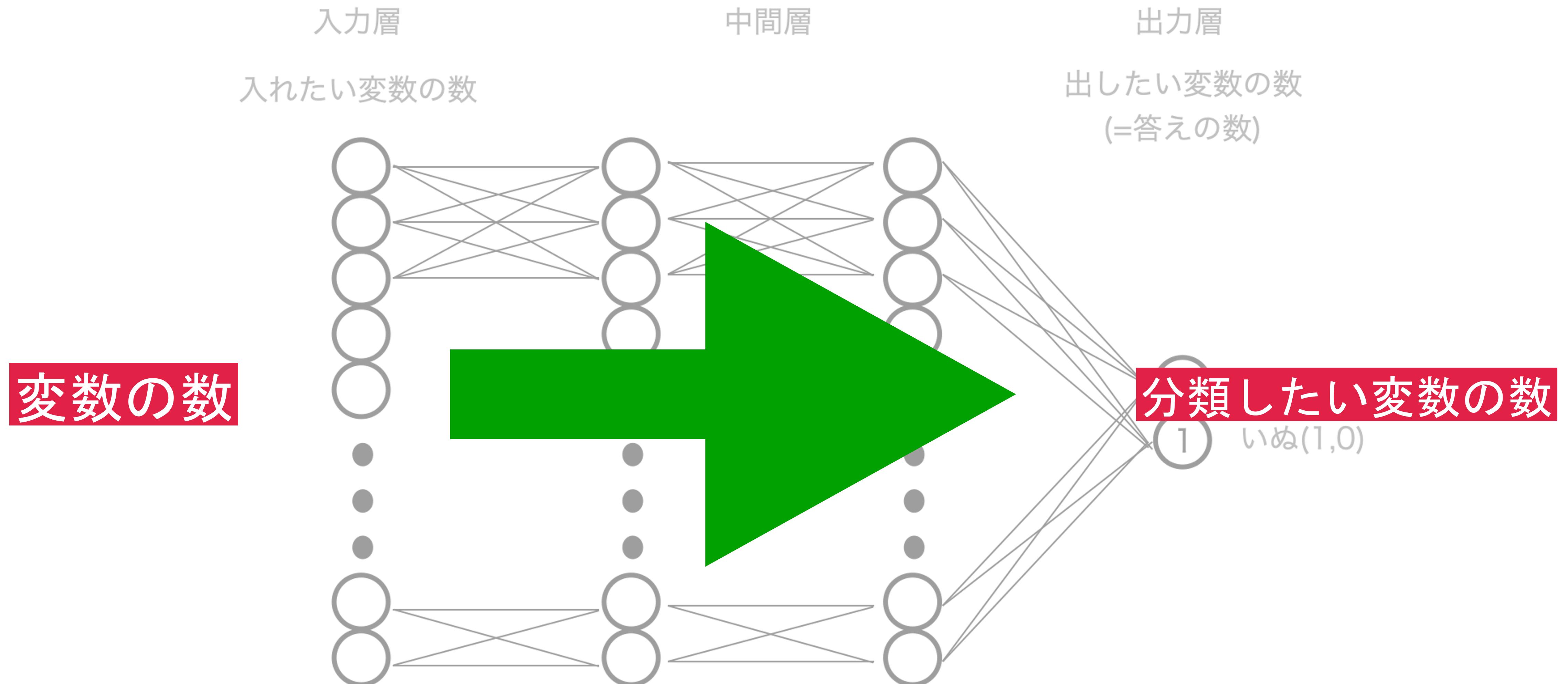
# ニューラルネットワークとは(軽く復習)



○が全てニューロン、繋がった線の数だけ数式(関数)が存在する。

使いたい変数の数を入力層のニューロンの数、出したい答えの数を出力層のニューロンの数にする

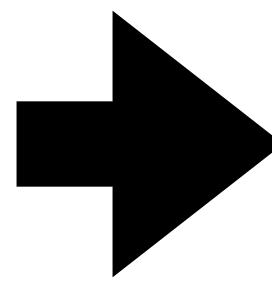
# ニューラルネットワークとは(軽く復習)



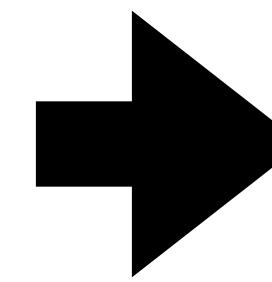
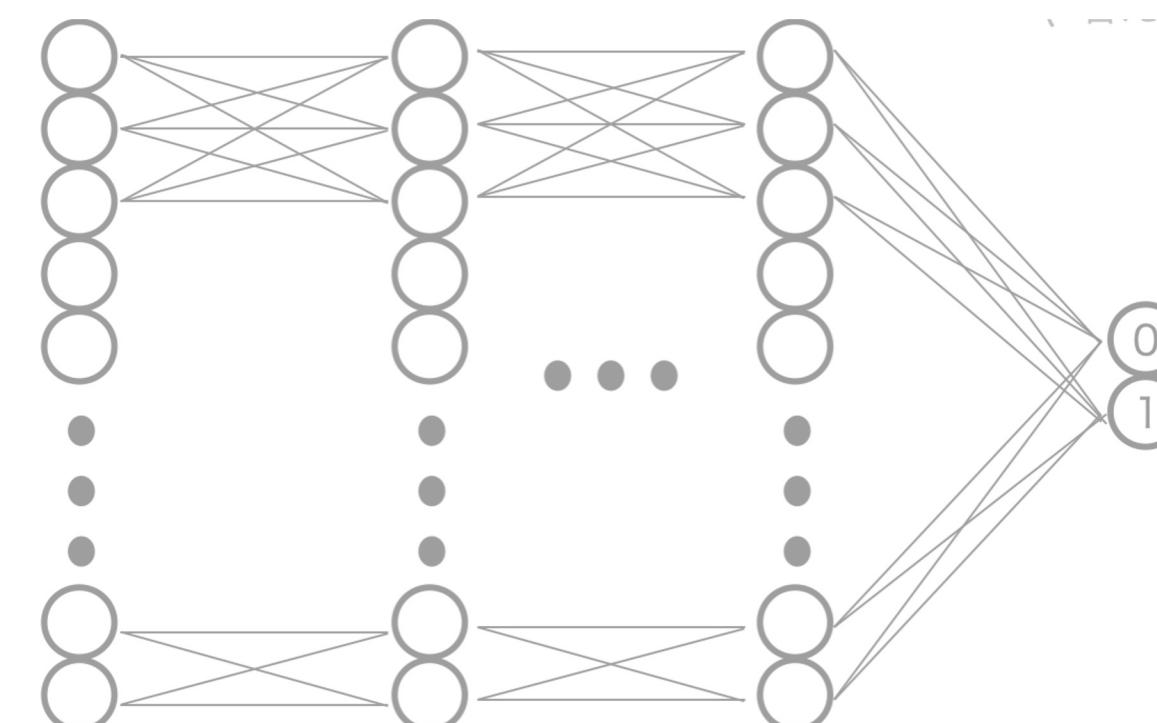
○が全てニューロン、繋がった線の数だけ数式(関数)が存在する。

