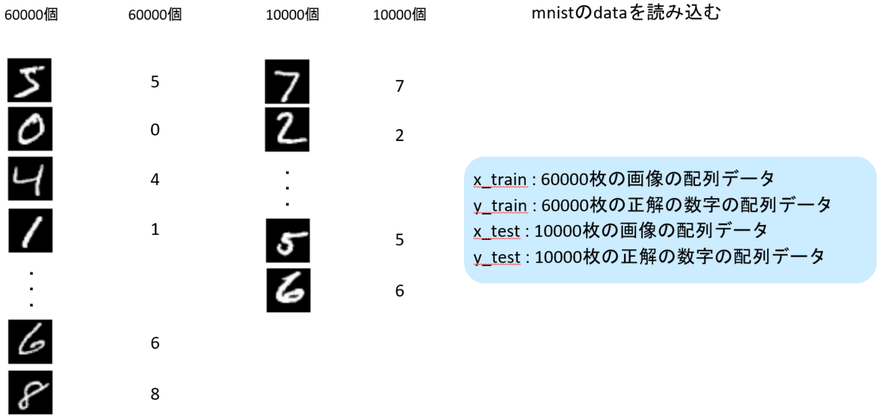
医療とAI・ビッグデータ応用
④MLP

統合教育機構
須藤毅顕

前回までの復習

データを読み込む

(x_train, y_train),(x_test, y_test) = mnist.load_data()




```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

```
print(x_train.shape)
print(x_test.shape)
```

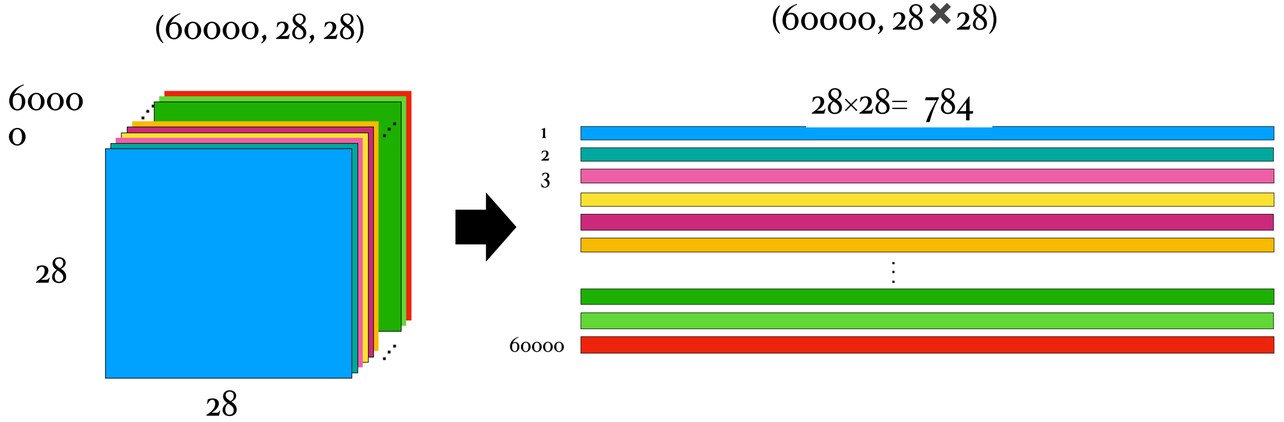
```
(60000, 784)
(10000, 784)
```