使いたい変数の数を入力層のニューロンの数、出したい答えの数を出力層のニューロンの数にする

# アヤメのデータでの深層学習の実践



ニューラルネットワーク

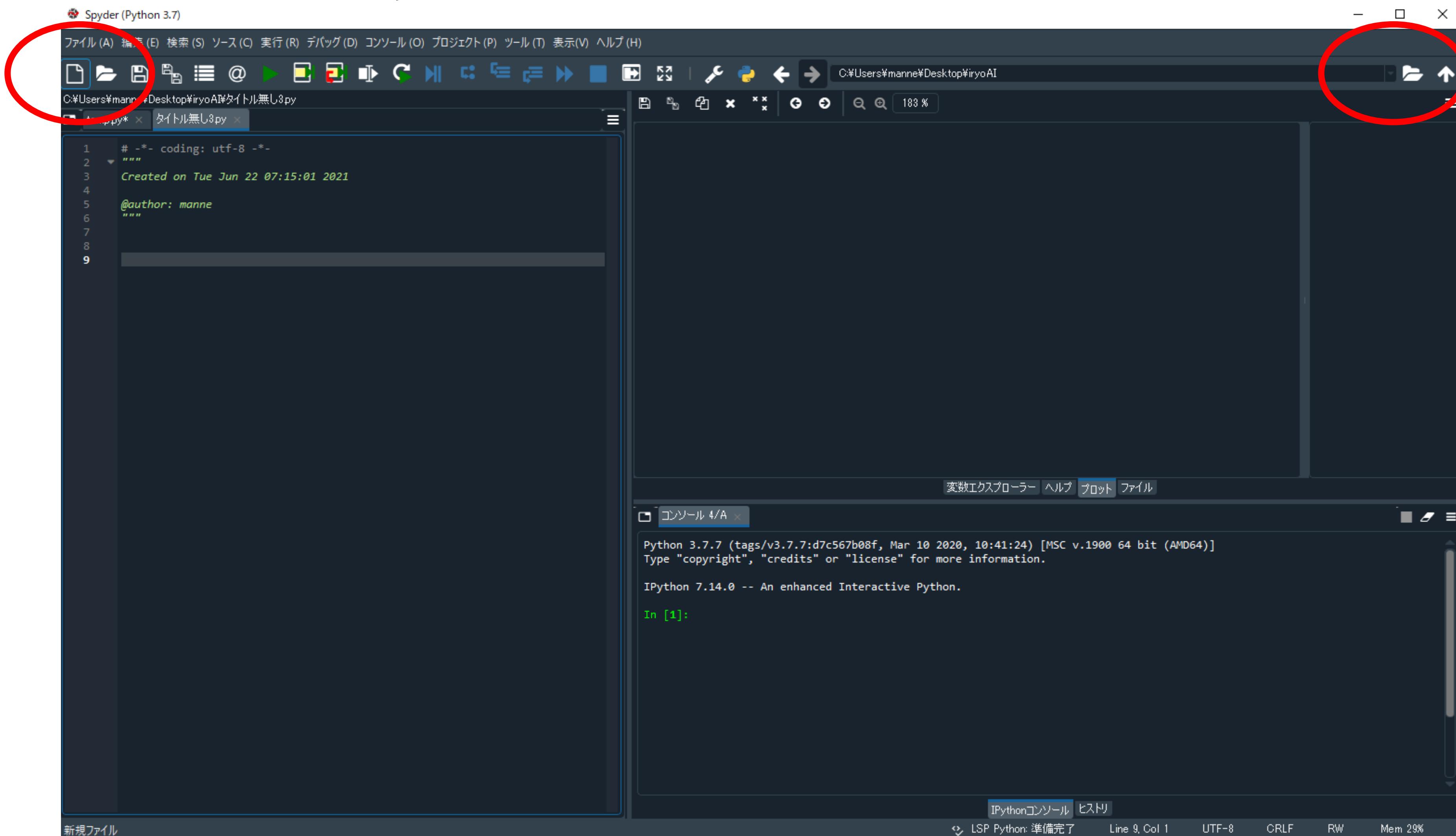


分類

ヒオウギアヤメorブルーフラッグ

# まず演習の準備をしましょう

- ・新規ファイル作成→iryoAIのディレクトリで”enshu4.py”
- ・作業場所をiryoAIに設定



# ライブラリのインストールとデータの読み込み・加工

## 4.txtの”1)アイリスデータを読み込む”の前半部分をコピーする

```
# 前半部分

import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Hiragino Maru Gothic Pro', 'Yu Gothic', 'Meirio']

from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense

import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'

# 後半部分

iris_train2 = pd.read_csv('4-1.csv')
print(iris_train2)
iris_train = pd.read_csv('4-1.csv').to_numpy()
print(iris_train)
x_train = iris_train[:, 0:4].astype('float')
y_train = iris_train[:, 4:5].astype('int')
iris_test = pd.read_csv('4-2.csv').to_numpy()
x_test = iris_test[:, 0:4].astype('float')
y_test = iris_test[:, 4:5].astype('int')
```

# ライブラリのインストールとデータの読み込み・加工

## 4.txtの”1)アイリスデータを読み込む”の前半部分

```
# 1)アイリスデータを読み込む
# 必要なライブラリのインポート
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.family'] ='sans-serif'
rcParams['font.sans-serif'] =['Hiragino Maru Gothic Pro', 'Yu Gothic', 'Meirio']

# 深層学習用のライブラリを取り込む
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense

# 以下のコード2行は本来必要ないが、anacondaでのOMP Abort エラーを防ぐために入れた
import os
os.environ['KMP_DUPLICATE_LIB_OK']=True
```

tensorflowおよびkerasという深層学習ライブラリを使用

# ライブラリのインストールとデータの読み込み・加工

## 4.txtの”1)アイリスデータを読み込む”の後半部分をコピーする

```
# 前半部分

import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Hiragino Maru Gothic Pro', 'Yu Gothic', 'Meirio']

from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense

import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'

# 後半部分

iris_train2 = pd.read_csv('4-1.csv')
print(iris_train2)
iris_train = pd.read_csv('4-1.csv').to_numpy()
print(iris_train)
x_train = iris_train[:, 0:4].astype('float')
y_train = iris_train[:, 4:5].astype('int')
iris_test = pd.read_csv('4-2.csv').to_numpy()
x_test = iris_test[:, 0:4].astype('float')
y_test = iris_test[:, 4:5].astype('int')
```

# ライブラリのインストールとデータの読み込み・加工

今回は学習用(4-1.csv)と検証用データ(4-2.csv)に分けて準備しています。  
順に読み込んで、特徴量(説明変数)と正解データ(目的変数)に分けます。

4-1.CSV

	A	B	C	D	E	F
1	がく片の長さ	がく片の幅	花びらの長さ	花びらの幅	教師データー	アヤメの種類
2	5.1	3.5	1.4	0.2	0	ヒオウギアヤメ
3	4.9	3	1.4	0.2	0	ヒオウギアヤメ
4	4.7	3.2	1.3	0.2	0	ヒオウギアヤメ
5	4.6	3.1	1.5	0.2	0	ヒオウギアヤメ
6	5	3.6	1.4	0.2	0	ヒオウギアヤメ
7	5.4	3.9	1.7	0.4	0	ヒオウギアヤメ
8	4.6	3.4	1.4	0.3	0	ヒオウギアヤメ
9	5	3.4	1.5	0.2	0	ヒオウギアヤメ
10	4.4	2.9	1.4	0.2	0	ヒオウギアヤメ
11	4.9	3.1	1.5	0.1	0	ヒオウギアヤメ
12	5.4	3.7	1.5	0.2	0	ヒオウギアヤメ
13	4.8	3.4	1.6	0.2	0	ヒオウギアヤメ
14	4.8	3	1.4	0.1	0	ヒオウギアヤメ
15	4.3	3	1.1	0.1	0	ヒオウギアヤメ
16	5.8	4	1.2	0.2	0	ヒオウギアヤメ
17	5.7	4.4	1.5	0.4	0	ヒオウギアヤメ
18	5.4	3.9	1.3	0.4	0	ヒオウギアヤメ

4-2.CSV

	A	B	C	D	E	F
1	がく片の長さ	がく片の幅	花びらの長さ	花びらの幅	教師データー	アヤメの種類
2	5	3.5	1.3	0.3	0	ヒオウギアヤメ
3	4.5	2.3	1.3	0.3	0	ヒオウギアヤメ
4	4.4	3.2	1.3	0.2	0	ヒオウギアヤメ
5	5	3.5	1.6	0.6	0	ヒオウギアヤメ
6	5.1	3.8	1.9	0.4	0	ヒオウギアヤメ
7	4.8	3	1.4	0.3	0	ヒオウギアヤメ
8	5.1	3.8	1.6	0.2	0	ヒオウギアヤメ
9	4.6	3.2	1.4	0.2	0	ヒオウギアヤメ
10	5.3	3.7	1.5	0.2	0	ヒオウギアヤメ
11	5	3.3	1.4	0.2	0	ヒオウギアヤメ
12	5.5	2.6	4.4	1.2	1	ブルーフラッグ
13	6.1	3	4.6	1.4	1	ブルーフラッグ
14	5.8	2.6	4	1.2	1	ブルーフラッグ
15	5	2.3	3.3	1	1	ブルーフラッグ
16	5.6	2.7	4.2	1.3	1	ブルーフラッグ

学習用データが80行、検証用データが20行  
アヤメの種類をヒオウギアヤメが0、ブルーフラッグが1と数値化している

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
iris_train = pd.read_csv('4-1.csv').to_numpy()
```

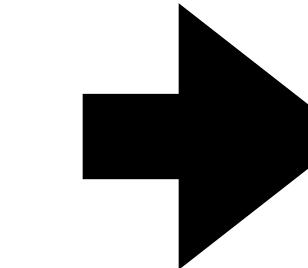
pd.read\_csv('csvファイル').to\_numpy()

→pandasのデータフレームをnumpy配列に変換する

```
iris_train2 = pd.read_csv('4-1.csv')
print(iris_train2)
```

```
iris_train = pd.read_csv('4-1.csv').to_numpy()
print(iris_train)
```

	がく片の長さ	がく片の幅	花びらの長さ	花びらの幅	教師データー	アヤメの種類
0	5.1	3.5	1.4	0.2	0	ヒオウギアヤメ
1	4.9	3.0	1.4	0.2	0	ヒオウギアヤメ
2	4.7	3.2	1.3	0.2	0	ヒオウギアヤメ
3	4.6	3.1	1.5	0.2	0	ヒオウギアヤメ
4	5.0	3.6	1.4	0.2	0	ヒオウギアヤメ
..	..	..	..	..	..	..
75	6.0	3.4	4.5	1.6	1	ブルーフラッグ
76	6.7	3.1	4.7	1.5	1	ブルーフラッグ
77	6.3	2.3	4.4	1.3	1	ブルーフラッグ
78	5.6	3.0	4.1	1.3	1	ブルーフラッグ
79	5.5	2.5	4.0	1.3	1	ブルーフラッグ
[80 rows x 6 columns]						



```
[[5.1 3.5 1.4 0.2 0 'ヒオウギアヤメ']
 [4.9 3.0 1.4 0.2 0 'ヒオウギアヤメ']
 [4.7 3.2 1.3 0.2 0 'ヒオウギアヤメ']
 [4.6 3.1 1.5 0.2 0 'ヒオウギアヤメ']
 [5.0 3.6 1.4 0.2 0 'ヒオウギアヤメ']
 [5.4 3.9 1.7 0.4 0 'ヒオウギアヤメ']
 [4.6 3.4 1.4 0.3 0 'ヒオウギアヤメ']
 [5.0 3.4 1.5 0.2 0 'ヒオウギアヤメ']
 [4.4 2.9 1.4 0.2 0 'ヒオウギアヤメ']
 [4.9 3.1 1.5 0.1 0 'ヒオウギアヤメ']
 [5.4 3.7 1.5 0.2 0 'ヒオウギアヤメ']
 [4.8 3.4 1.6 0.2 0 'ヒオウギアヤメ']
 [4.8 3.0 1.4 0.1 0 'ヒオウギアヤメ']]
```

numpy配列の2次元配列

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
iris_train = pd.read_csv('4-1.csv').to_numpy()
```

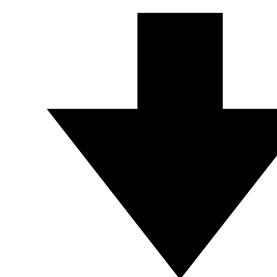
```
pd.read_csv('csvファイル').to_numpy()
```

→pandasのデータフレームをnumpy配列に変換する

```
iris_train2 = pd.read_csv('4-1.csv')  
print(iris_train2)
```

```
iris_train = pd.read_csv('4-1.csv').to_numpy()  
print(iris_train)
```

```
iris_train2 DataFrame (80, 6) Column names: がく片の長さ, がく片の幅, 花びらの長さ, 花びらの幅, 教師データ, アヤメの種類
```



```
iris_train Array of object (80, 6) ndarray object of numpy module
```

## ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
x_train = iris_train[:, 0:4]  
y_train = iris_train[:, 4:5]
```

### 特定の行、列を抜き出す

(numpy配列)[**〇〇:〇〇**, **〇〇:〇〇**]

(numpy配列)[**行の始まり:行の終わり(-1)**, **列の始まり:列の終わり(-1)** ]

1列目は0から数えるので、0:4は1列目から**4**列目になります。

“**:**”だけで数字がない場合は全ての行もしくは列を意味します

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

例)

```
data =  
np.array([[1,2,3],[4,5,6],[7,8,9]])  
print(data)
```

	0	1	2
0	[1	2	3]
1	[4	5	6]
2	[7	8	9]]

3 × 3の二次元配列を作成

```
x = data[0:2,0:2]  
print(x)
```

	0	1
0	[1	2]
1	[4	5]]

1 次元目(=行)も2次元目(=列)も"0:2"なので  
0番目から1番目の行と列を取り出す

```
x2 = data[1:3,1:2]  
print(x2)
```

	1
1	[5]
2	[8]]

行が1番目から2番目まで、  
列が1番目から1番目までを取り出す

```
x3 = data[1:2,:]  
print(x3)
```

	0	1	2
1	[4	5	6]]

行が1番目から1番目まで、  
列は全ての列を取り出す

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
x_train = iris_train[ :, 0:4]  
y_train = iris_train[ :, 4:5]
```

0 1 2 3 4 5

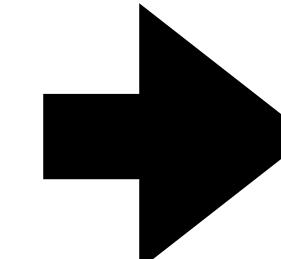
0 [[5.1 3.5 1.4 0.2 0 'ヒオウギアヤメ']  
1 [4.9 3.0 1.4 0.2 0 'ヒオウギアヤメ']  
2 [4.7 3.2 1.3 0.2 0 'ヒオウギアヤメ']  
3 [4.6 3.1 1.5 0.2 0 'ヒオウギアヤメ']  
4 [5.0 3.6 1.4 0.2 0 'ヒオウギアヤメ']  
5 [5.4 3.9 1.7 0.4 0 'ヒオウギアヤメ']  
[4.6 3.4 1.4 0.3 0 'ヒオウギアヤメ']  
[5.0 3.4 1.5 0.2 0 'ヒオウギアヤメ']  
[4.4 2.9 1.4 0.2 0 'ヒオウギアヤメ']  
[4.9 3.1 1.5 0.1 0 'ヒオウギアヤメ']  
[5.4 3.7 1.5 0.2 0 'ヒオウギアヤメ']  
[4.8 3.4 1.6 0.2 0 'ヒオウギアヤメ']  
[4.8 3.0 1.4 0.1 0 'ヒオウギアヤメ']

全ての行と1列目から4列目

[ [5.1 3.5 1.4 0.2]  
[4.9 3. 1.4 0.2]  
[4.7 3.2 1.3 0.2]  
[4.6 3.1 1.5 0.2]  
[5. 3.6 1.4 0.2]  
[5.4 3.9 1.7 0.4]  
[4.6 3.4 1.4 0.3]  
[5. 3.4 1.5 0.2]  
[4.4 2.9 1.4 0.2]  
[4.9 3.1 1.5 0.1]  
[5.4 3.7 1.5 0.2]  
[4.8 3.4 1.6 0.2]  
[4.8 3. 1.4 0.1]

全ての行と5列目から5列目

[ [0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]  
[0]



iris\_train(学習用の説明変数+目的変数)

x\_train(学習用の説明変数)

y\_train(学習用の目的変数)

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
x_train = iris_train[:, 0:4].astype('float')
y_train = iris_train[:, 4:5].astype('int')
```

numpy配列.astype('型')

→配列の型を指定する

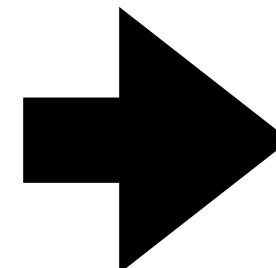
.astype('float')は中身を小数、.astype('int')は中身を整数に指定する指示です

print(iris\_train)

```
[[5.1 3.5 1.4 0.2 0 'ヒオウギアヤメ']
 [4.9 3.0 1.4 0.2 0 'ヒオウギアヤメ']
 [4.7 3.2 1.3 0.2 0 'ヒオウギアヤメ']
 [4.6 3.1 1.5 0.2 0 'ヒオウギアヤメ']
 [5.0 3.6 1.4 0.2 0 'ヒオウギアヤメ']
 [5.4 3.9 1.7 0.4 0 'ヒオウギアヤメ']
 [4.6 3.4 1.4 0.3 0 'ヒオウギアヤメ']
 [5.0 3.4 1.5 0.2 0 'ヒオウギアヤメ']
 [4.4 2.9 1.4 0.2 0 'ヒオウギアヤメ']
 [4.9 3.1 1.5 0.1 0 'ヒオウギアヤメ']
 [5.4 3.7 1.5 0.2 0 'ヒオウギアヤメ']
 [4.8 3.4 1.6 0.2 0 'ヒオウギアヤメ']
 [4.8 3.0 1.4 0.1 0 'ヒオウギアヤメ']]
```

print(x\_train)

```
[[5.1 3.5 1.4 0.2]
 [4.9 3. 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5. 3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5. 3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3. 1.4 0.1]]
```



データに小数も含まれている

print(y\_train)

```
[[0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]]
```

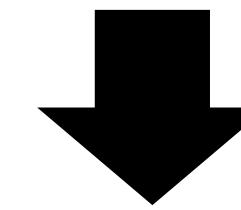
データは全て整数

# ライブラリのインストールとデータの読み込み・加工(1の後半部分)

```
x_train = iris_train[:, 0:4].astype('float')  
y_train = iris_train[:, 4:5].astype('int')
```

astype()がない場合

iris_train	Array of object	(80, 6)	ndarray object of numpy module
x_train	Array of object	(80, 4)	ndarray object of numpy module
y_train	Array of object	(80, 1)	ndarray object of numpy module



astype()がある場合

iris_train	Array of object	(80, 6)	ndarray object of numpy module
x_train	Array of float64	(80, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
y_train	Array of int32	(80, 1)	[[0] [0]]

## 加工したデータの確認

“2) 加工したデータの確認”を実行して中身を見てみよう

```
# 2) 加工したデータの確認
print(iris_train)
print(x_train)
print(y_train)
print(iris_test)
print(x_test)
print(y_test)
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

全て2次元配列

```
In [7]: print(x_test)
[[5.  3.5 1.3 0.3]
 [4.5 2.3 1.3 0.3]
 [4.4 3.2 1.3 0.2]
 [5.  3.5 1.6 0.6]
 [5.1 3.8 1.9 0.4]
 [4.8 3.  1.4 0.3]
 [5.1 3.8 1.6 0.2]
 [4.6 3.2 1.4 0.2]
 [5.3 3.7 1.5 0.2]
 [5.  3.3 1.4 0.2]
 [5.5 2.6 4.4 1.2]
 [6.1 3.  4.6 1.4]
 [5.8 2.6 4.  1.2]
 [5.  2.3 3.3 1. ]
 [5.6 2.7 4.2 1.3]
 [5.7 3.  4.2 1.2]
 [5.7 2.9 4.2 1.3]
 [6.2 2.9 4.3 1.3]
 [5.1 2.5 3.  1.1]
 [5.7 2.8 4.1 1.3]]
```

```
In [8]: print(y_test)
[[0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]]
```

```
In [9]: print(x_train.shape)
(80, 4)
```

```
In [10]: print(y_train.shape)
(80, 1)
```

```
In [11]: print(x_test.shape)
(20, 4)
```

```
In [12]: print(y_test.shape)
(20, 1)
```

学習用データは80行 4列  
検証用データは20行 1列

# 加工したデータの確認

名前	型	サイズ	値
iris_test	Array of object	(20, 6)	ndarray object of numpy module
iris_train	Array of object	(80, 6)	ndarray object of numpy module
x_test	Array of float64	(20, 4)	<code>[[5.  3.5 1.3 0.3]  [4.5 2.3 1.3 0.3]]</code>
x_train	Array of float64	(80, 4)	<code>[[5.1 3.5 1.4 0.2]  [4.9 3.  1.4 0.2]]</code>
y_test	Array of int32	(20, 1)	<code>[[0]  [0]]</code>
y_train	Array of int32	(80, 1)	<code>[[0]  [0]]</code>