```
print(y_train.shape)
print(y_test.shape)
```

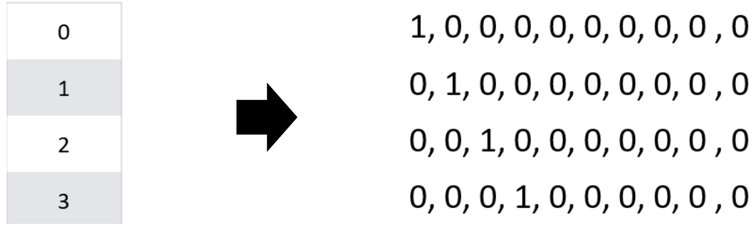
```
(60000, 10)
(10000, 10)
```

データを加工する

x_train, x_test: 画像の2次元配列を１次元にする
x_train = x_train.reshape(60000, 784)
正規化する
x_train = x_train / 255



y_train, y_test : one-hot encodingで01のみの配列にする
y_train = to_categorical(y_train)

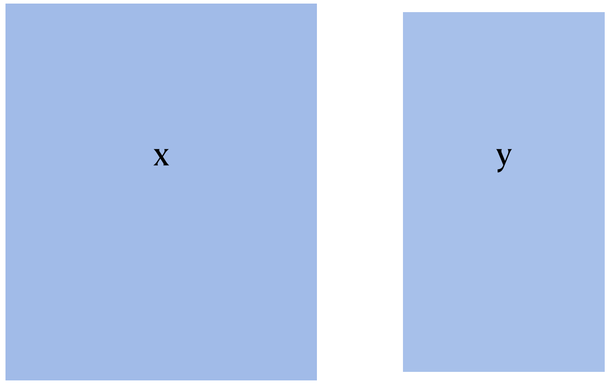


深層学習(教師あり機械学習)の復習

データを用意する

x(特徴量データ)

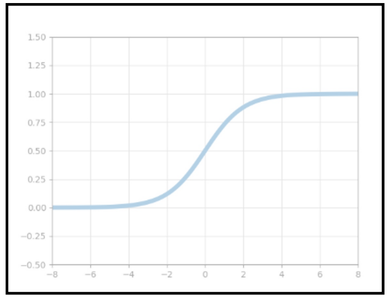
y(正解データ)



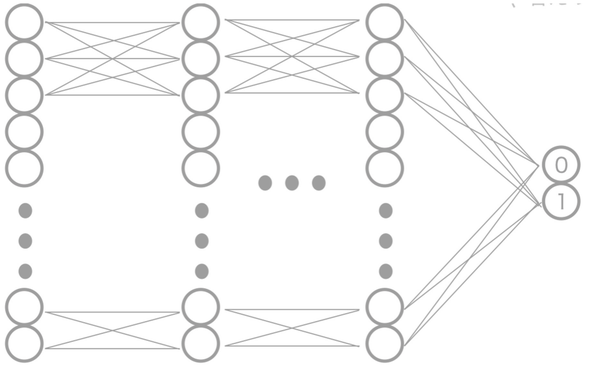
データを配列に整える

学習させる

ロジスティック回帰分析



ニューラルネットワーク



深層学習(教師あり機械学習)の復習

データを用意する

x(特徴量データ)

x

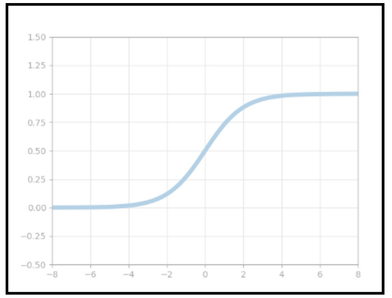
y(正解データ)

y

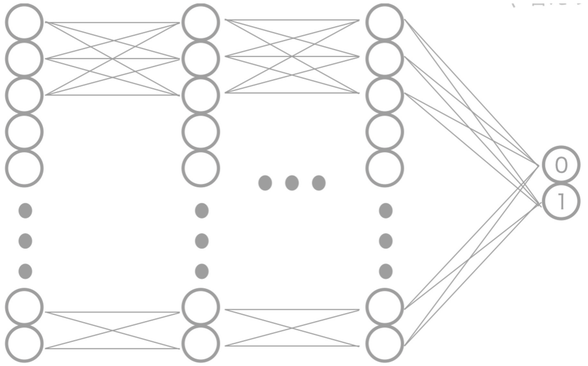
データを配列に整える

学習させる

ロジスティック回帰分析

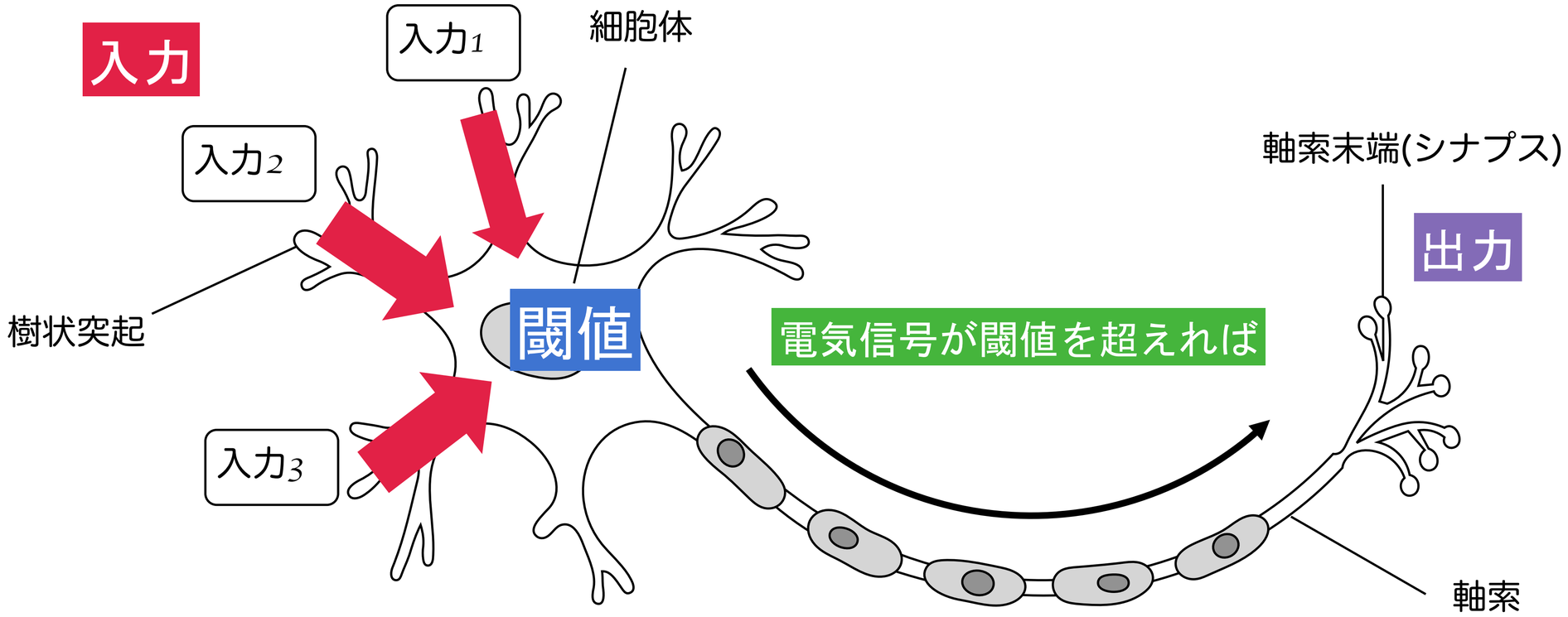


ニューラルネットワーク



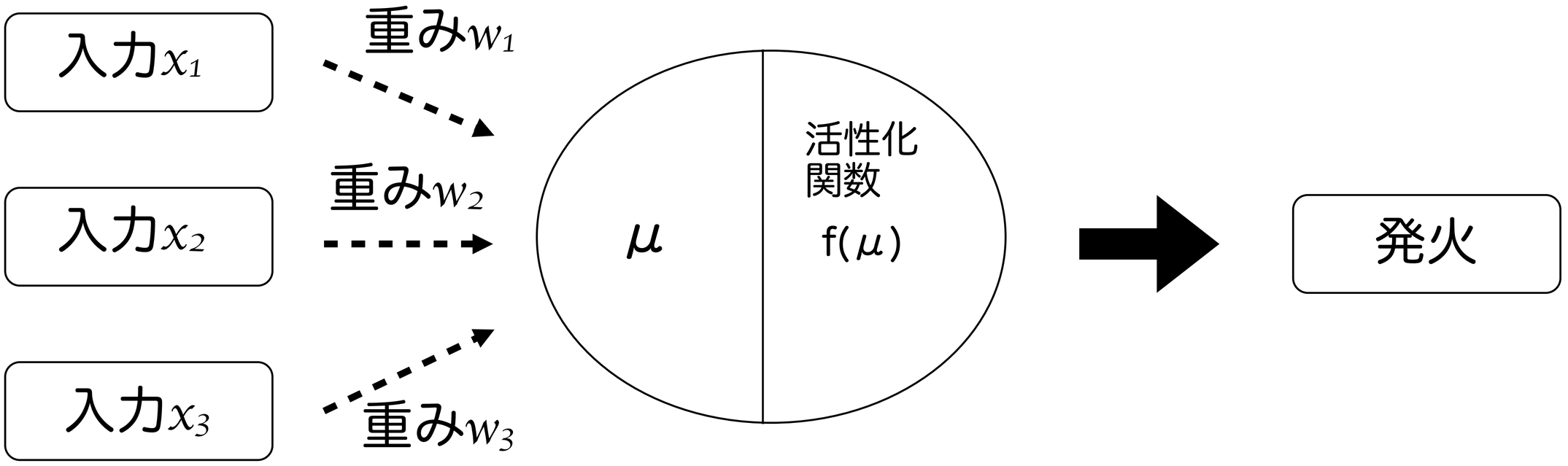
ニューロンとパーセプトロン

・ニューロンは、樹状突起、細胞体、軸索からなる
・ニューロンは、樹状突起から入力された電気信号が神経細胞内の電位を超えるかどうかの閾値を持っている
・閾値を超えるとニューロンは興奮状態となり、軸索末端から電気信号が出力される



ニューロンとパーセプトロン

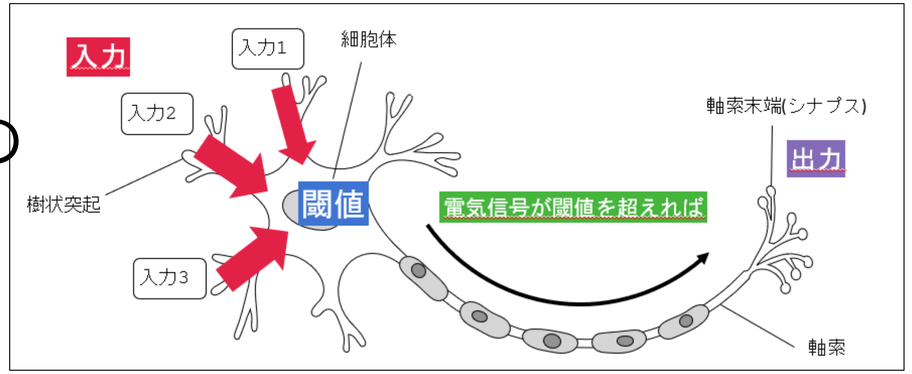
単一の人工ニューロンはこのようなモデルで表すことができる。



ニューロンとパーセプトロン

(イメージ)

X1～X3：入力. 各電気信号
w1～w3：重み. 細胞体までに受ける抵抗の様なもの
μ：各電気信号が細胞体に集まった際の総和
f(μ)：活性化関数. 閾値



入力x_1
入力x_2
入力x_3

重みw_1
重みw_2
重みw_3

μ

活性化関数
f(μ)

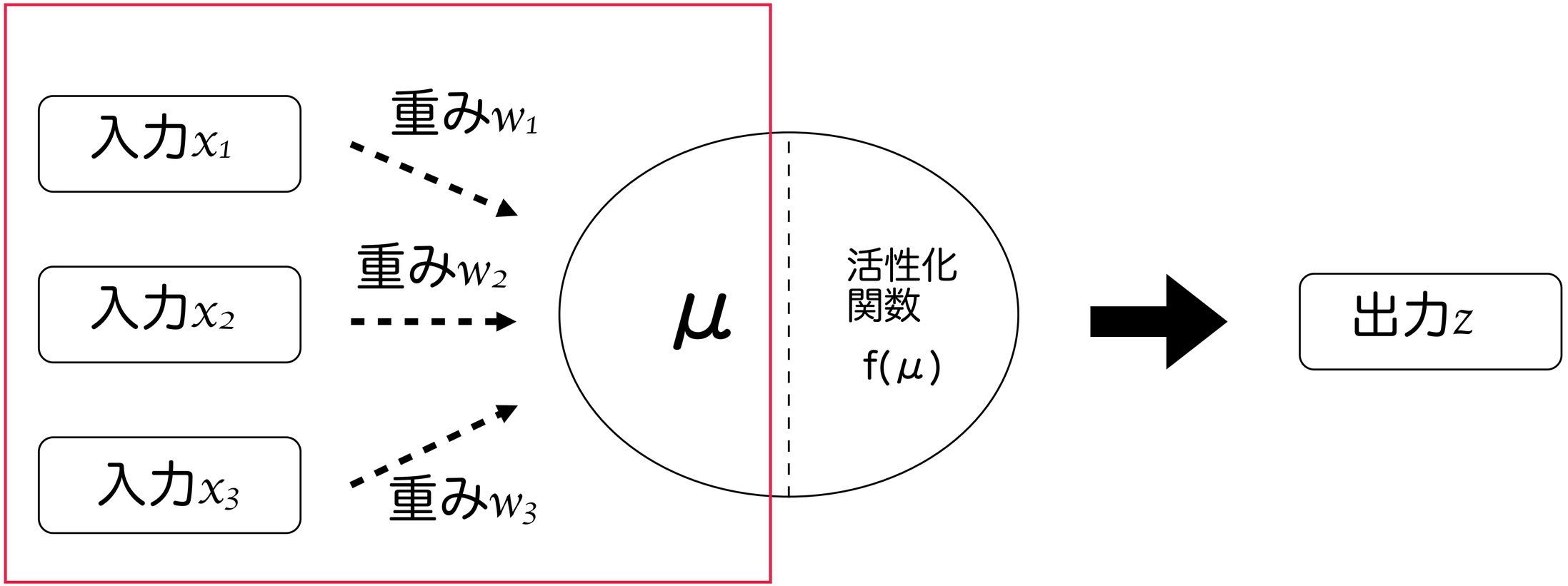
発火

樹状突起　　細胞体　　軸索

ニューロンとパーセプトロン

μは入力値に重みを掛け合わせた合計で計算される

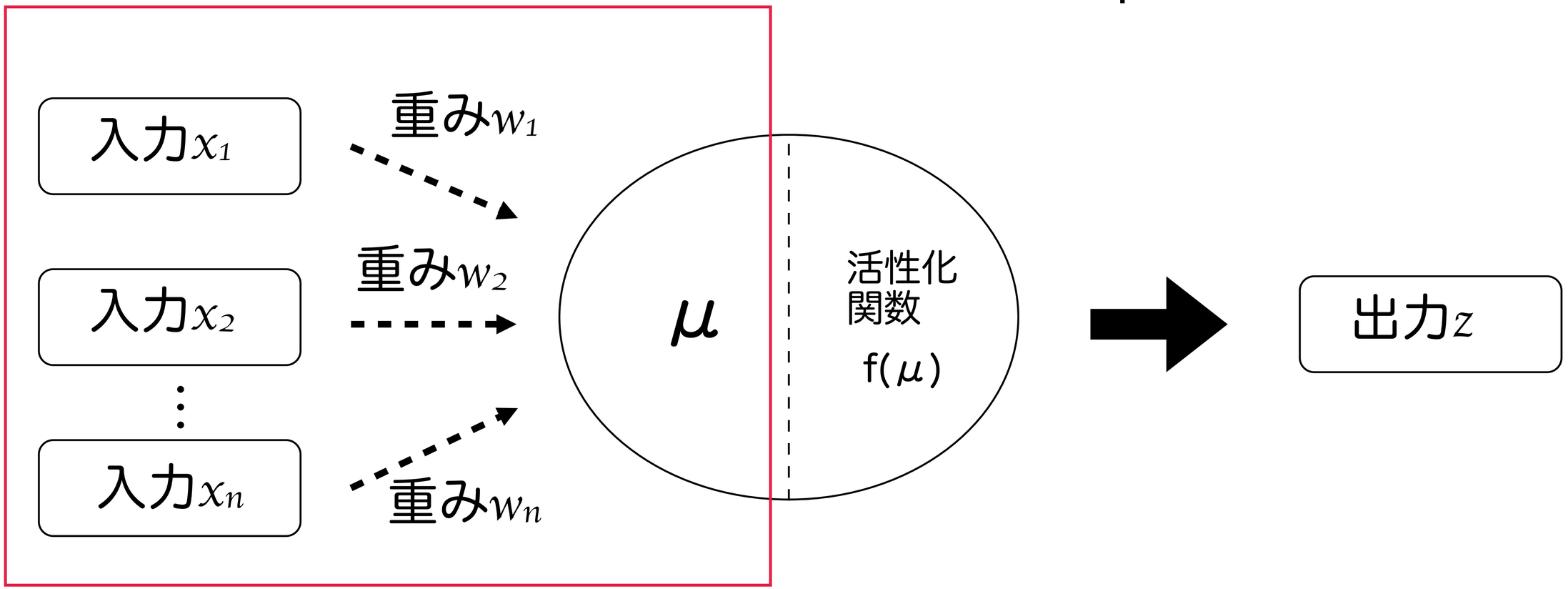
$$\mu = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$$



ニューロンとパーセプトロン

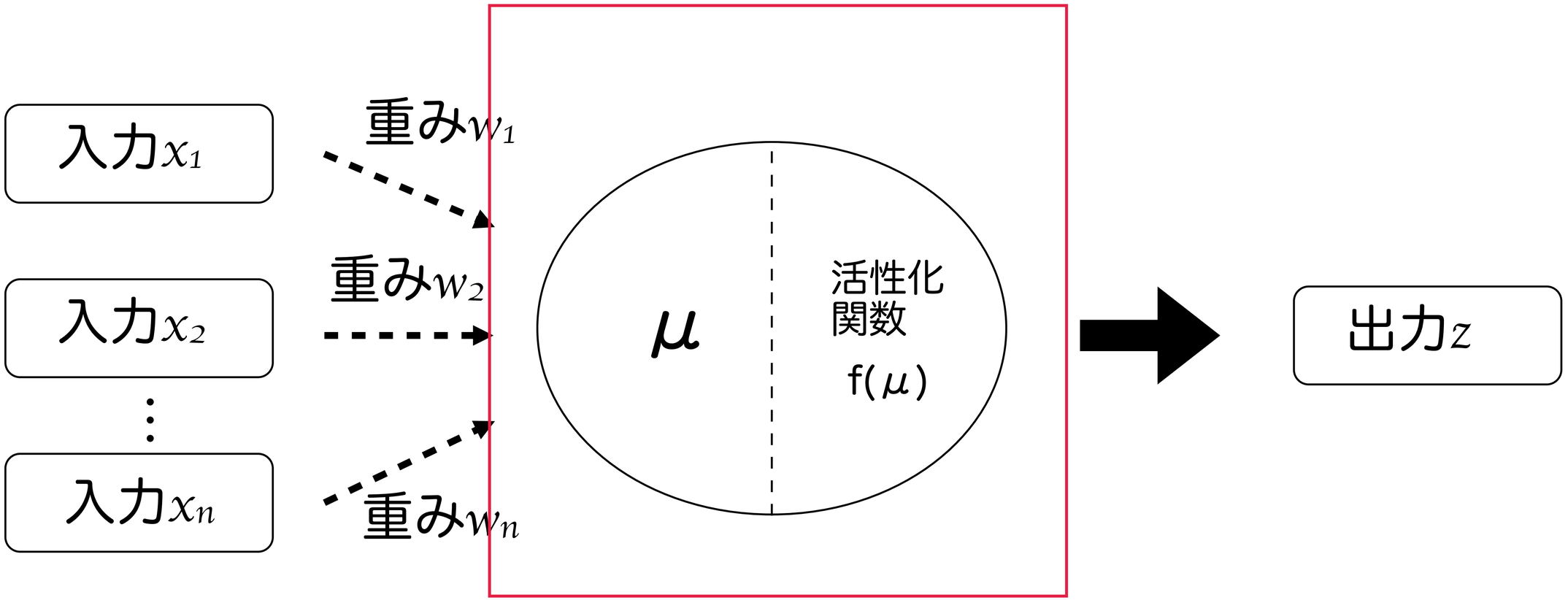
入力がn個あった場合は下のように一般化できる

$$\mu = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n = \sum_{i}^{n} x_i w_i$$



活性化関数

(人工)ニューロンが受け取った値を発火するかしないか
判断するための関数を活性化関数という



活性化関数

(人工)ニューロンが受け取った値を発火するかしないか
判断するための関数を活性化関数という

活性化関数には多くの種類がある

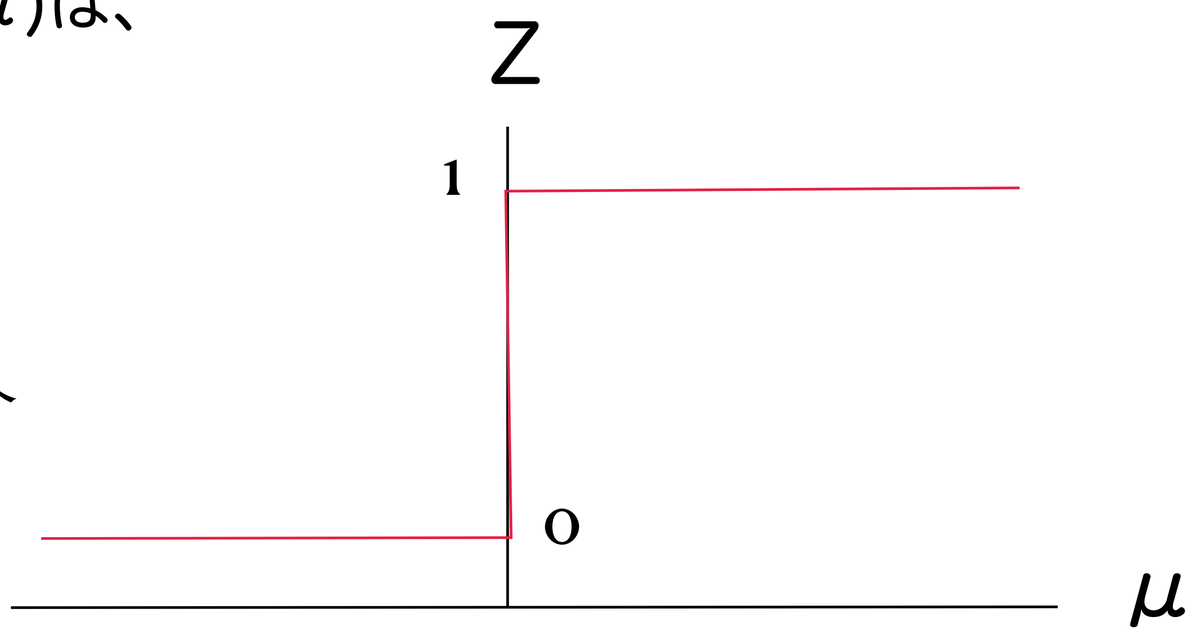
- ステップ関数
- 恒等関数
- シグモイド関数
- tanh関数
- ReLU関数
- ソフトプラス関数
- Leaky ReLU
- ソフトマックス関数
- PReLU / Parametric ReLU
- ELU
- SELU
- Swish関数
- Mish関数

など

例えば活性化関数にステップ関数を用いると

出力値をZとすると活性化関数f(μ)は、

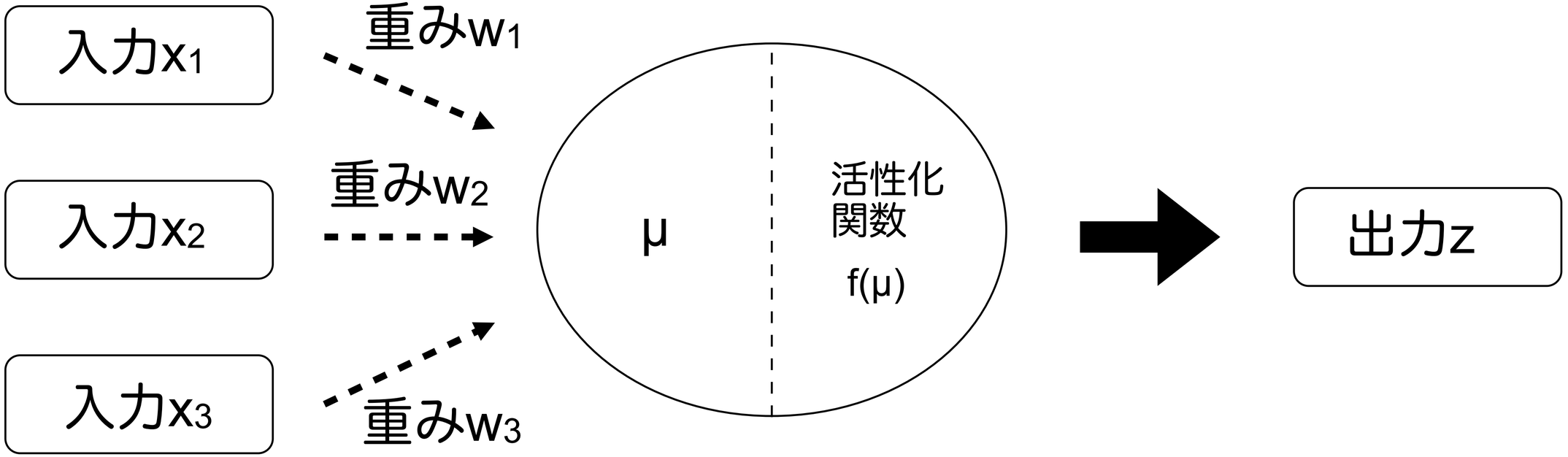
$$Z = f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



μがどんな値でも出力は
0か1のいずれかになる！

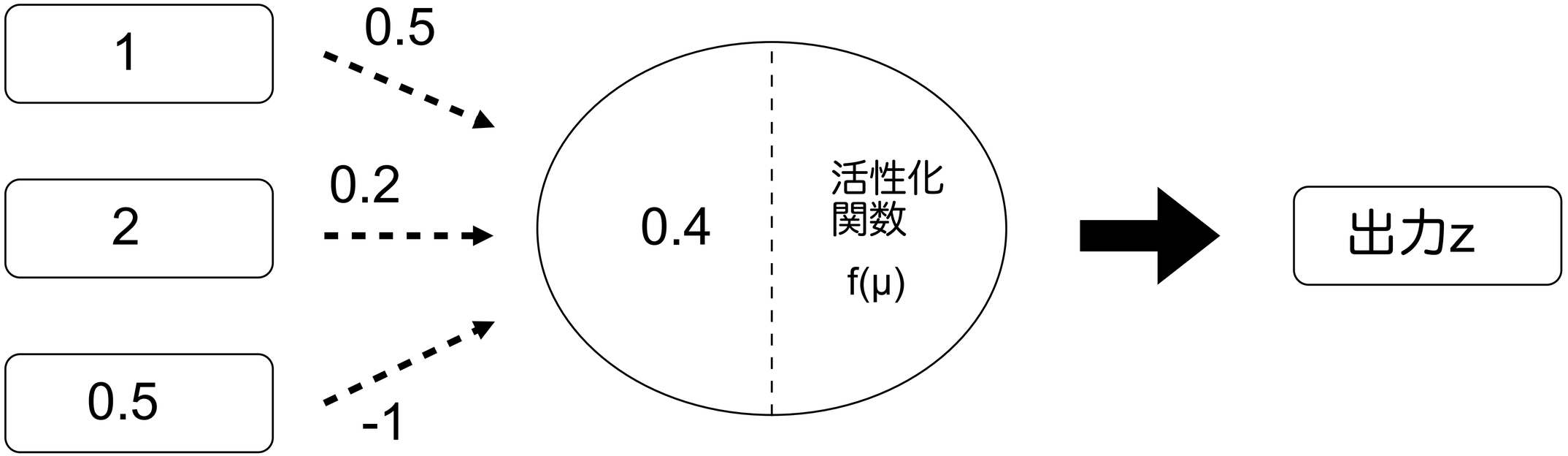
閾値を0とすると、μが0より大きければzは1となり(発火)、
0以下であれば0となる(発火しない)

例えばx_1=1、x_2=2、x_3=0.5、w_1=0.5、w_2=0.2、w_3=-1の時は？

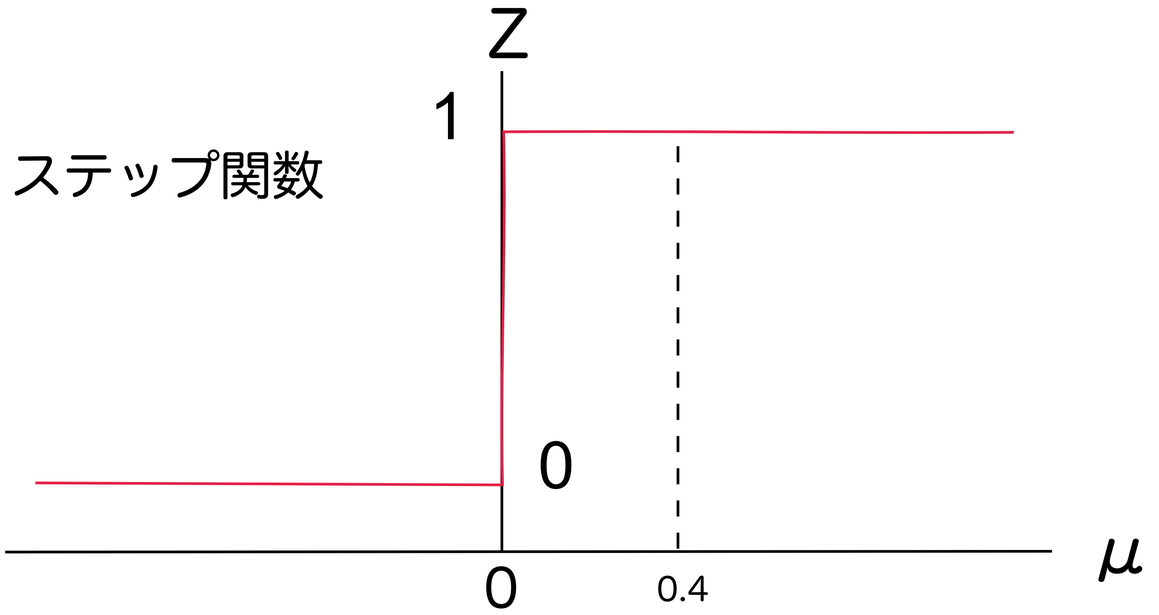


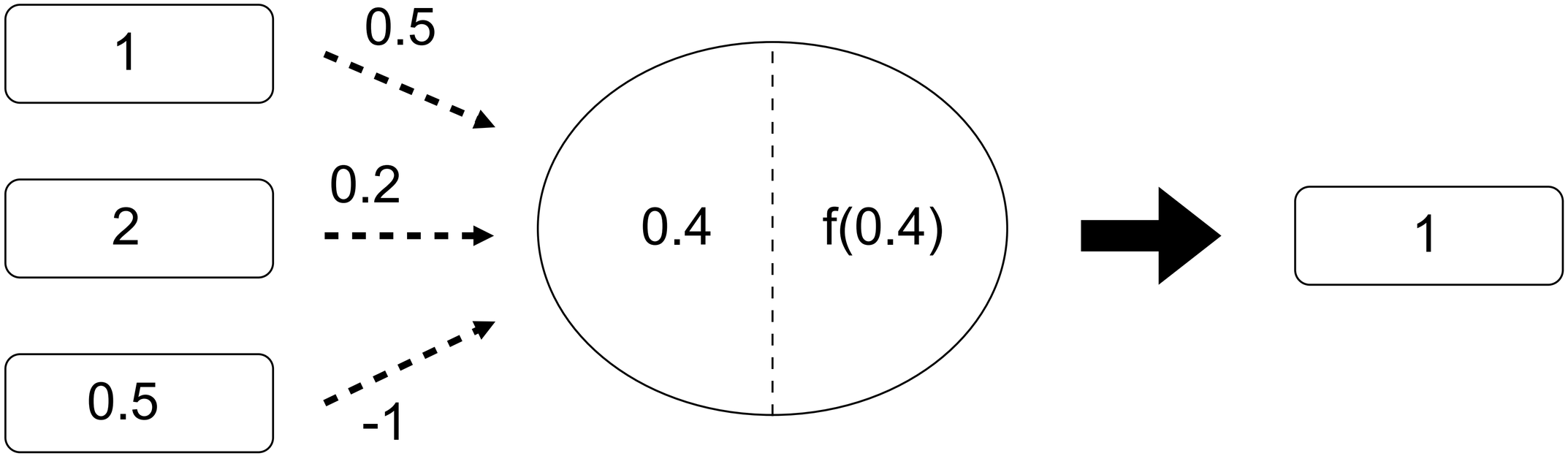
例えばx_1=1、x_2=2、x_3=0.5、w_1=0.5、w_2=0.2、w_3=-1の時は？

$$\mu = 1 \times 0.5 + 2 \times 0.2 + 0.5 \times (-1) = 0.4$$

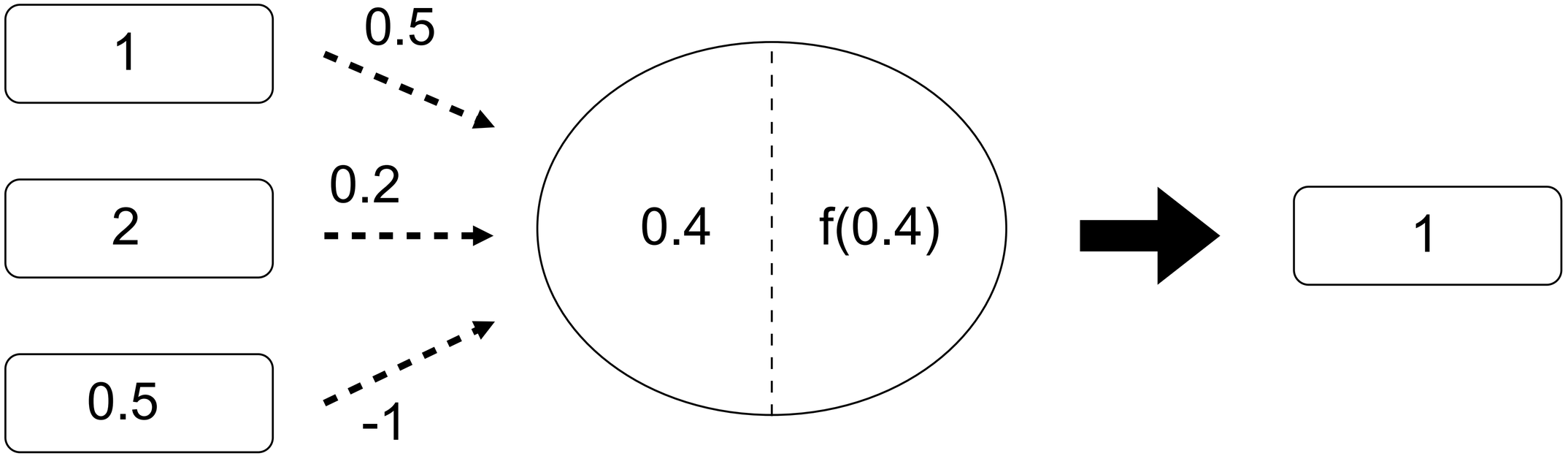


$$z = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



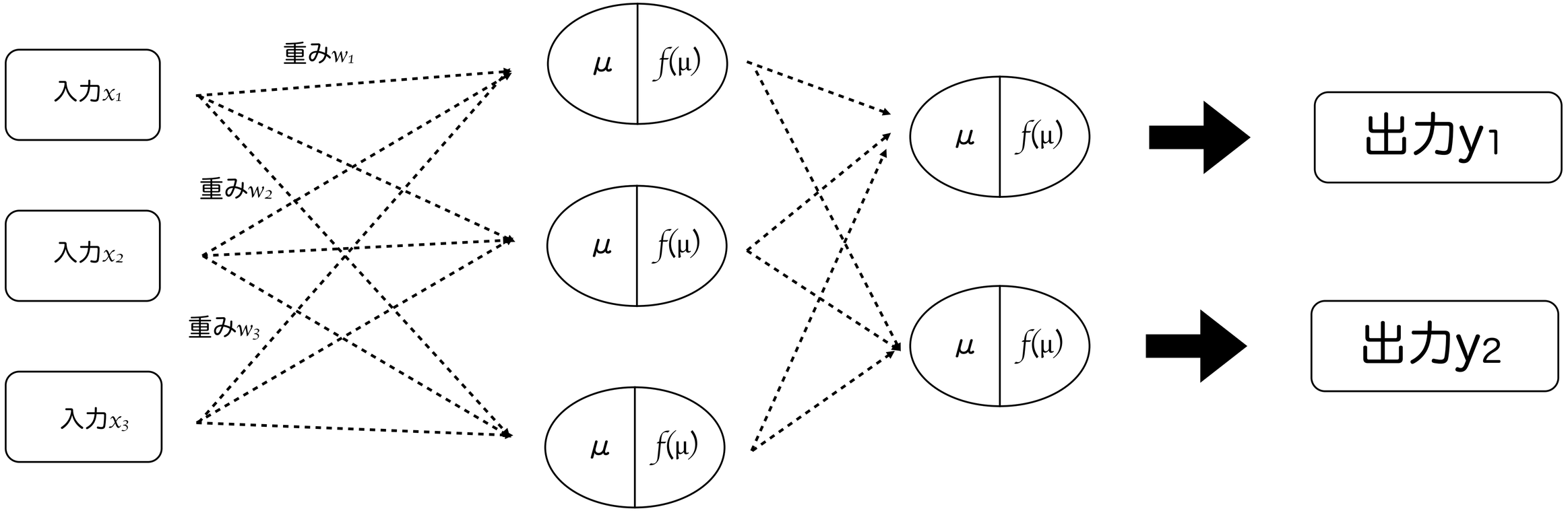


このような重みw_iや閾値を調整することできる
人工ニューラルネットワークで学習する仕組みをパーセプトロンと呼ぶ。

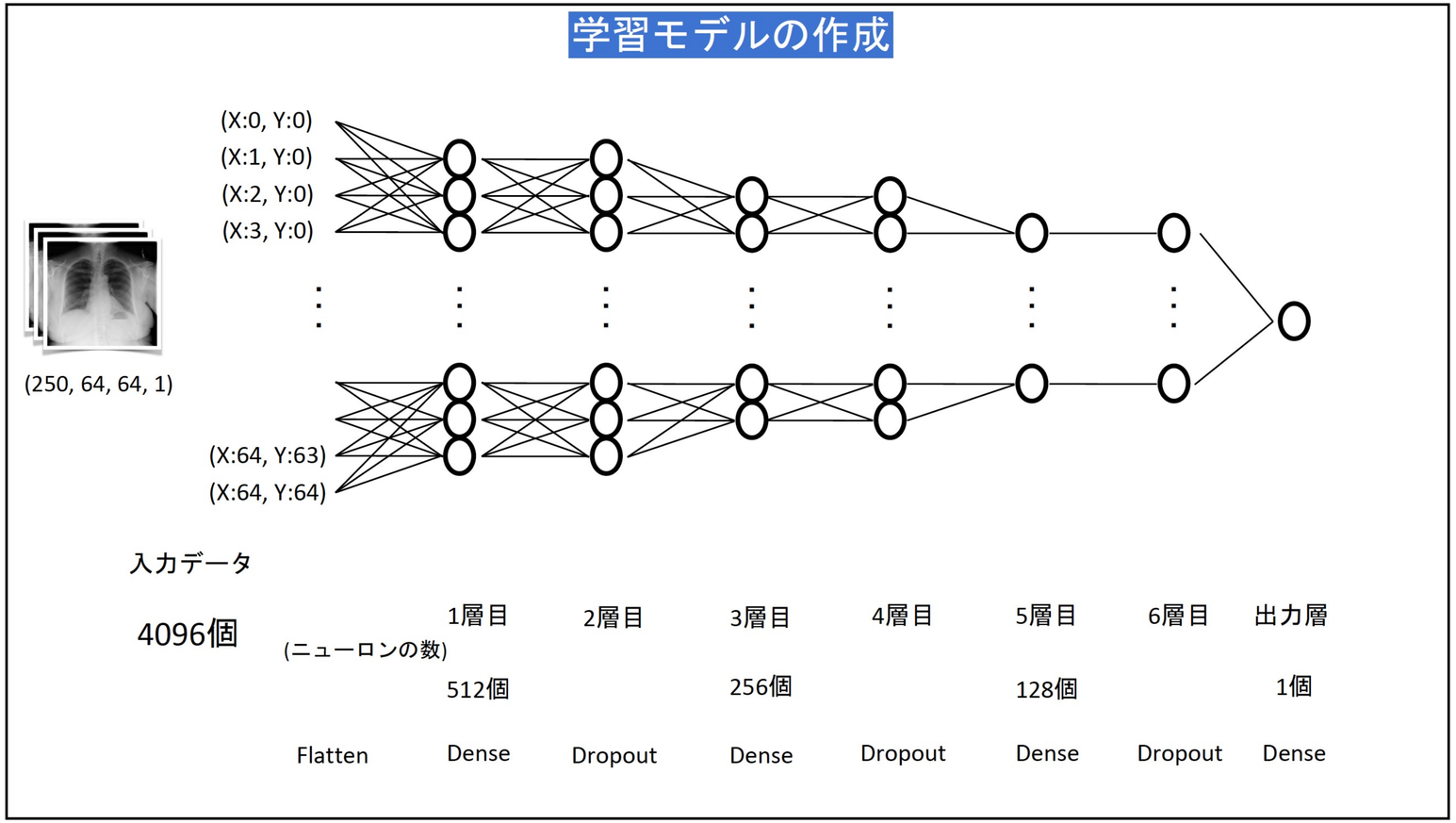


多層パーセプトロン（MLP：Multi Layer Perceptron）

複数のパーセプトロンを用いてパーセプトロンの層を作ったものを
多層パーセプトロンという
（この中間層を複数作ってどんどん層を深く出来るので深層学習という）

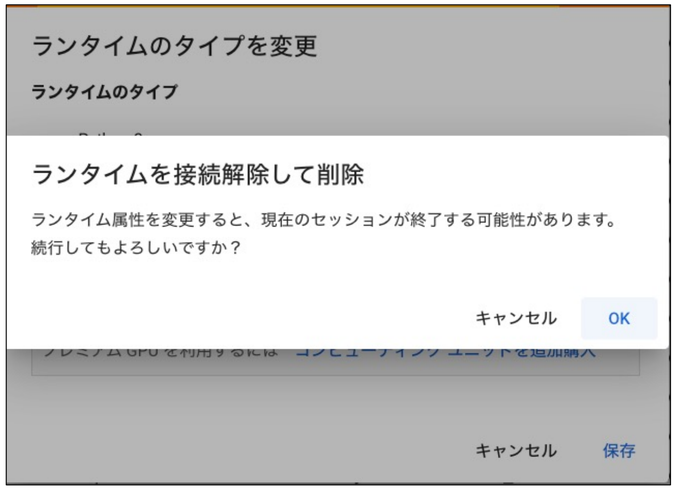


入門編で行った深層学習



これは実はMLP(Multi Layer Perceptron)

今日からGPUに切り替えて実行します
「ランタイム」→「ランタイムのタイプを変更」
ハードウェアタイプをCPUからT4 GPUに変更



(前回のファイルを開いて)
前処理までを再度実行しましょう

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(x_train.shape[0],784)/255
x_test = x_test.reshape(x_test.shape[0],784)/255

from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train,10)
y_test = to_categorical(y_test,10)
```