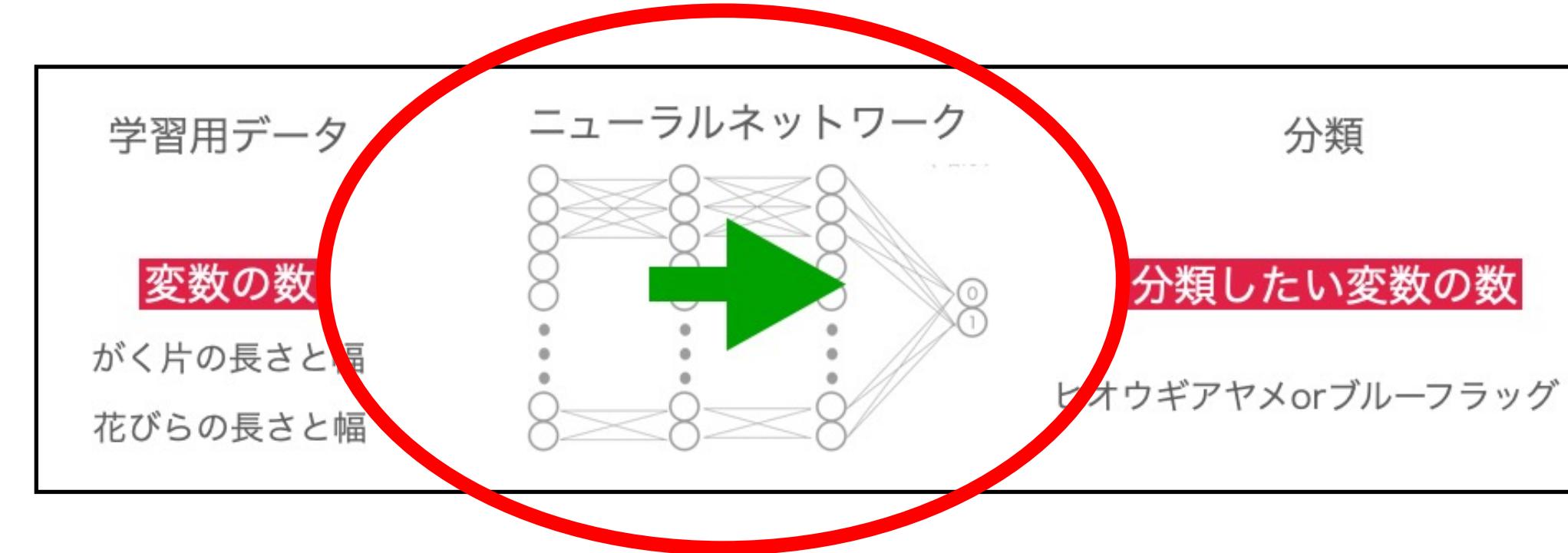
x\_train - NumPy オブジェクト配列

	0	1	2	3
0	5.1	3.5	1.4	0.2
1	4.9	3	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5	3.6	1.4	0.2

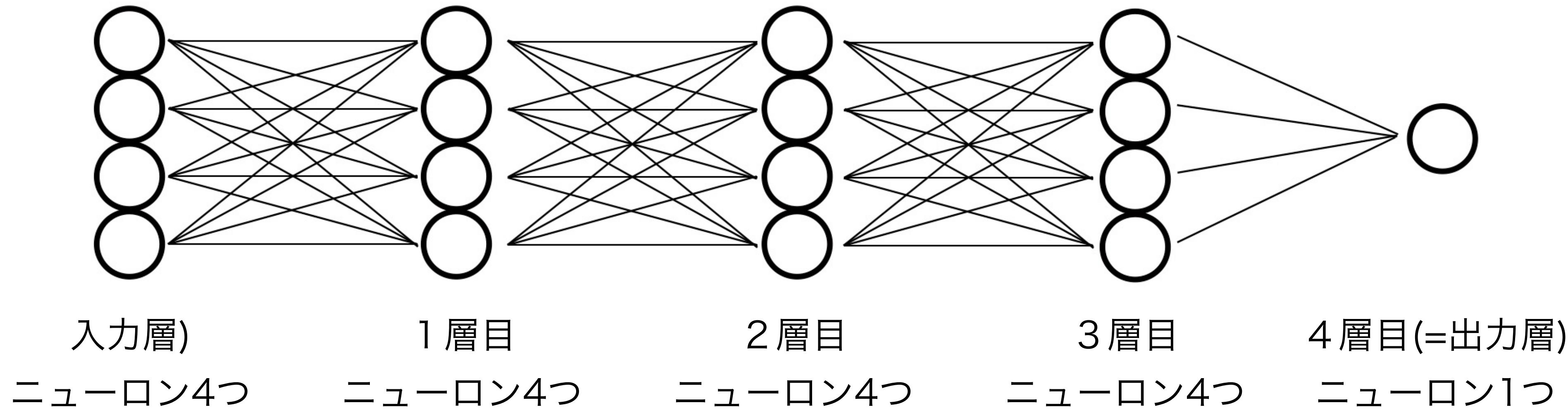
y\_train - NumPy オブジェクト配列

	0
0	0
1	0
2	0
3	0
4	0

### 3) 学習モデルの作成(ニューラルネットワーク)



今回作成する学習モデル



今回はシンプルなネットワークにするため、出力層は1つにしてます。  
(2値分類の場合、最後に出る値を確率 $p$ が出ると自動的にもう1つの確率は $1-p$ になります。)

### 3) 学習モデルの作成(ニューラルネットワーク)

ニューラルネットワークを作成していく

モデル名 = Sequential()

ニューラルネットワークを作るモデルも沢山ある中で、  
今回はKeras(ケラス)のSequentialモデルを使用します

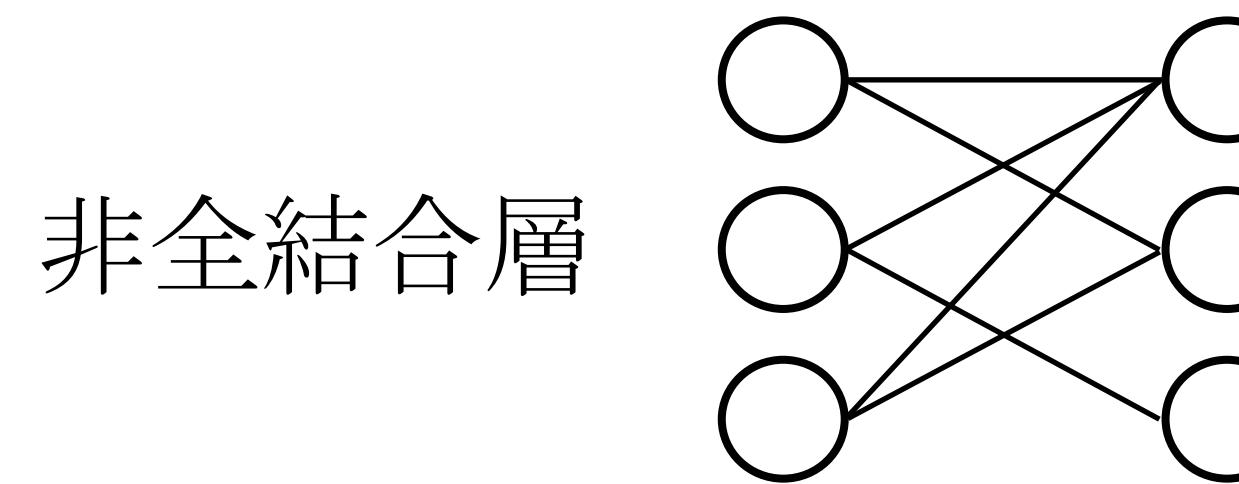
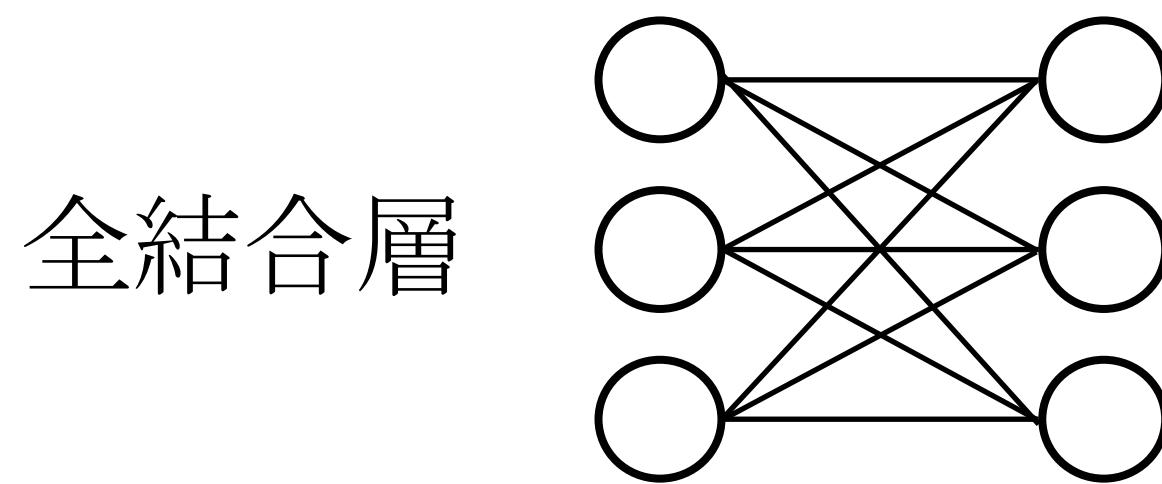
```
dl_model = Sequential()
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(1, activation='sigmoid'))
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])
dl_model.summary()
```