前回のファイルが見つからない人はwebclassの応用
4.20231102_template.ipynbをダウンロードして開いてください

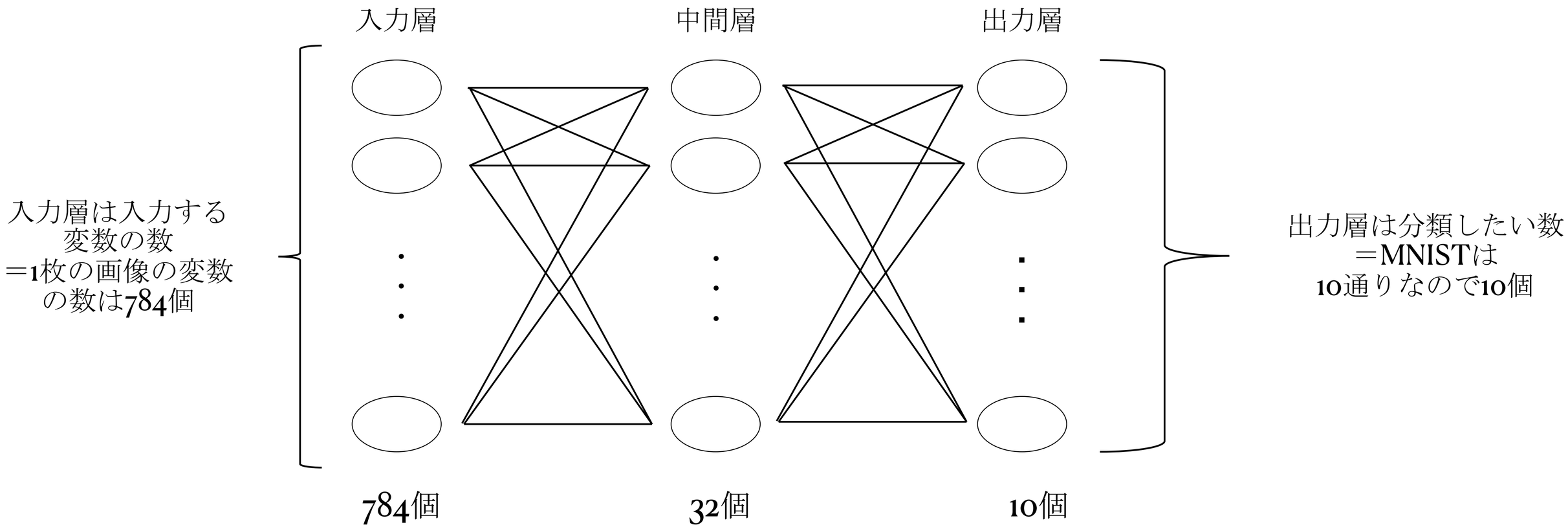
```
from keras.datasets import mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
```

60000個	60000個	10000個	10000個
	5		7
	0		2
	4	・ ・ ・	
	1		5
・ ・ ・			6
	6		
	8		

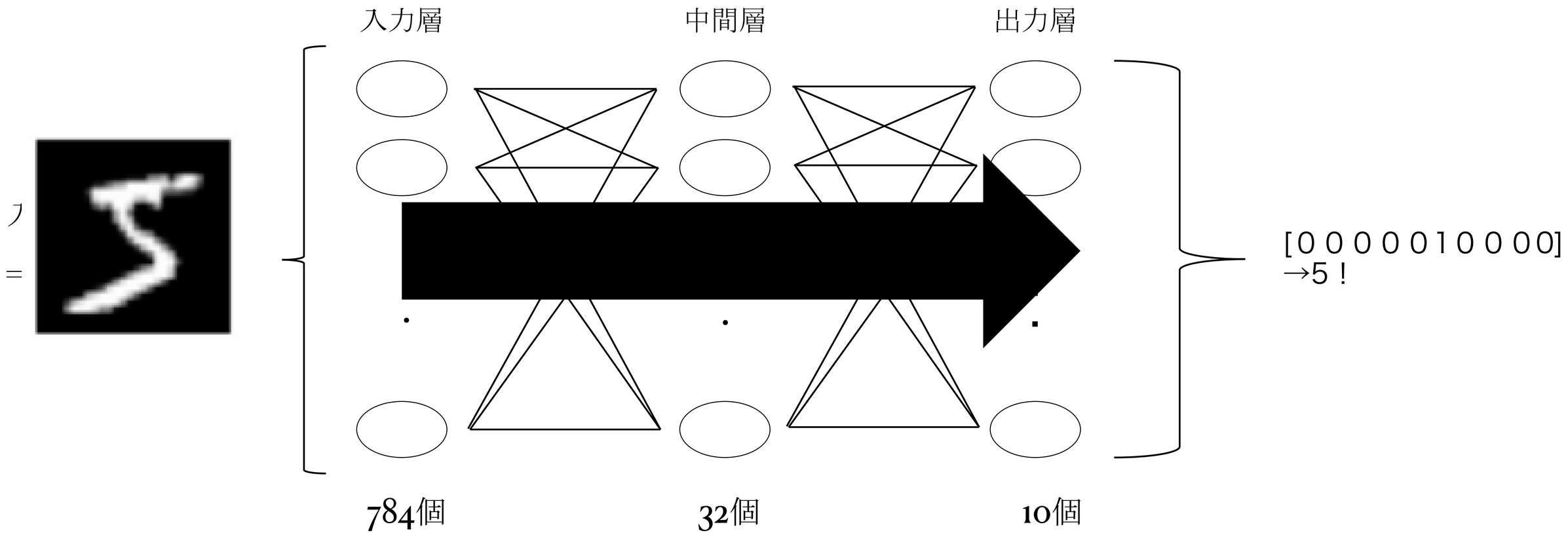
mnistのdataを読み込む

x_train : 60000枚の画像の配列データ
y_train : 60000枚の正解の数字の配列データ
x_test : 10000枚の画像の配列データ
y_test : 10000枚の正解の数字の配列データ