今回はモデル名を”dl\_model”とする

### 3) 学習モデルの作成(ニューラルネットワーク)

モデル名.add()で層の追加を行う

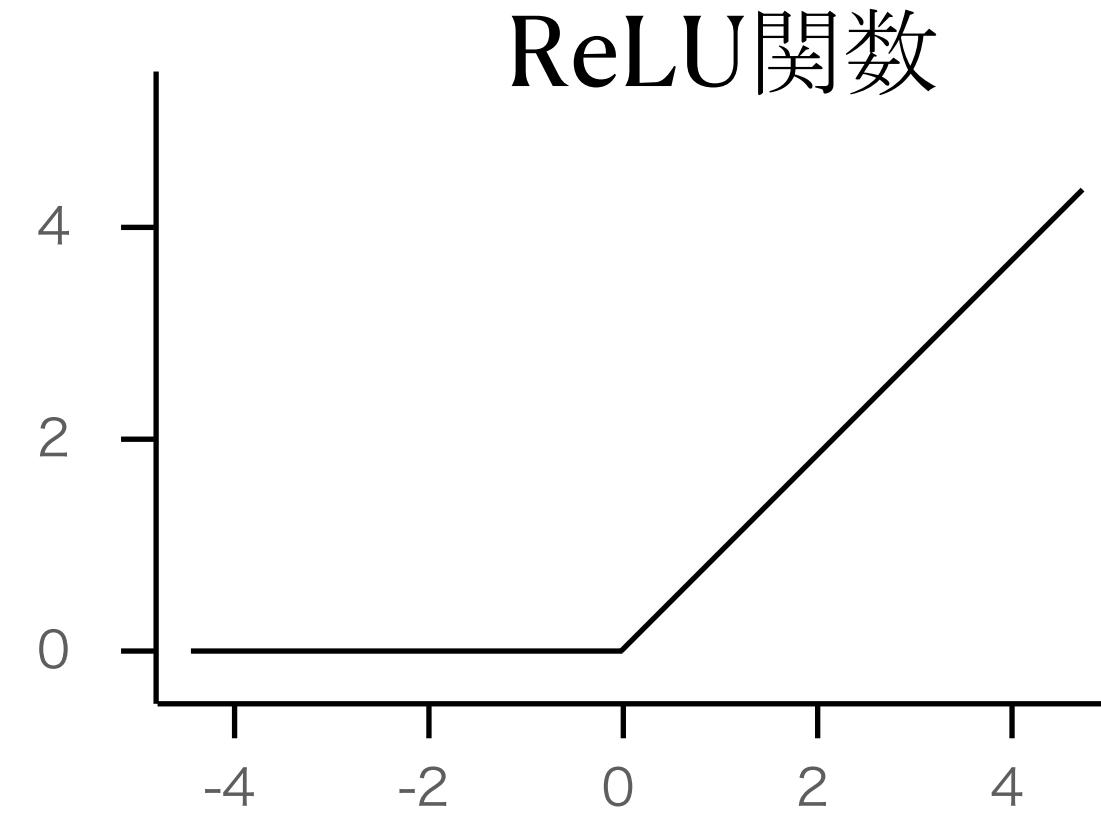
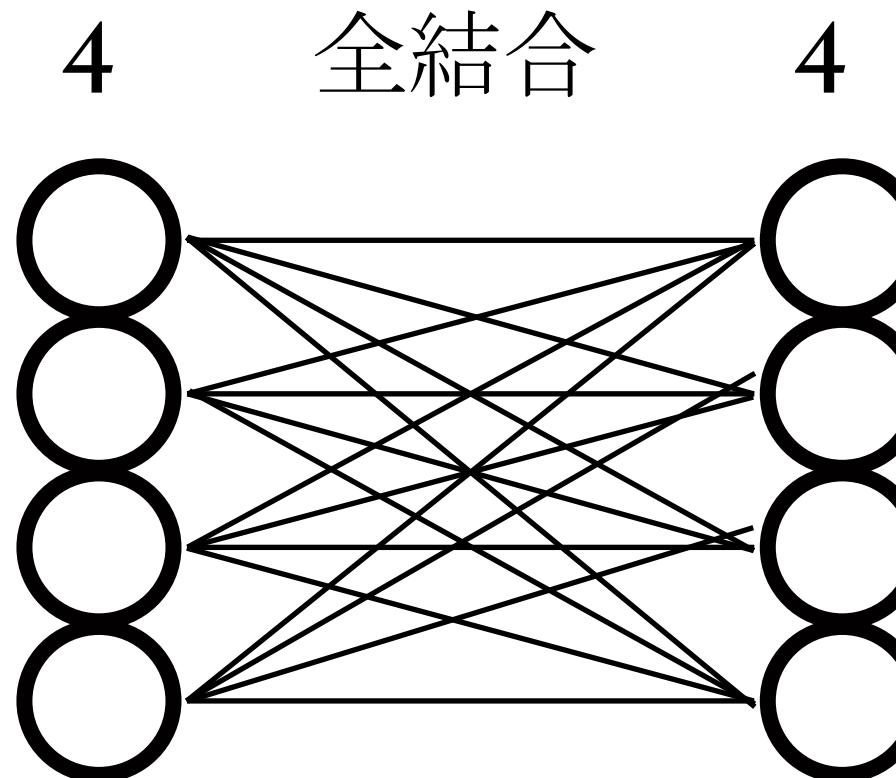
Denseは全ての入力が全てのニューロンと結合している状態(=全結合層という)



```
dl_model = Sequential()
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(1, activation='sigmoid'))
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])
dl_model.summary()
```

### 3) 学習モデルの作成(ニューラルネットワーク)

model.add(Dense(出力の変数の数, activation='活性化関数', input\_shape=(入力の変数の数,)))

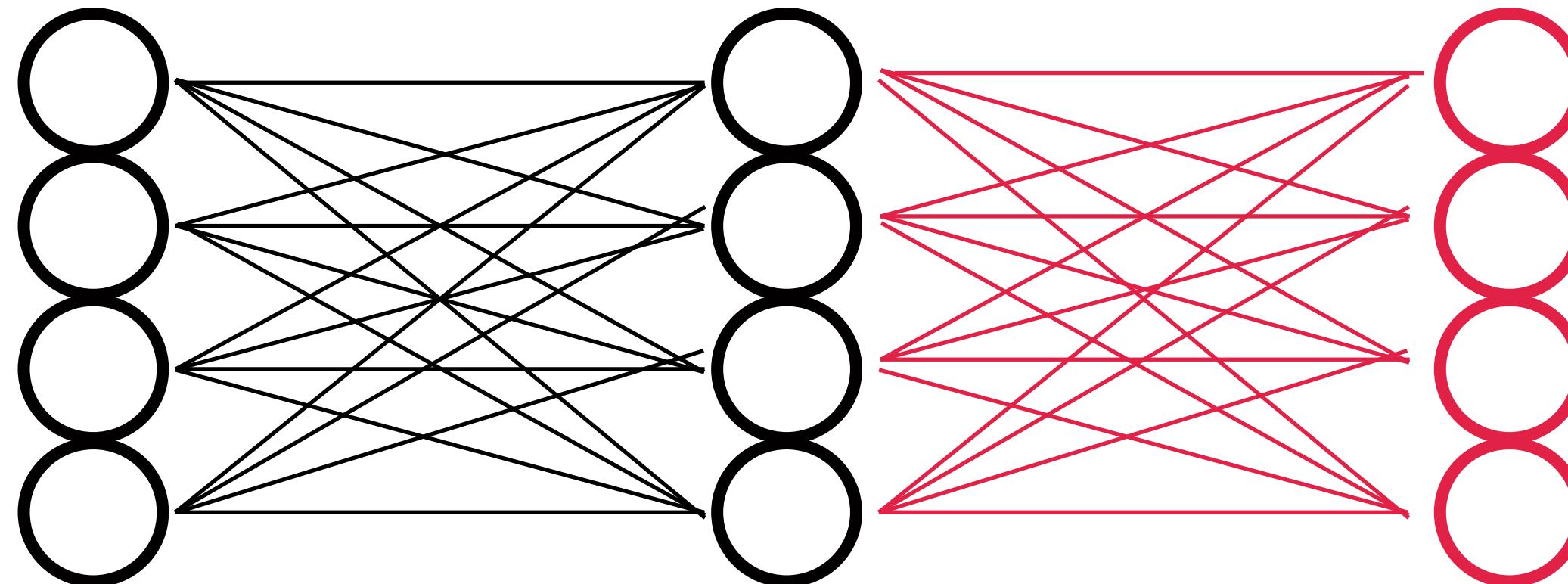


入力の変数4つ、出力の変数4つ  
活性化関数はReLUを選択(全結合層)

(入力の4つの変数は、がく片の  
長さと幅、花びらの長さと幅)

```
dl_model = Sequential()  
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(1, activation='sigmoid'))  
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])  
dl_model.summary()
```

### 3) 学習モデルの作成(ニューラルネットワーク)

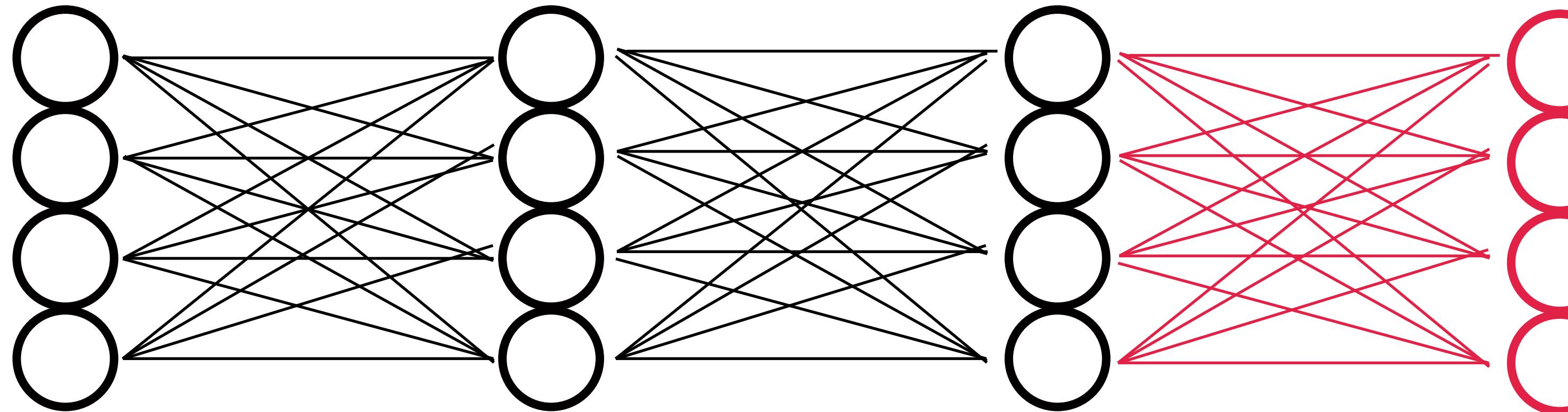


model.add()でさらに層の追加

2回目以降はinput\_shapeは不要  
入力の変数はそのまま(4つ)、出力の変数4つ  
ReLUという(活性化)関数を選択

```
dl_model = Sequential()  
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(1, activation='sigmoid'))  
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])  
dl_model.summary()
```

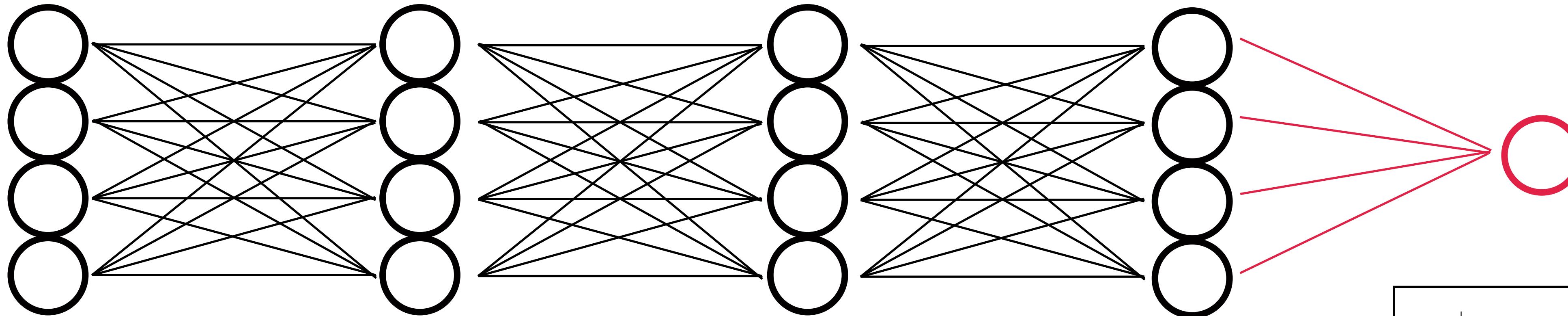
### 3) 学習モデルの作成(ニューラルネットワーク)



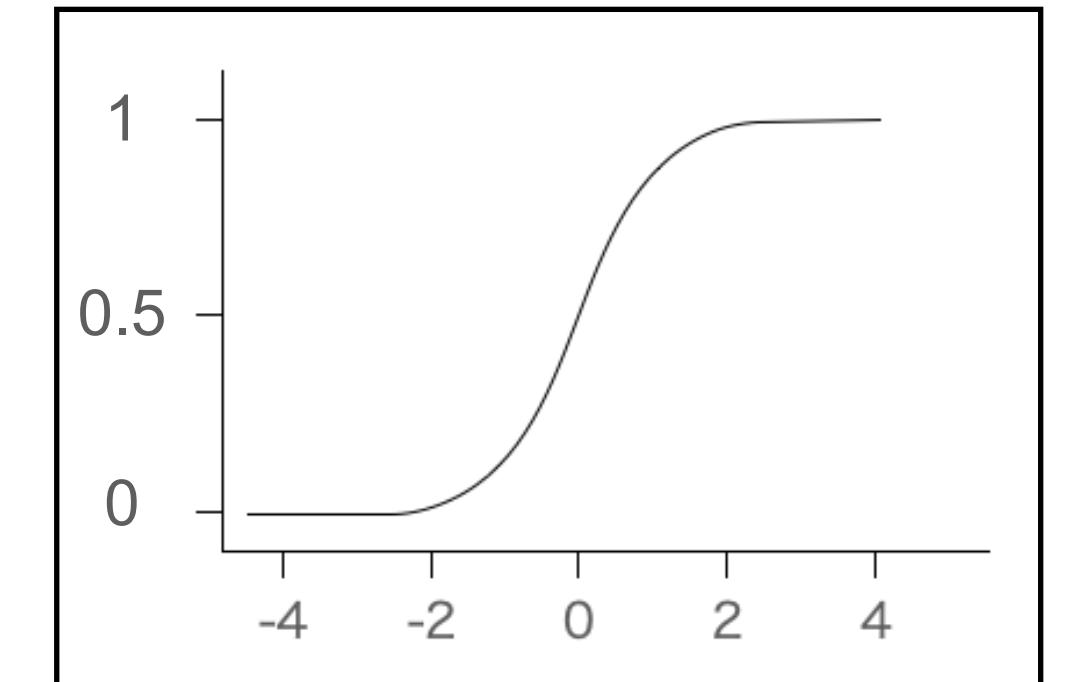
入力の変数はそのまま(4つ)、出力の変数4つ  
ReLUという(活性化)関数を選択

```
dl_model = Sequential()  
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(1, activation='sigmoid'))  
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])  
dl_model.summary()
```

### 3) 学習モデルの作成(ニューラルネットワーク)



```
dl_model = Sequential()  
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(1, activation='sigmoid'))  
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])  
dl_model.summary()
```



出力の変数1つ  
シグモイド(活性化)関数を選択

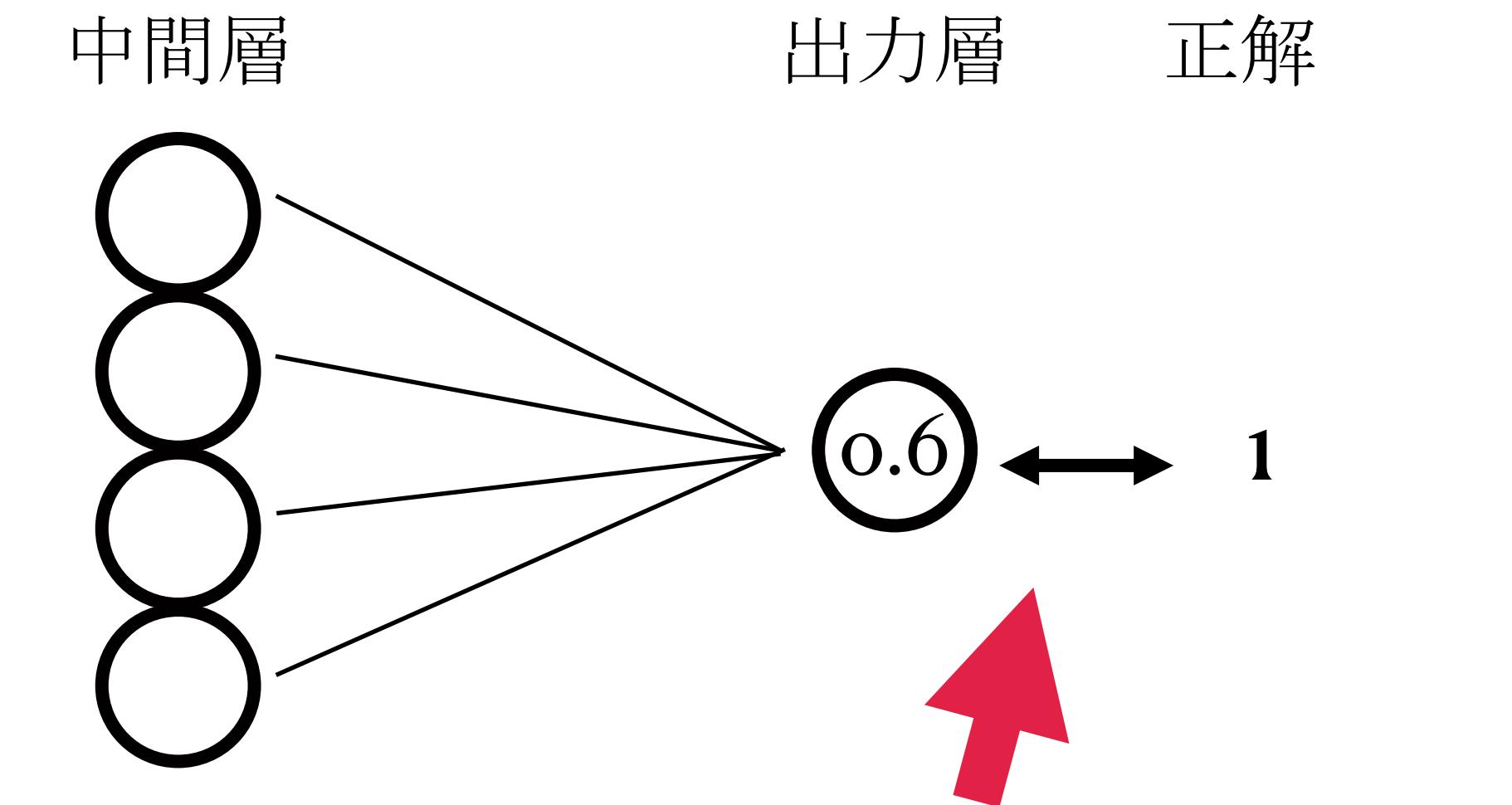
### 3) 学習モデルの作成(ニューラルネットワーク)

model.compile()で、学習時の評価方法の選択をする

loss = “損失関数”, optimizer = ‘最適化関数’, metrics=[“評価指標”]