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
model.summary()
```



```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
model.summary()
```



```
from keras.models import Sequential
```

→tensorflowのkerasのmodelsの中のSequentialという関数を読み込む
→ここから下ではSequential()という書き方で使用できる

```
from keras.layers import Dense
```

→tensorflowのkerasのlayersの中のDenseという関数を読み込む
→ここから下ではDense()という書き方で使用できる

```
model = Sequential()
```

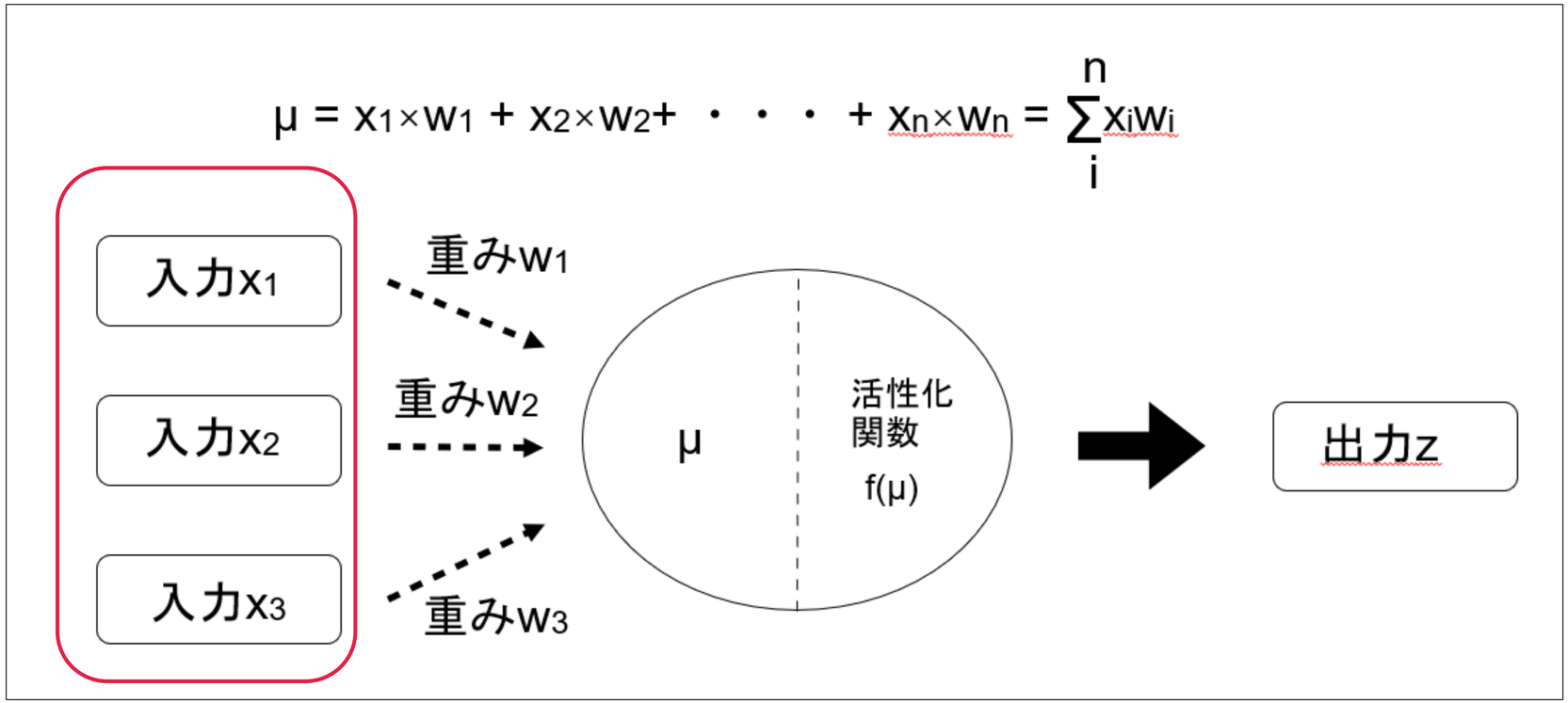
→”model”という変数名でSequential()を使用する
→ここからmodel.～～という書き方でSequential()の機能を使える

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
model.summary()
```

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個

0.53
0.24
0.88
0.34

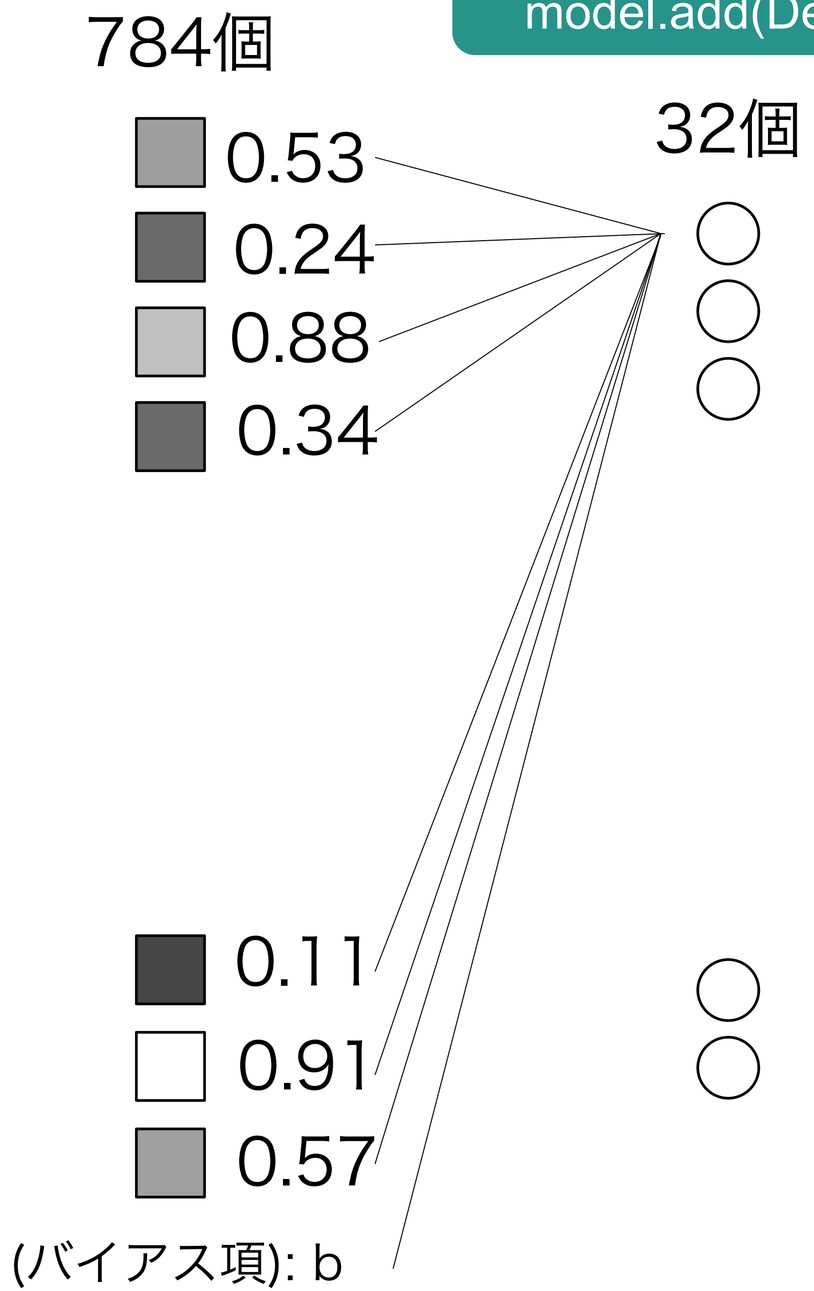


0.11
0.91
0.57

(バイアス項): b

MLPでは１枚ずつモデルに入力する
１枚は784個の数値（０～１）で表されている
＝上の図の入力値がX_1～X_{784}の784個ある

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

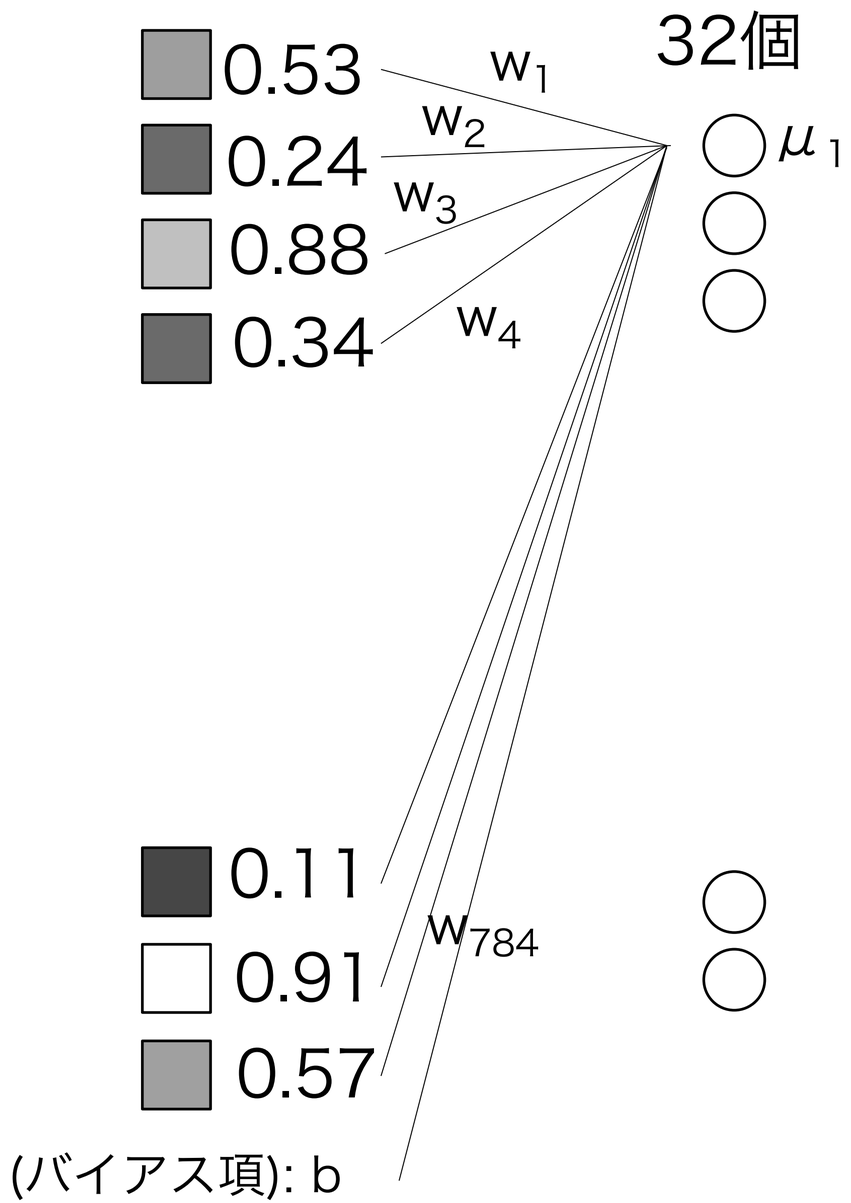


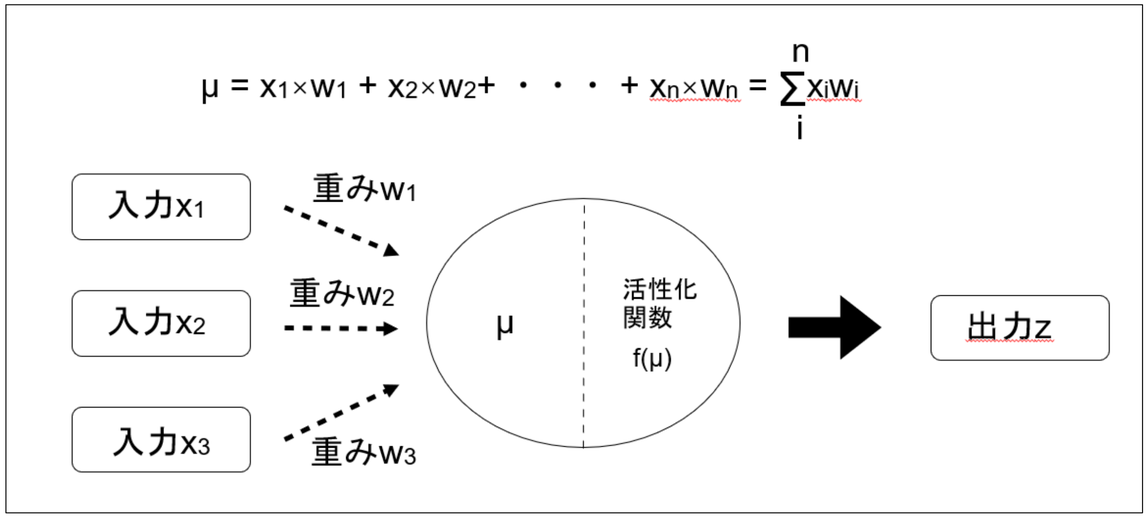
model.add()で層を追加する

Dense()で次の層のニューロンと全てつなげる(全結合)
(units=)32：次の層のニューロンの数が32個(unitsは省略可)
input_shape=(784,)：入力する変数の数
（自動的にバイアス項という定数も1つ追加される）
activation='relu'：活性化関数はReLU関数

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個





$$\mu_1 = 0.53 \times w_1 + 0.24 \times w_2 + 0.88 \times w_3 + \ldots + 0.57 \times w_{784} + b$$

この時の重みw1, w2, ..., w784とバイアス項b
はランダムに与えられる

（コンピュータがテキトーに値を決める）

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

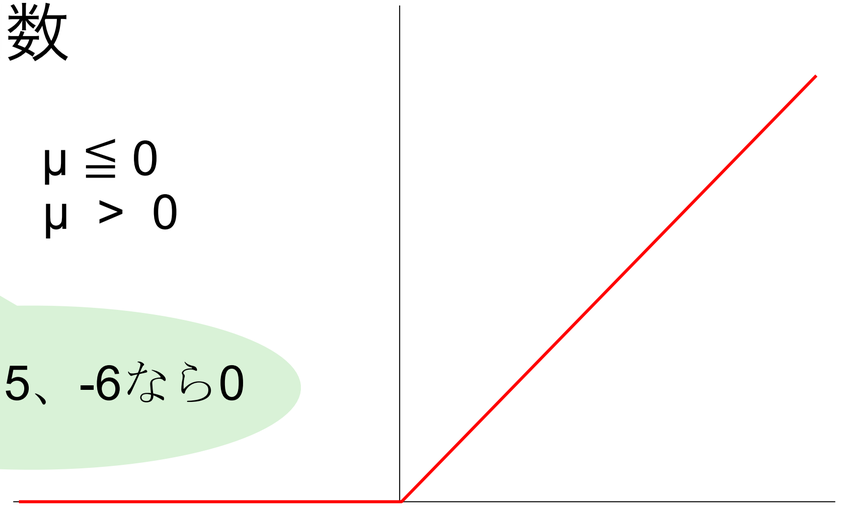
32個

(バイアス項): b

ReLU関数

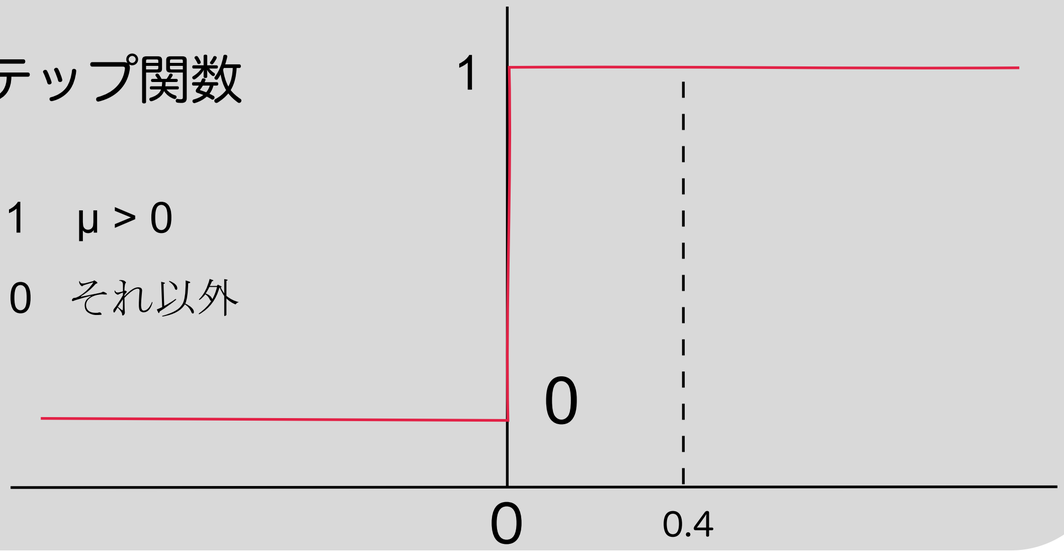
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0