今回は2クラス分類なのでbinary\_crossentropyを選択  
評価指標は正解率を示すaccuracyを選択

```
dl_model = Sequential()  
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(4, activation='relu'))  
dl_model.add(Dense(1, activation='sigmoid'))  
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['accuracy'])  
dl_model.summary()
```



この誤差を0に近づけるように、  
誤差逆伝播を行い各パラメーターを調整

(今回はAdamという最適化アルゴリズムを使用)

### 3) 学習モデルの作成(ニューラルネットワーク)

dl\_model.summary()は、  
作成した学習モデルを要約する

Model: "sequential"		
Layer (type)	Output Shape	Param #
dense (Dense)	(None, 4)	20
dense_1 (Dense)	(None, 4)	20
dense_2 (Dense)	(None, 4)	20
dense_3 (Dense)	(None, 1)	5
<hr/>		
Total params: 65		
Trainable params: 65		
Non-trainable params: 0		
<hr/>		

```
dl_model = Sequential()
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(1, activation='sigmoid'))
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])
dl_model.summary()
```

### 3) 学習モデルの作成(ニューラルネットワーク)

“ 3)神経回路の作成をコピーして実行してみよう

```
# 3)神経回路の作成
dl_model = Sequential()
dl_model.add(Dense(4, activation='relu', input_shape=(4,)))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(4, activation='relu'))
dl_model.add(Dense(1, activation='sigmoid'))
dl_model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=["accuracy"])
dl_model.summary()
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 4)	20
dense_1 (Dense)	(None, 4)	20
dense_2 (Dense)	(None, 4)	20
dense_3 (Dense)	(None, 1)	5
<hr/>		
Total params:	65	
Trainable params:	65	
Non-trainable params:	0	

## 4 ) 学習用データでの学習

```
history = dl_model.fit(x_train, y_train, epochs=300)
```

historyに学習過程を記録する。 300回学習させる

epochs

1エポックは1試行のことで、学習用データを1通り使って1エポックと数える  
ここでは300回学習させている。

## 4 ) 学習用データでの学習

```
history = dl_model.fit(x_train, y_train, epochs=300)
```

historyに学習過程を記録する。 300回学習させる

epochs

1エポックは1試行のこと、学習用データを1通り使って1エポックと数える  
ここでは300回学習させている。

```
80/80 [=====] - 0s 76us/sample - loss: 0.0231 - accuracy: 1.0000
Epoch 290/300
80/80 [=====] - 0s 69us/sample - loss: 0.0228 - accuracy: 1.0000
Epoch 291/300
80/80 [=====] - 0s 66us/sample - loss: 0.0225 - accuracy: 1.0000
Epoch 292/300
80/80 [=====] - 0s 66us/sample - loss: 0.0222 - accuracy: 1.0000
Epoch 293/300
80/80 [=====] - 0s 76us/sample - loss: 0.0219 - accuracy: 1.0000
Epoch 294/300
80/80 [=====] - 0s 66us/sample - loss: 0.0216 - accuracy: 1.0000
Epoch 295/300
80/80 [=====] - 0s 60us/sample - loss: 0.0214 - accuracy: 1.0000
Epoch 296/300
80/80 [=====] - 0s 64us/sample - loss: 0.0211 - accuracy: 1.0000
Epoch 297/300
80/80 [=====] - 0s 66us/sample - loss: 0.0208 - accuracy: 1.0000
Epoch 298/300
80/80 [=====] - 0s 74us/sample - loss: 0.0206 - accuracy: 1.0000
Epoch 299/300
80/80 [=====] - 0s 82us/sample - loss: 0.0203 - accuracy: 1.0000
Epoch 300/300
80/80 [=====] - 0s 90us/sample - loss: 0.0201 - accuracy: 1.0000
```

実行するとコンソール画面に学習過程が出力される  
loss: 学習用データの誤差、0に近いほどよい  
acc: 学習用データの正解率、1に近いほどよい

## 5) テスト用データでの検証

```
score = dl_model.evaluate(x_test, y_test)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

(変数名) = dl\_model.evaluate(x\_test, y\_test)  
→誤差と正解率が算出される

誤差→(変数名)[0]、正解率→(変数名)[1]

```
print("Test loss:", score[0])
Test loss: 0.009264961816370487
```

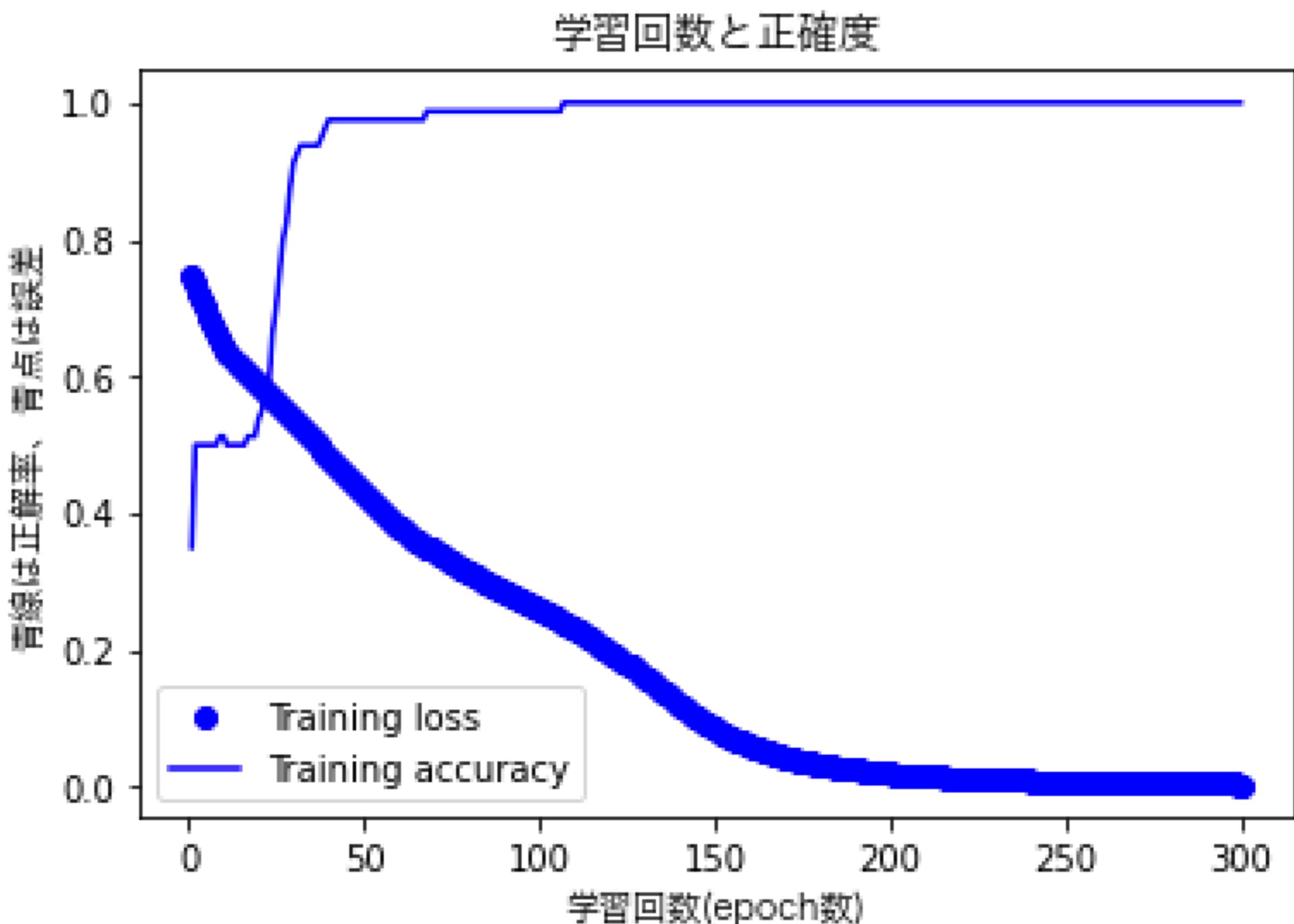
```
print("Test accuracy:", score[1])
Test accuracy: 1.0
```

誤差0.9%、正解率100%という結果

# 結果の可視化

“6) 学習過程をグラフ表示する”を実行しよう

```
# Loss(正解との誤差)をloss_valuesに入れる  
loss_values = history.history['loss']  
  
# 正確度をaccに入れる  
acc = history.history['accuracy']  
  
# 1からepoch数までのリストを作る  
epochlist = range(1, len(loss_values) + 1)  
  
# Loss(正解との誤差)のグラフを作る  
# 'bo'は青点  
plt.plot(epochlist, loss_values, 'bo', label='Training loss')  
  
# 正確率のグラフを作る  
# 'b'は青い線  
plt.plot(epochlist, acc, 'b', label='Training accuracy')  
  
plt.title('学習回数と正確度')  
plt.ylabel('青線は正解率、青点は誤差')  
plt.xlabel('学習回数(epoch数)')  
plt.legend()  
plt.show()
```