ステップ関数

$$f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

32個

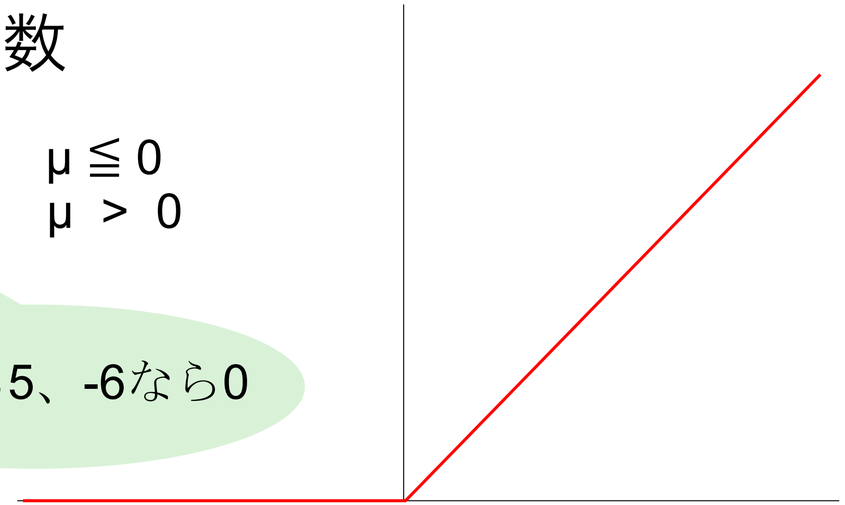
3.23

(バイアス項): b

ReLU関数

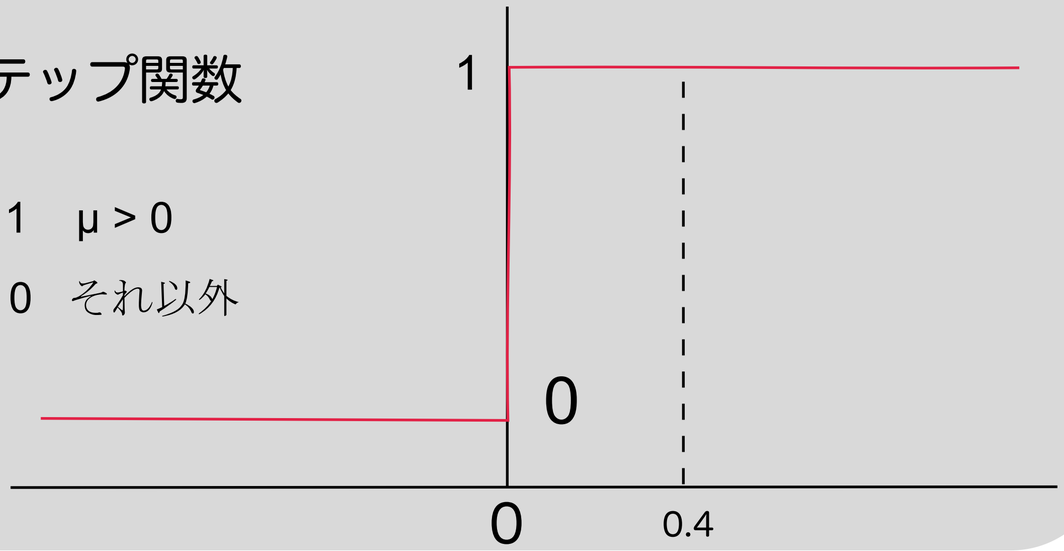
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0



ステップ関数

$$f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

(バイアス項): b

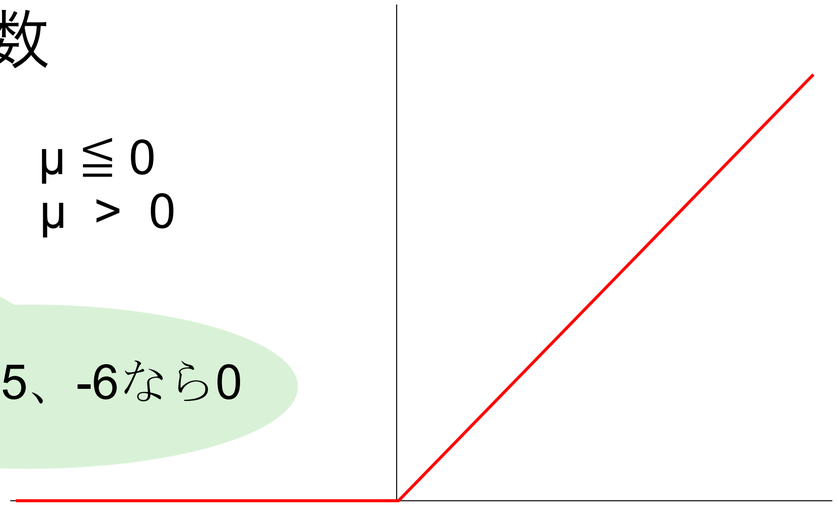
32個

3.23
0
8.45

ReLU関数

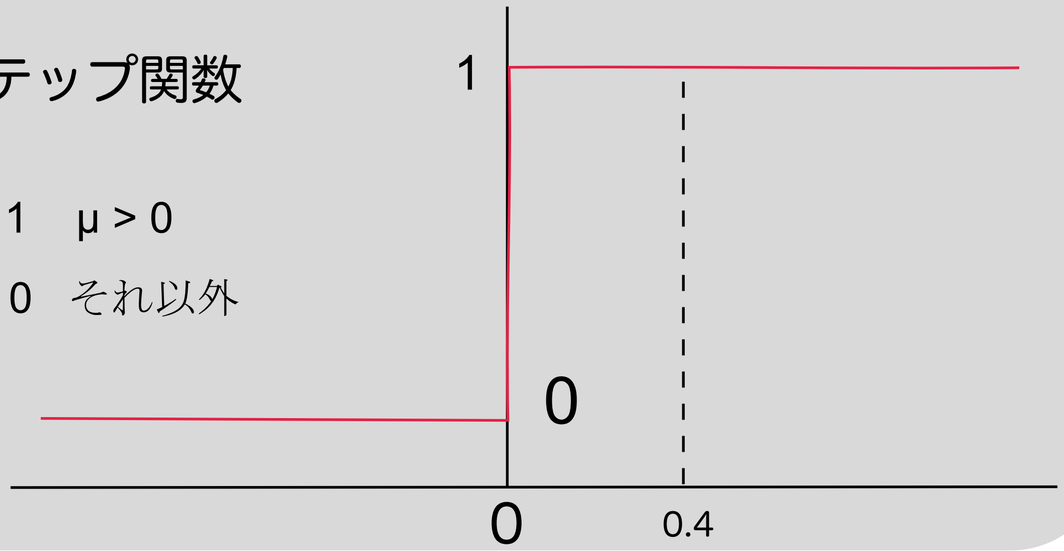
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0



ステップ関数

$$f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

784個

0.53
0.24
0.88
0.34

0.11
0.91
0.57

(バイアス項): b

32個

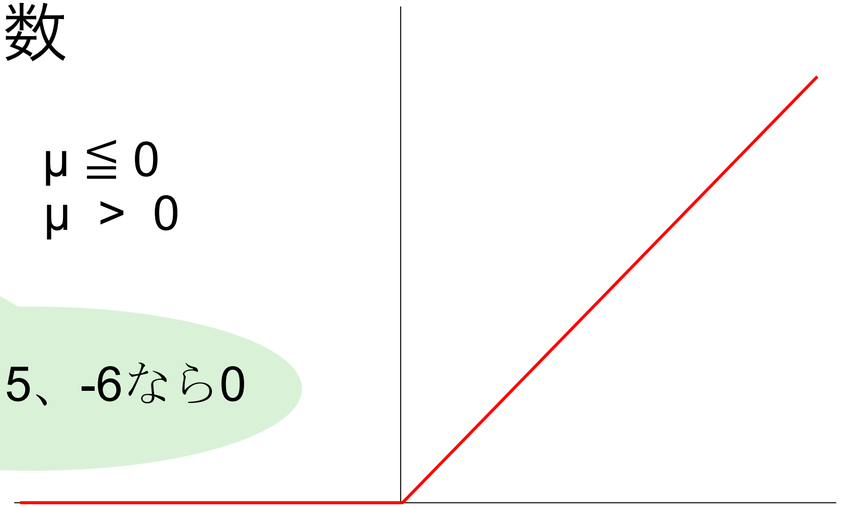
3.23
0
8.45

0
4.33

ReLU関数

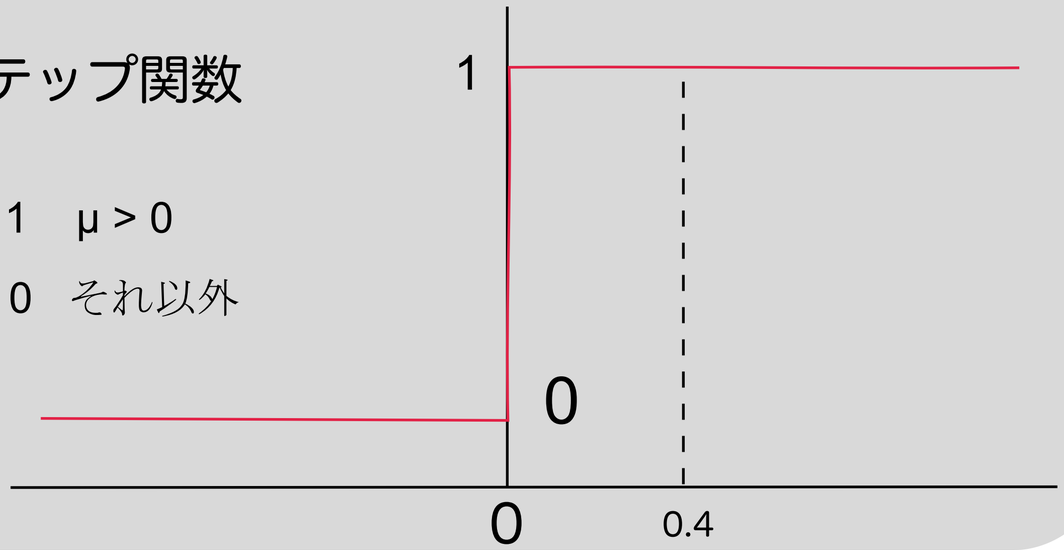
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0

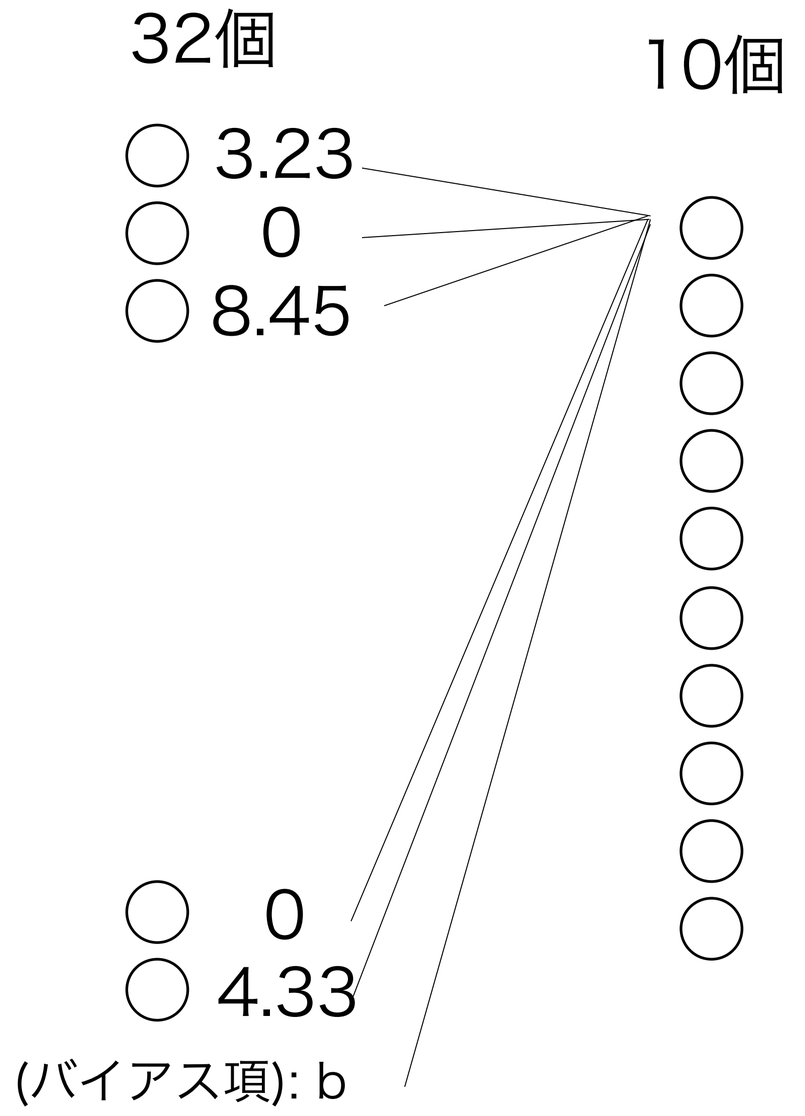


ステップ関数

$$f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```



次の層を作りたいのでmodel.add()
最初の層以外はinput_shapeは要らない
(数が決まっているため)

活性化関数：softmax関数

多値分類の出力層で用いられる活性化関数

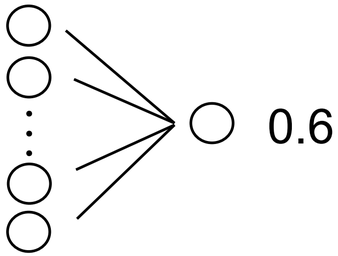
```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

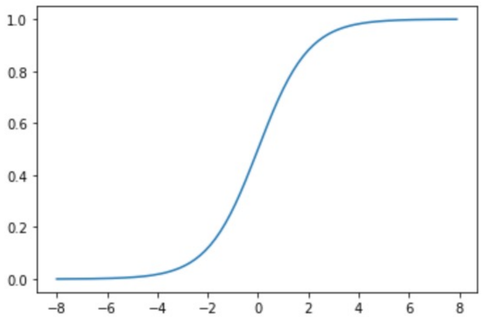
2値分類

ねこかいぬか
〇か×か
病気か否か
0か1か

出力層でよく使われる活性化関数

sigmoid関数
(activation='sigmoid')





最後が1つのニューロンで0から1の値(確率)を出力

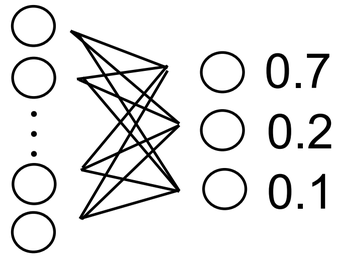
ex) 猫である確率が0.6 (=犬である確率は0.4)

多値分類

晴れか雨か曇りか
数学か国語か英語か物理か
1か2か3か4か5か

出力層でよく使われる活性化関数

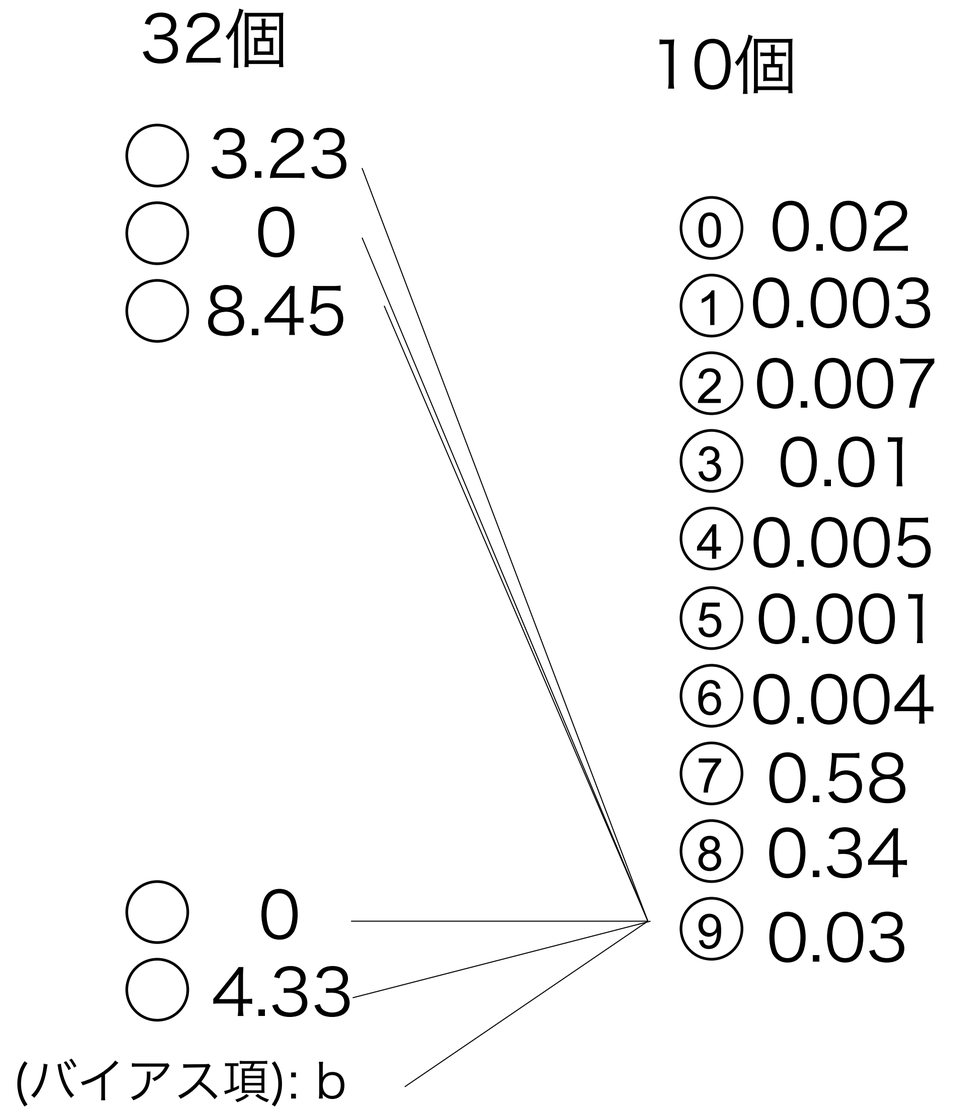
softmax関数
(activation='softmax')



最後が分類したい数のニューロンで全てを
足すと１になるように値(確率)を出力

ex) 晴れである確率が0.7、雨が0.2、曇りが0.1

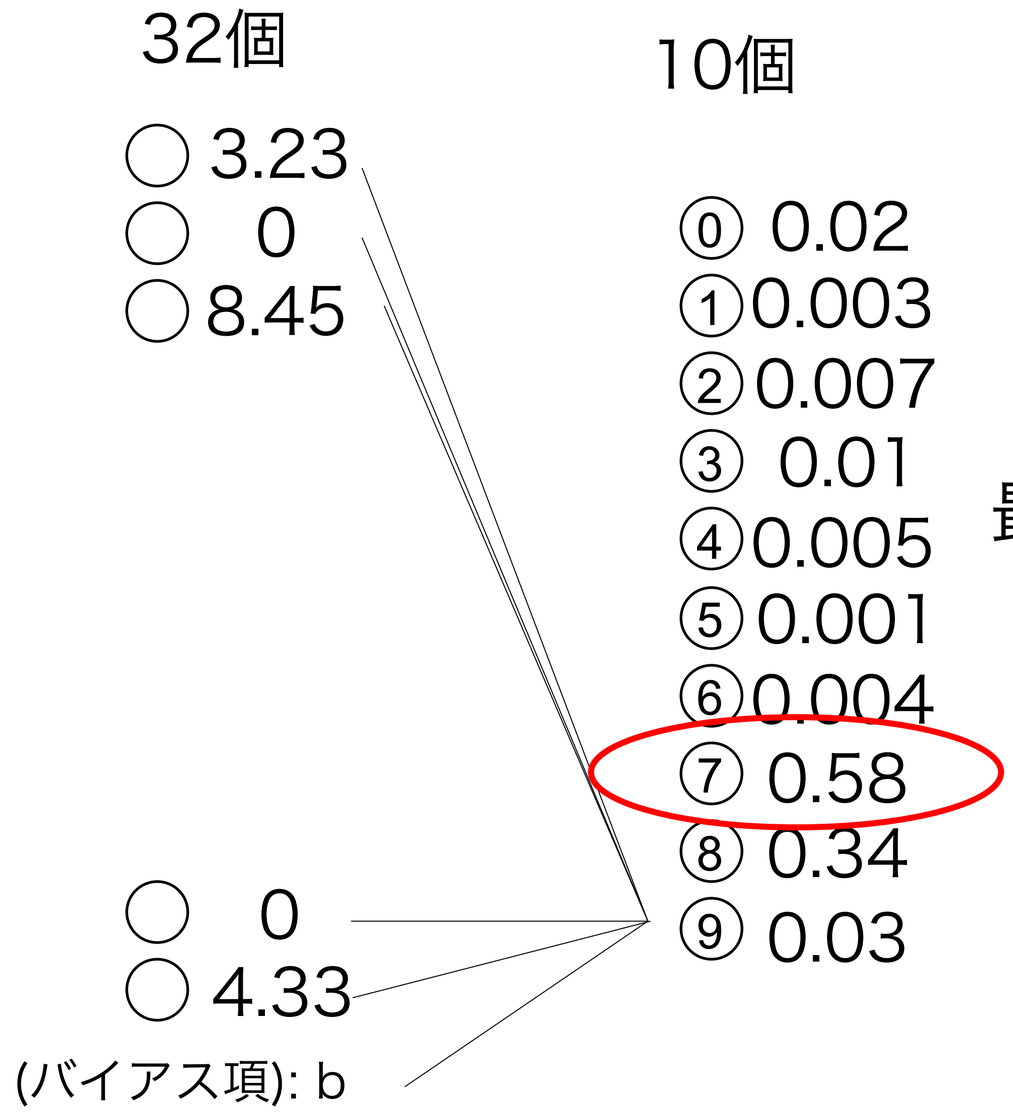
```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```