学習する度に正解率が上がって誤差が下がっていることが分かる

# 結果の可視化(補足)

## “6) 学習過程をグラフ表示する”を実行しよう

```
# Loss(正解との誤差)をloss_valuesに入れる  
loss_values = history.history['loss']  
  
# 正確度をaccに入れる  
acc = history.history['accuracy']  
  
# 1からepoch数までのリストを作る  
epochlist = range(1, len(loss_values) + 1)  
  
# Loss(正解との誤差)のグラフを作る  
# 'bo'は青点  
plt.plot(epochlist, loss_values, 'bo', label='Training loss')  
  
# 正確率のグラフを作る  
# 'b'は青い線  
plt.plot(epochlist, acc, 'b', label='Training accuracy')  
  
plt.title('学習回数と正確度')  
plt.ylabel('青線は正解率、青点は誤差')  
plt.xlabel('学習回数(epoch数)')  
plt.legend()  
plt.show()
```

history = model.fit(x\_train, y\_train, epochs=300)

kerasのfit()は各エポック毎の正解率、誤差を保存出来る

history.history["accuracy"]が正解率のリスト

history.history[["loss"]]が誤差のリスト

```
loss_values = history.history['loss']  
acc = history.history['accuracy']
```

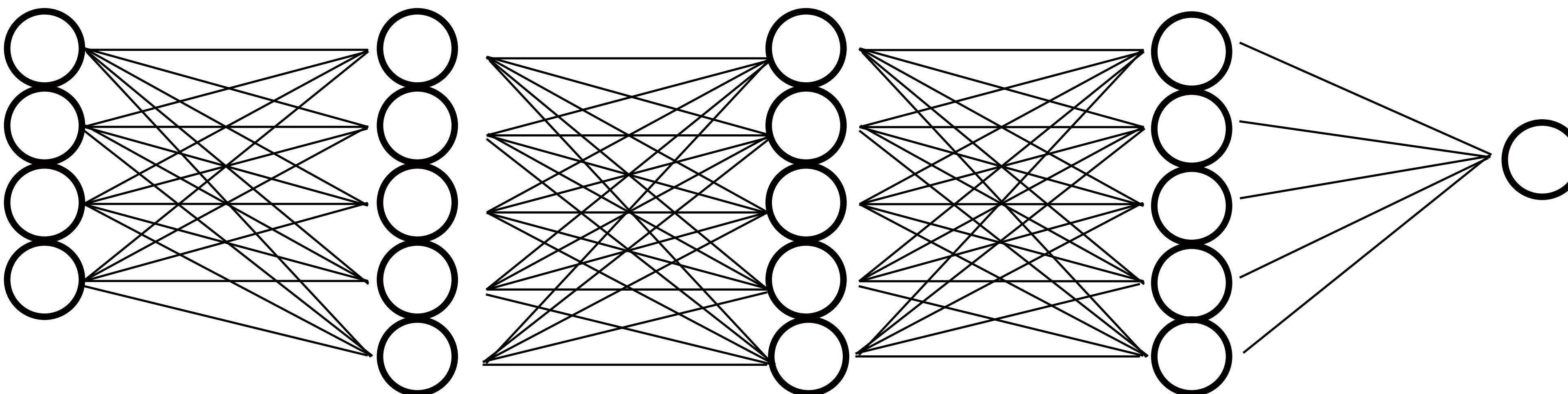
epochlist = range(1, len(loss\_values) + 1)

→range(1, 301) :1から300までの範囲を指定

```
plt.plot(epochlist, loss_values, 'bo', label='Training loss')  
→X軸が1から300(学習回数)、Y軸が各回の誤差で直線で結ぶ
```

# 自分たちで実践してみよう

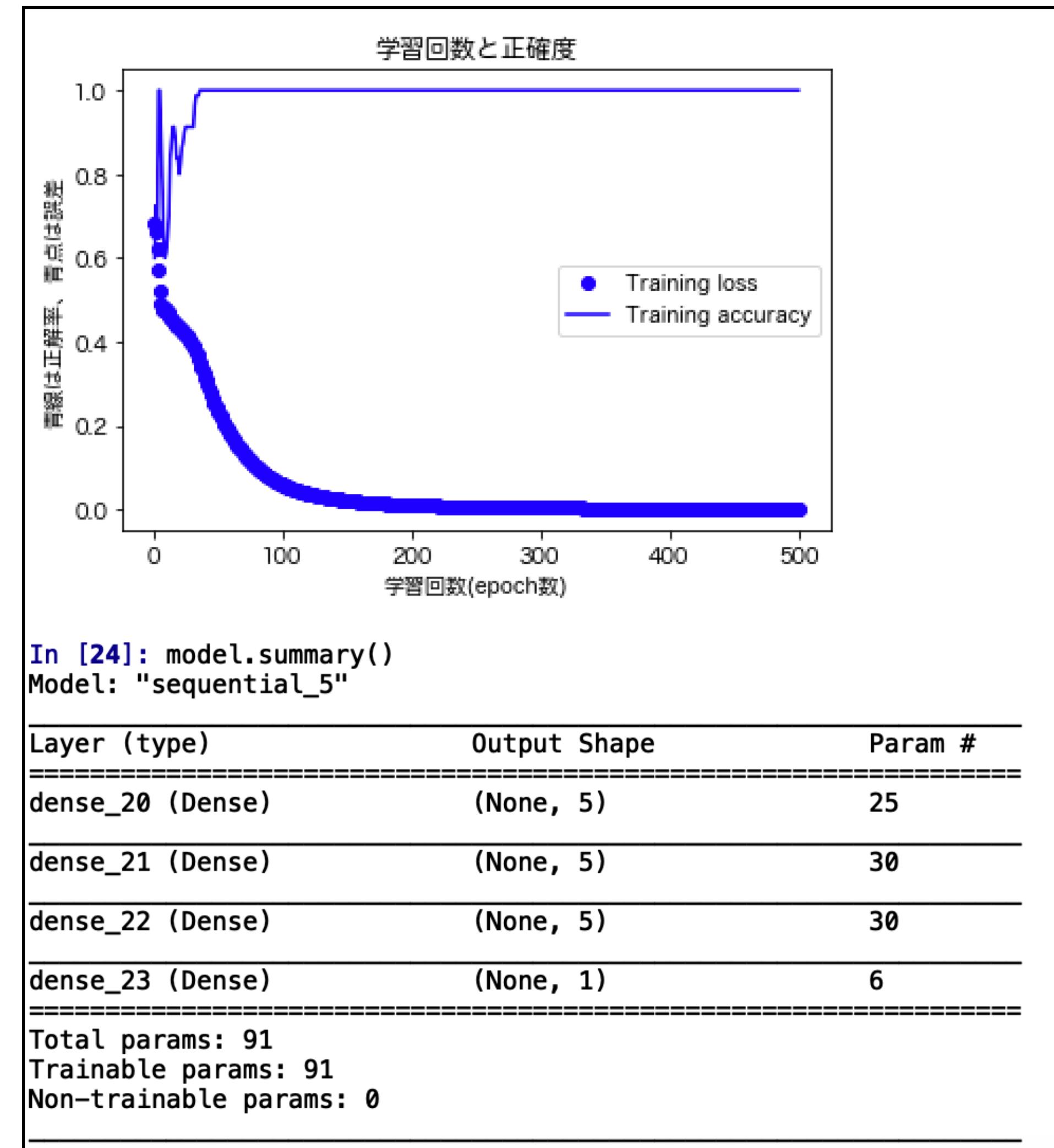
1層目から3層目のニューロンを1つ増やして、  
学習の回数を500回にして実践してみよう



# 自分たちで実践してみよう

このサマリーと学習過程が表示されていれば正しく実行出来ています。

```
3/3 [=====] - 0s 912us/step - loss: 0.0013 - accuracy: 1.0000
Epoch 480/500
3/3 [=====] - 0s 1ms/step - loss: 0.0014 - accuracy: 1.0000
Epoch 481/500
3/3 [=====] - 0s 1ms/step - loss: 0.0013 - accuracy: 1.0000
Epoch 482/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 483/500
3/3 [=====] - 0s 959us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 484/500
3/3 [=====] - 0s 1ms/step - loss: 0.0014 - accuracy: 1.0000
Epoch 485/500
3/3 [=====] - 0s 955us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 486/500
3/3 [=====] - 0s 995us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 487/500
3/3 [=====] - 0s 1ms/step - loss: 0.0014 - accuracy: 1.0000
Epoch 488/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 489/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 490/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 491/500
3/3 [=====] - 0s 1ms/step - loss: 0.0013 - accuracy: 1.0000
Epoch 492/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 493/500
3/3 [=====] - 0s 1ms/step - loss: 0.0013 - accuracy: 1.0000
Epoch 494/500
3/3 [=====] - 0s 1ms/step - loss: 0.0011 - accuracy: 1.0000
Epoch 495/500
3/3 [=====] - 0s 905us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 496/500
3/3 [=====] - 0s 952us/step - loss: 0.0013 - accuracy: 1.0000
Epoch 497/500
3/3 [=====] - 0s 991us/step - loss: 0.0013 - accuracy: 1.0000
Epoch 498/500
3/3 [=====] - 0s 1ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 499/500
3/3 [=====] - 0s 900us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 500/500
3/3 [=====] - 0s 933us/step - loss: 0.0013 - accuracy: 1.0000
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_test_function.<locals>.test_function at 0x7fb95879dcb0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.
Test loss: 0.003523191437125206
Test accuracy: 1.0
```



# 課題

- ・中間層の2層目をニューロン6個、3層目をニューロン3個にして実行しなさい  
(モデル名).summary()の出力結果と図を提出して下さい。(エポック数300)
- ・(余力がある人は)  
上のモデルを使って新たなアヤメのデータでブルーフラッグである確率を算出しなさい

No.	がく片の長さ	がく片の幅	花びらの長さ	花びらの幅
1	5.3	2.4	4.8	1.5
2	4.4	2.5	1.8	0.4

ヒント: import numpy as np

(変数) = model.predict((調べたいNumpy配列))

print(変数) = [[(1つ目の確率)]、[(2つ目の確率)], [ … ], [ … ], … ]

## 課題の提出方法