mnistは10クラスあるので
出力するニューロンの数は10個

最大の確率が予測される値

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```



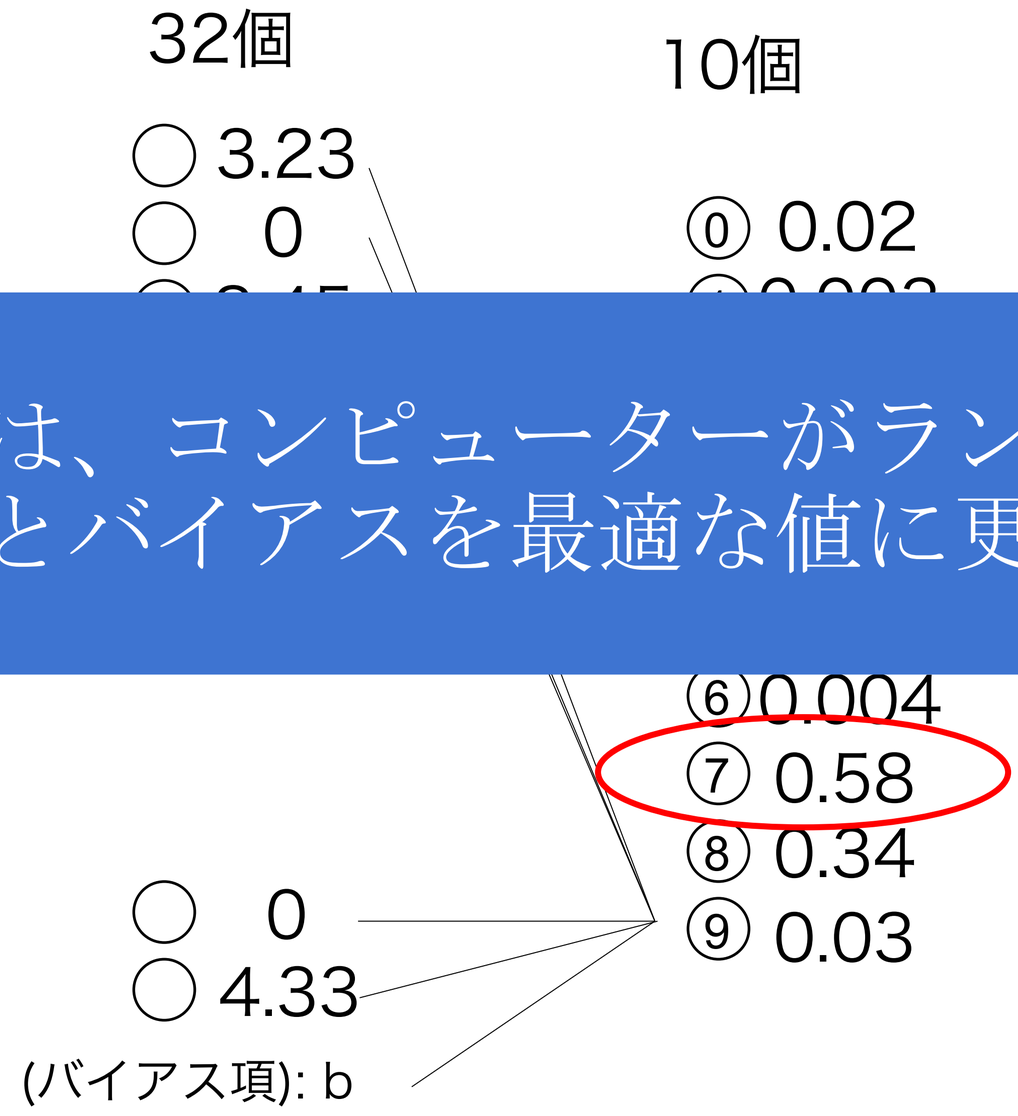
最大の確率が予測される値

例えばこの場合は7と予測

最初は全ての重みwはランダム
なので全然正解にならない

学習させる！

```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```



学習とは、コンピューターがランダムに振った
各重みとバイアスを最適な値に更新していくこと

る値

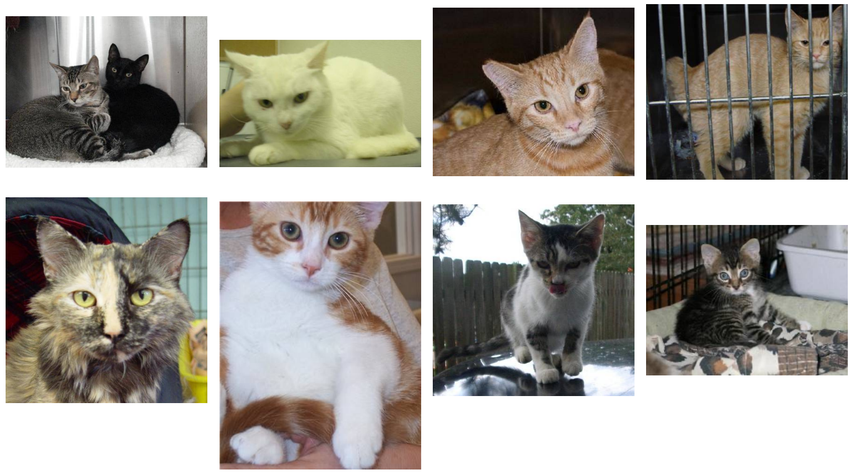
例えばこの場合は7と予測

最初は全ての重みwはランダム
なので全然正解にならない

学習させる！

学習の仕組みの概要

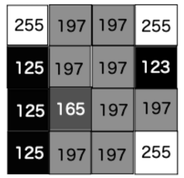
仮に猫と犬の画像の2値分類で考えると、

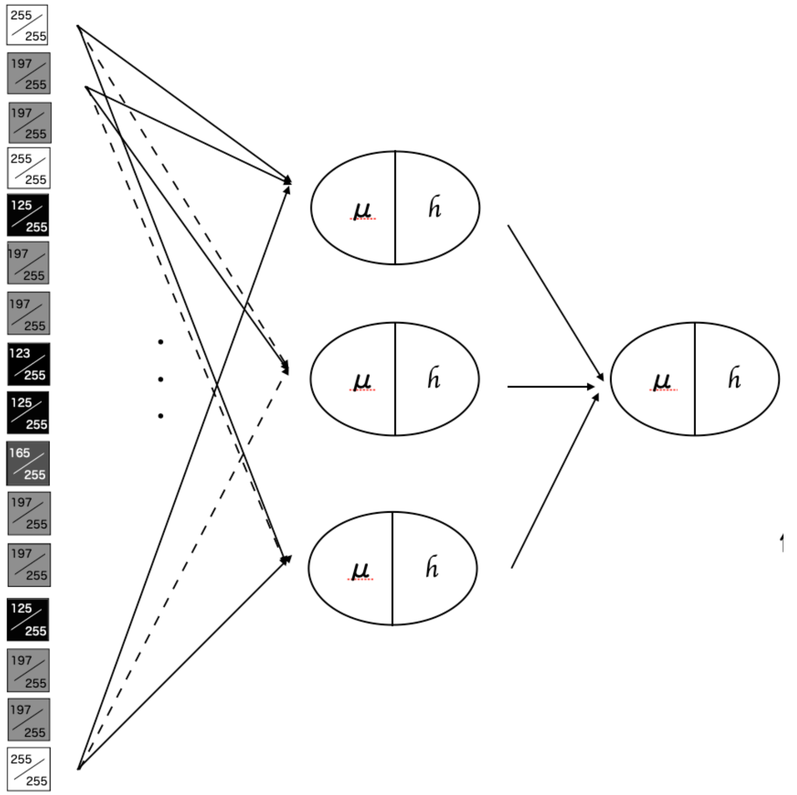


これらの画像の正解は猫(=1とする)

これらの画像の正解は犬(=0とする)

一部を取り出して予測する(ここでは仮に8枚)



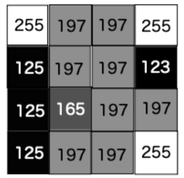


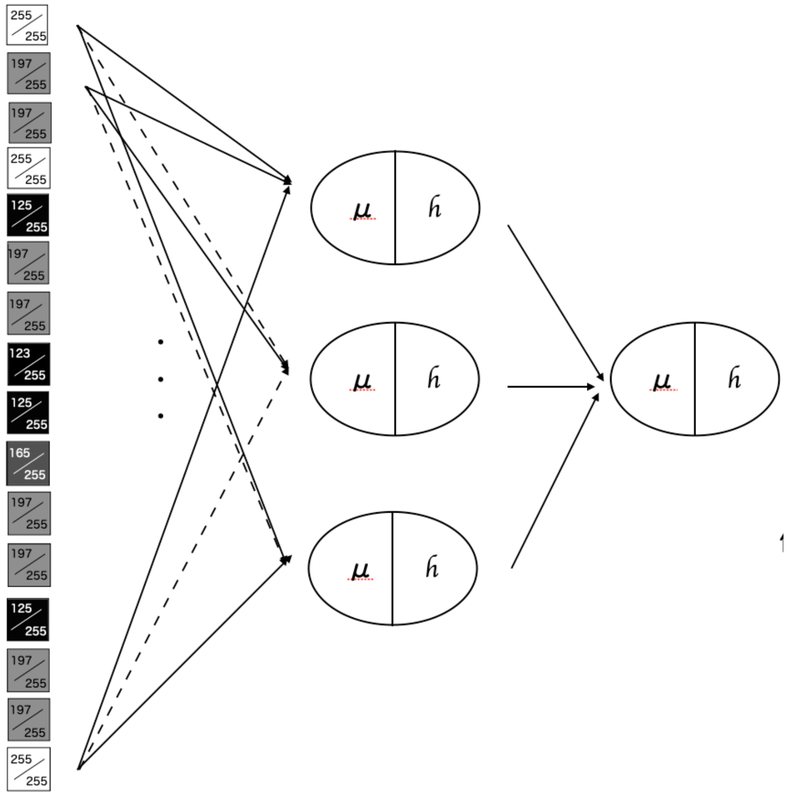
猫が１，犬が０

No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

最初はテキトーな重みwとバイアスbなので正解にならない
（＝確率が低い）

一部を取り出して予測する(ここでは仮に8枚)





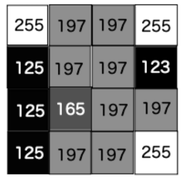
猫が１，犬が０

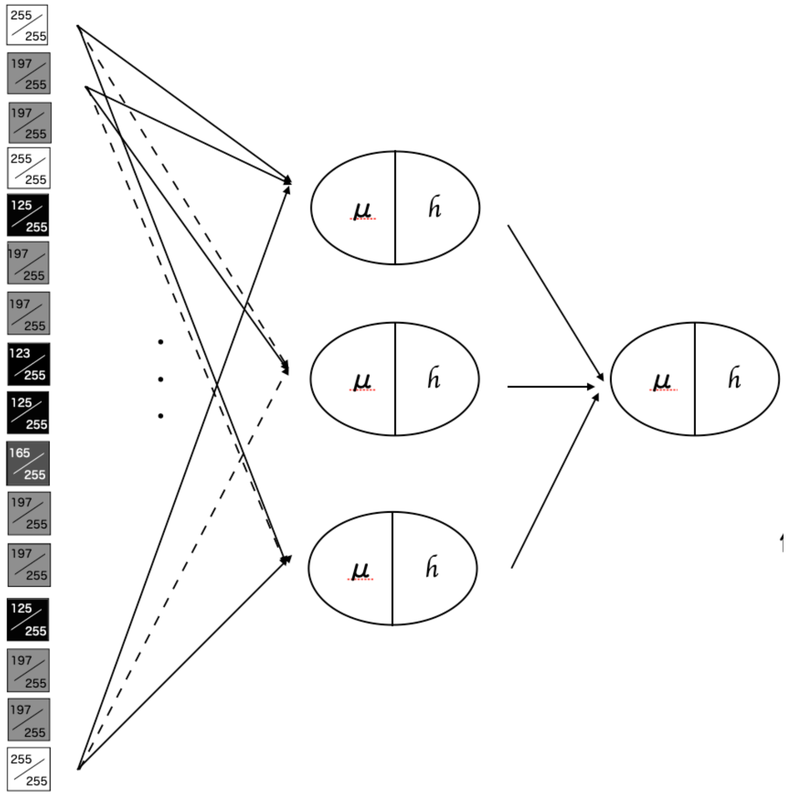
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

この出力と正解のズレ(誤差)を数値化したい

最初はテキトーな重みwとバイアスbなので正解にならない
（＝確率が低い）

誤差の計算方法(損失関数)にはいろんな方法がある





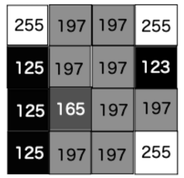
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

2値分類の時はbinary crossentropy

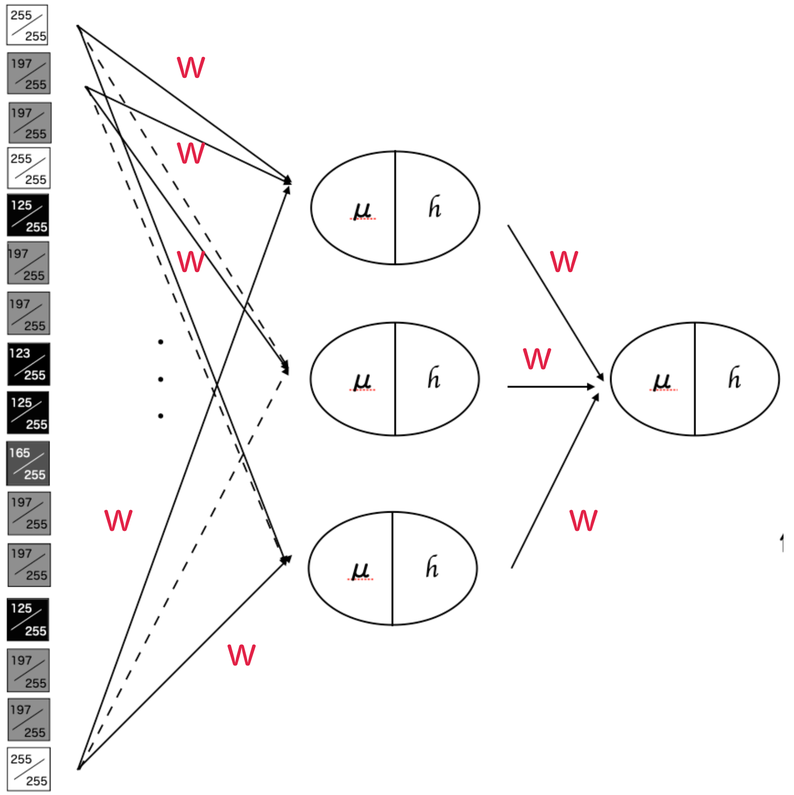
多値分類の時はcategorical crossentropy　　がよく使われる

誤差E = 0.8
(だったとします)

損失関数は重みとバイアスの式で表せる　E(w,b) = 0.8



最適化関数で誤差(損失関数)を小さくなるように重みを更新する

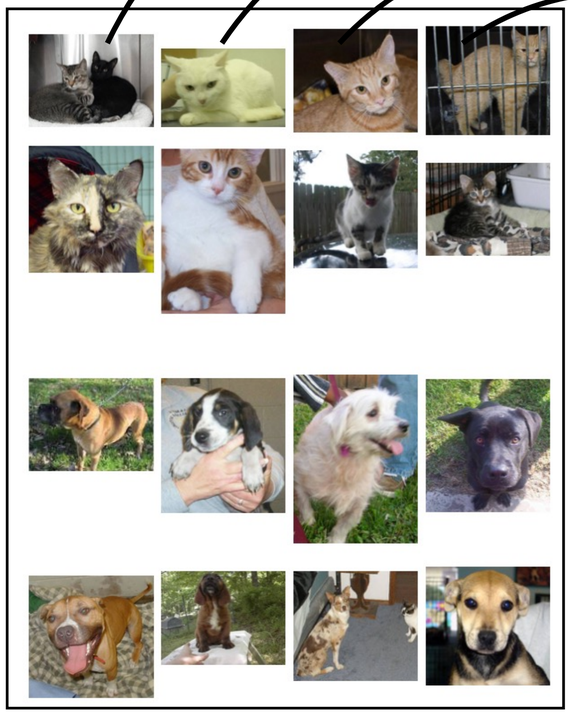


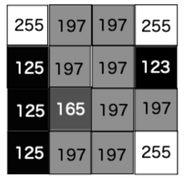
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

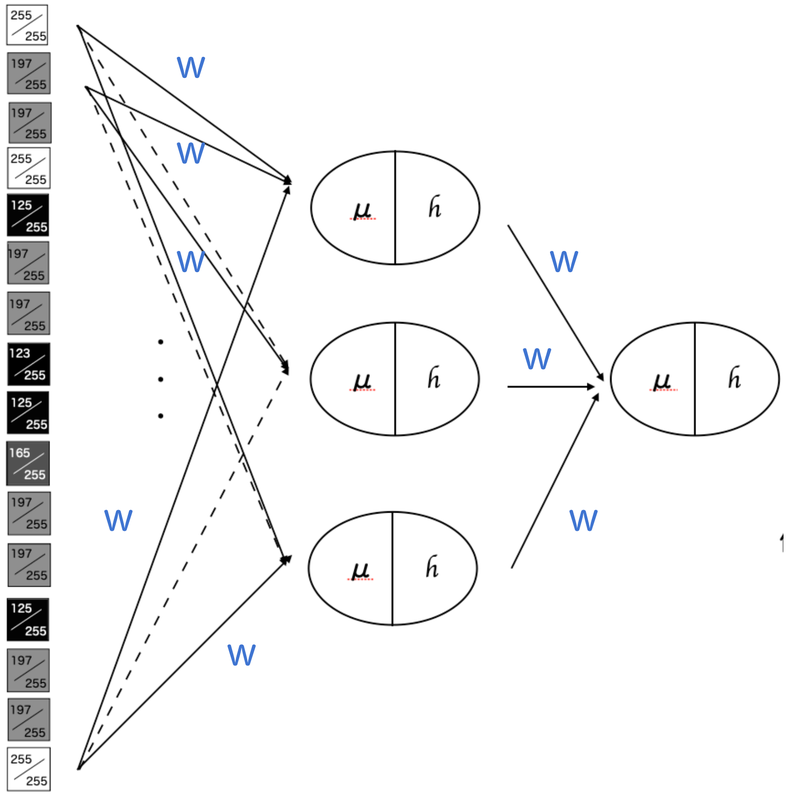
誤差E = 0.8

損失関数は重みとバイアスの式で表せる　E(w,b) = 0.8

最適化関数で誤差(損失関数)を小さくなる
ように重みを更新する







No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

誤差E = 0.8

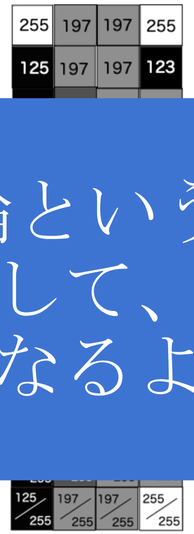
最適化関数
Adam

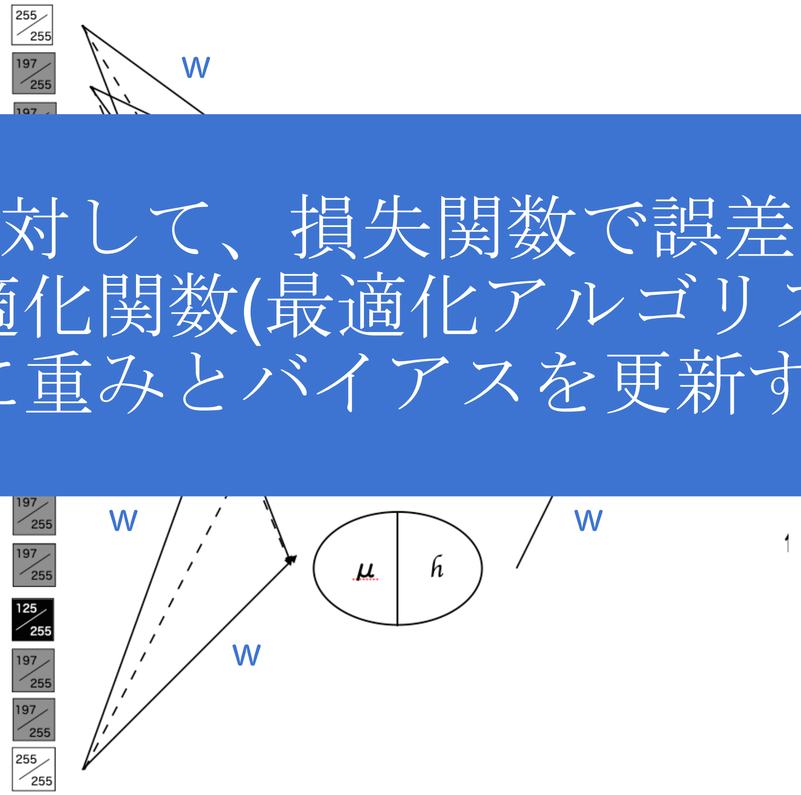
重みとバイアスを更新
各ニューロンのwとbが変わる

次の画像セットで
再度学習

損失関数は重みとバイアスの式で表せる　E(w,b) = 0.8

最適化関数で誤差(損失関数)を小さくなる
ように重みを更新する





- 予測結果(推論という)に対して、損失関数で誤差を算出する
- 損失関数に対して、最適化関数(最適化アルゴリズムともいう)で誤差が小さくなるように重みとバイアスを更新する

No.	出力	正解
		1
		1
		1
		1
		0
6	0.3	0
7	0.7	0
8	0.5	0

誤差E = 0.8

最適化関数
Adam

重みとバイアスを更新
各ニューロンのwとbが変わる

次の画像セットで
再度学習

model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy']

model.compile()で評価方法を決める
　loss=損失関数は'categorical_crossentropy'
　optimizer=最適化関数は'Adam'
　metrics=評価関数(モデルの評価方法)は['accuracy'](正解率)を指定

model.summary()

作ったモデルの要約を表示する

paramsはパラメーター(変数)のことでwとbの数

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 32)                25120
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

(784+1) × 32 = 25120

(32+1) × 10 = 330

25120 + 330 = 25450

まずはそのまま予測してみる

予測はmodel.predict()

test = model.predict(x_test)で、10000枚の予測結果がtestに入る

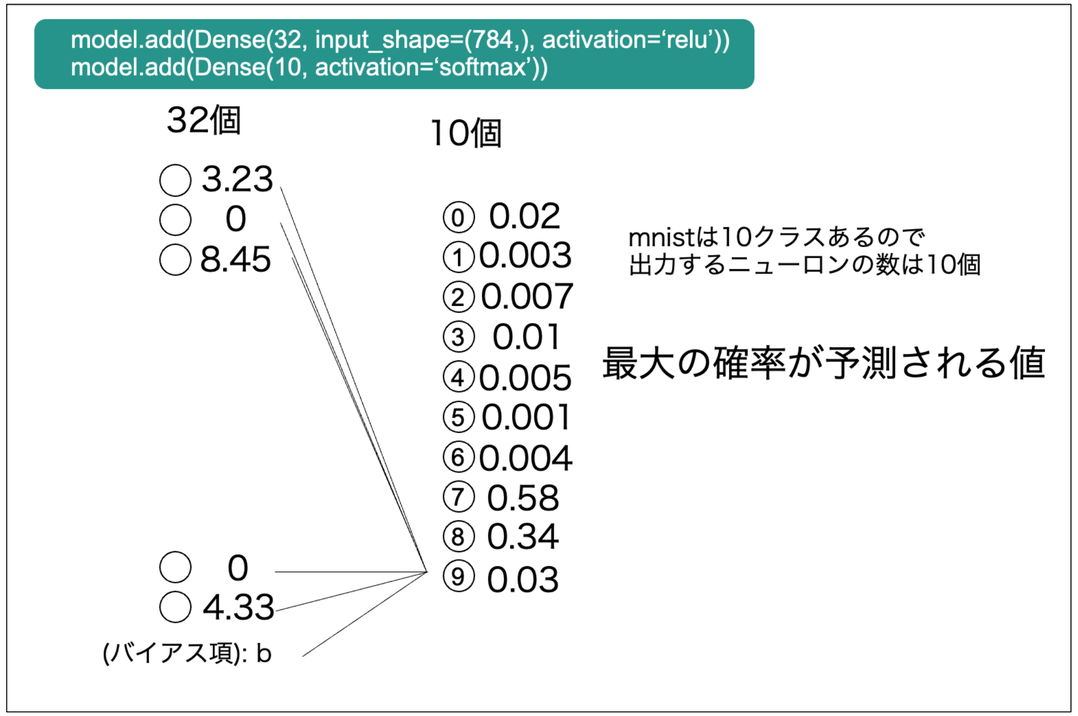
```
test = model.predict(x_test)
```

```
313/313 [==============================] - 5s 2ms/step
```

test.shapeは(10000,10)で10個の要素からなる列が10000 行

```
test.shape
```

```
(10000, 10)
```



[[x_test[0]の0の確率, x_test[0]の1の確率, … , x_test[0]の9の確率]
[x_test[1]の0の確率, x_test[1]の1の確率, … , x_test[1]の9の確率]

10000行

.
.
.

[[x_test[9998]の0の確率, x_test[9998]の1の確率, … , x_test[9998]の9の確率]
[x_test[9999]の0の確率, x_test[9999]の1の確率, … , x_test[9999]の9の確率]

10列

まずはそのまま予測してみる

予測はmodel.predict()

x_test[1]は2なので、

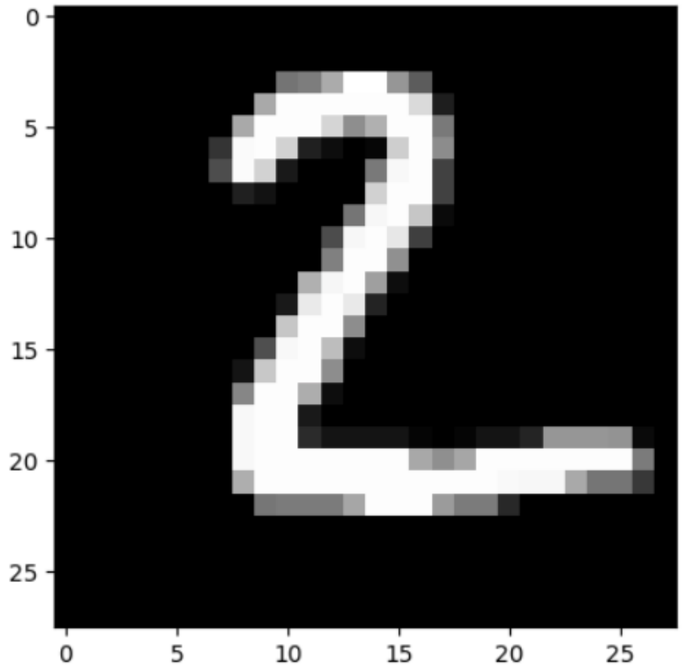
```
print(test[1])
print(y_test[1])
```

```
[0.06218787 0.13608842 0.06215866 0.06640156 0.28793582 0.04967605
 0.067972   0.10699823 0.11715493 0.04342646]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```

まだ学習していないので、予想結果は全く合っていない

学習するとどうなるか

```
import matplotlib.pyplot as plt
plt.imshow(x_test[1],'gray')
plt.show()
```




```
print(y_test[1])
```

```
2
```

result = model.fit(x_train, y_train, epochs=30, batch_size=64, validation_split=0.2)

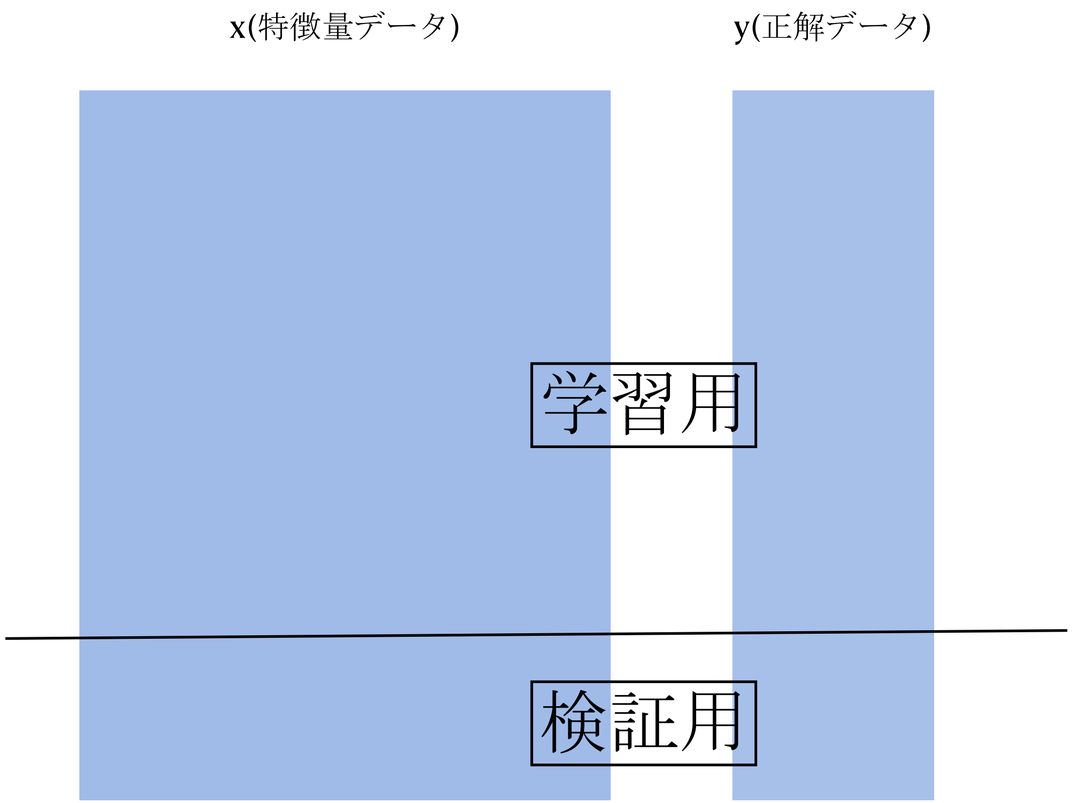
```
Epoch 1/30
750/750 [==============================] - 4s 4ms/step - loss: 0.4633 - accuracy: 0.8709 - val_loss: 0.2517 - val_accuracy: 0.9298
Epoch 2/30
750/750 [==============================] - 3s 4ms/step - loss: 0.2401 - accuracy: 0.9309 - val_loss: 0.2020 - val_accuracy: 0.9448
Epoch 3/30
750/750 [==============================] - 3s 4ms/step - loss: 0.1906 - accuracy: 0.9456 - val_loss: 0.1802 - val_accuracy: 0.9498

                                                                    .
                                                                    .
                                                                    .
Epoch 28/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0288 - accuracy: 0.9922 - val_loss: 0.1434 - val_accuracy: 0.9629
Epoch 29/30
750/750 [==============================] - 4s 5ms/step - loss: 0.0270 - accuracy: 0.9930 - val_loss: 0.1353 - val_accuracy: 0.9660
Epoch 30/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0264 - accuracy: 0.9930 - val_loss: 0.1381 - val_accuracy: 0.9654
```

model.fit()で実際に学習が行われる

学習は学習用データを全て使いません！
なぜだか覚えてますか？

機械学習ではそのままデータを丸ごと学習させない！

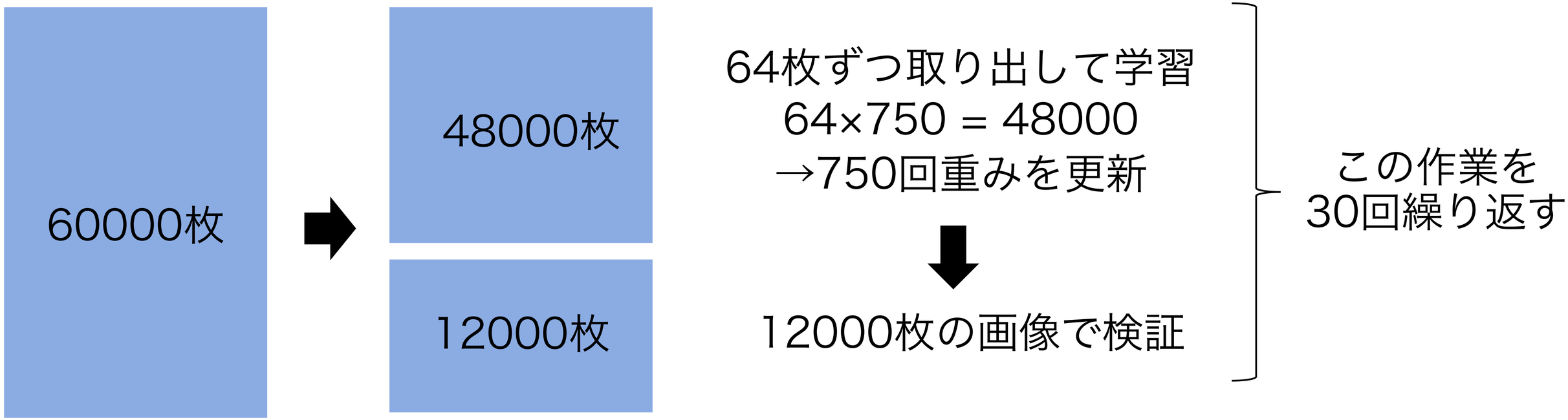
ホールドアウト法



新たにデータを用意するのではなく、
全データを学習用と検証用に分割する
(20~30%で分割するのが一般的)

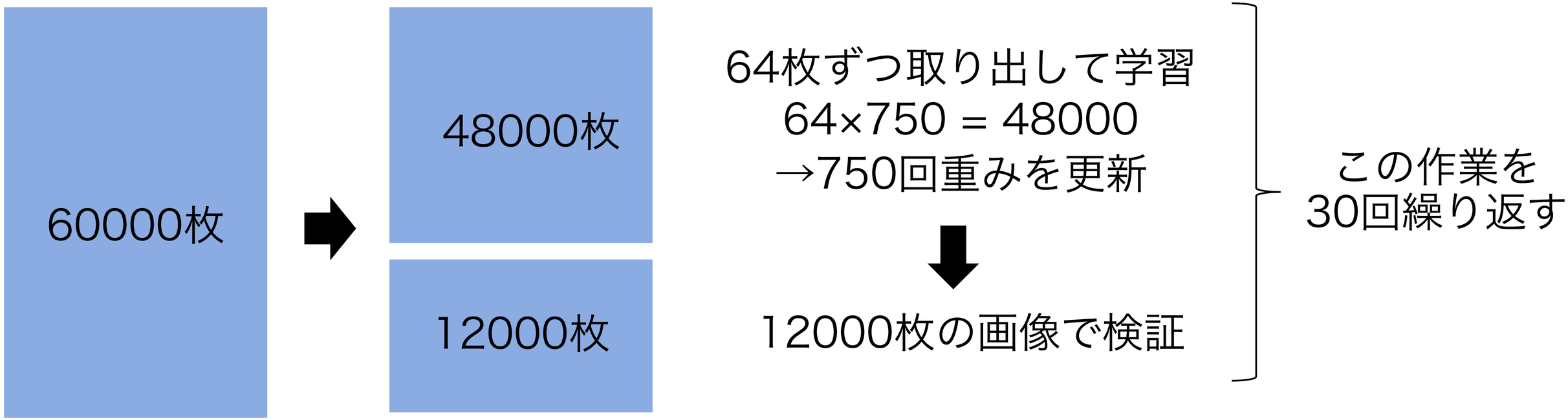
```
result = model.fit(x_train, y_train, epochs=30, batch_size=64, validation_split=0.2)
```

x_train : 60000枚の画像
y_train : 60000枚の画像の正解ラベル
epochs : 30回学習させる
batch_size : 64枚ずつ取り出して学習させる
validation_split : 学習用データの0.2(2割)を検証用データに使用する



result = model.fit(x_train, y_train, epochs=30, batch_size=64, validation_split=0.2)

```
Epoch 1/30
750/750 [==============================] - 4s 4ms/step - loss: 0.4633 - accuracy: 0.8709 - val_loss: 0.2517 - val_accuracy: 0.9298
Epoch 2/30
750/750 [==============================] - 3s 4ms/step - loss: 0.2401 - accuracy: 0.9309 - val_loss: 0.2020 - val_accuracy: 0.9448
Epoch 3/30
750/750 [==============================] - 3s 4ms/step - loss: 0.1906 - accuracy: 0.9456 - val_loss: 0.1802 - val_accuracy: 0.9498

                                                    .
                                                    .
                                                    .

Epoch 28/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0288 - accuracy: 0.9922 - val_loss: 0.1434 - val_accuracy: 0.9629
Epoch 29/30
750/750 [==============================] - 4s 5ms/step - loss: 0.0270 - accuracy: 0.9930 - val_loss: 0.1353 - val_accuracy: 0.9660
Epoch 30/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0264 - accuracy: 0.9930 - val_loss: 0.1381 - val_accuracy: 0.9654
```



最後に再度予測してみる

予測はmodel.predict()

一応名前を変更して学習後はtest2とする

```
test2 = model.predict(x_test)
```

```
313/313 [==============================] - 1s 2ms/step
```

2枚目を再度確認

```
print(test2[1])
```

```
[8.0701529e-11 3.9249046e-10 9.9999940e-01 8.3521821e-09 1.4485680e-28
 6.2819618e-07 2.5197353e-11 1.3647900e-22 2.3160052e-10 1.9060435e-24]
```

8.07e-11 は $8.07\times10^{-11} = 0.0000000000807$

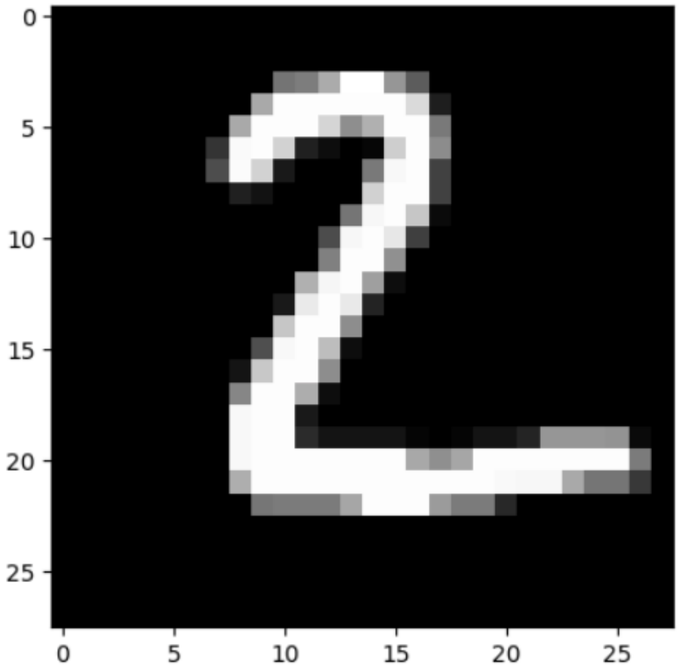
```
import numpy as np
print(np.around(test2[1],3))
print(y_test[1])
```

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

np.aroung(配列, num)で、
配列を少数第num位で四捨五入

99.9999...%で2と予想している

```
import matplotlib.pyplot as plt
plt.imshow(x_test[1],'gray')
plt.show()
```




```
print(y_test[1])
```

```
2
```

課題

・WebClassにある”kadai4.ipynb”をやってみましょう
・実行したら”学籍番号_名前_4.ipynb”という名前で保存して提出して下さい。

締め切りは2週間後の11/16の23:59です。
締め切りを過ぎた課題は受け取らないので注意して下さい

医療とAI・ビッグデータ応用

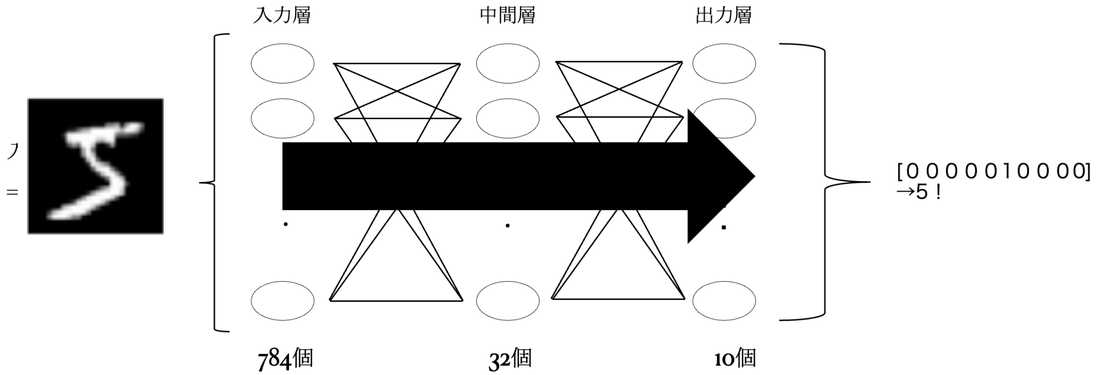
MLP②

統合教育機構
須藤毅顕

前回までの復習

モデルの作成

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
model.summary()
```



学習

```
result = model.fit(x_train, y_train, epochs=30, batch_size=64, validation_split=0.2)
```

```
Epoch 1/30
750/750 [==============================] - 4s 4ms/step - loss: 0.4633 - accuracy: 0.8709 - val_loss: 0.2517 - val_accuracy: 0.9298
Epoch 2/30
750/750 [==============================] - 3s 4ms/step - loss: 0.2401 - accuracy: 0.9309 - val_loss: 0.2020 - val_accuracy: 0.9448
Epoch 3/30
750/750 [==============================] - 3s 4ms/step - loss: 0.1906 - accuracy: 0.9456 - val_loss: 0.1802 - val_accuracy: 0.9498
                                                    :
Epoch 28/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0288 - accuracy: 0.9922 - val_loss: 0.1434 - val_accuracy: 0.9629
Epoch 29/30
750/750 [==============================] - 4s 5ms/step - loss: 0.0270 - accuracy: 0.9930 - val_loss: 0.1353 - val_accuracy: 0.9660
Epoch 30/30
750/750 [==============================] - 3s 4ms/step - loss: 0.0264 - accuracy: 0.9930 - val_loss: 0.1381 - val_accuracy: 0.9654
```

60000枚 → 48000枚 / 12000枚

64枚ずつ取り出して学習
64×750 = 48000
→750回重みを更新

12000枚の画像で検証

この作業を
30回繰り返す

予測

最後に再度予測してみる

予測はmodel.predict()

一応名前を変更して学習後はtest2とする

```
test2 = model.predict(x_test)
```

```
313/313 [==============================] - 1s 2ms/step
```

2枚目を再度確認

```
print(test2[1])
```

```
[8.0701529e-11 3.9249046e-10 9.9999940e-01 8.3521821e-09 1.4485680e-28
 6.2819618e-07 2.5197353e-11 1.3647900e-22 2.3160052e-10 1.9060435e-24]
```

8.07e-11 は $8.07 \times 10^{-11} = 0.0000000000807$

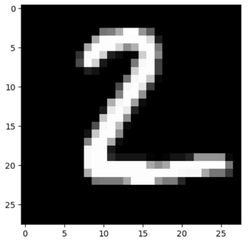
```
import numpy as np
print(np.around(test2[1],3))
print(y_test[1])
```

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

np.around(配列, num)で、
配列を少数第num位で四捨五入

99.9999...%で2と予想している

```
import matplotlib.pyplot as plt
plt.imshow(x_test[1],'gray')
plt.show()
```




```
print(y_test[1])
```

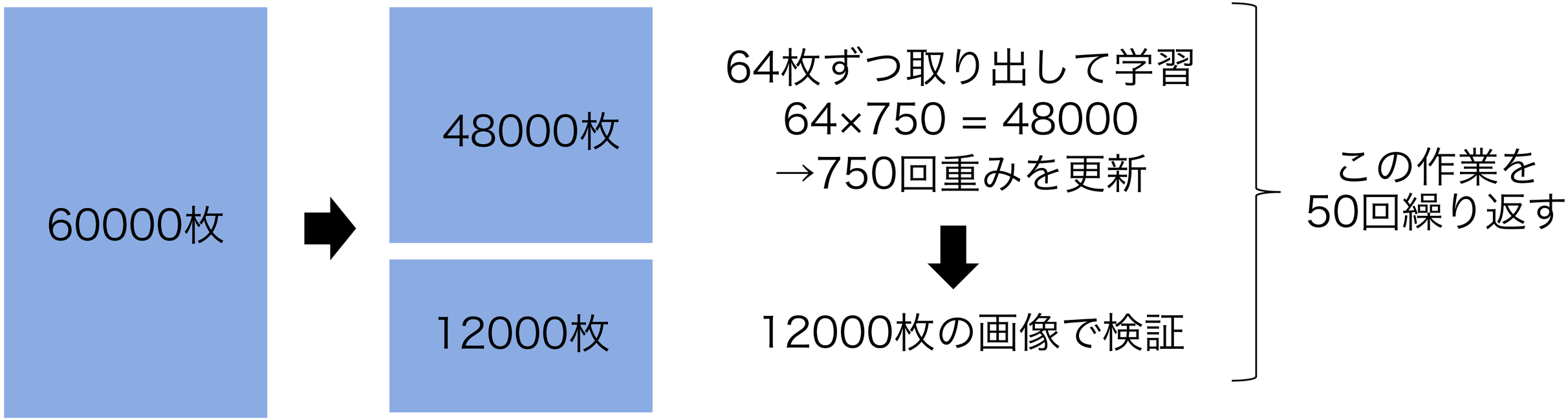
```
2
```

WebClassに「応用5_20231109_template.ipynb」があるので開いて実行していきましょう
(GPUに変更してください)

epochs=50で実行し直しましょう

```
result = model.fit(x_train, y_train, epochs=50, batch_size=64, validation_split=0.2)
```

```
Epoch 1/50 750/750 [==============================] - 2s 2ms/step - loss: 0.6172 - accuracy: 0.7892 - val_loss: 0.4628 - val_accuracy: 0.8407
Epoch 2/50 750/750 [==============================] - 1s 2ms/step - loss: 0.4376 - accuracy: 0.8482 - val_loss: 0.4198 - val_accuracy: 0.8545
Epoch 3/50 750/750 [==============================] - 1s 2ms/step - loss: 0.4035 - accuracy: 0.8605 - val_loss: 0.4151 - val_accuracy: 0.8565
Epoch 4/50 750/750 [==============================] - 1s 2ms/step - loss: 0.3793 - accuracy: 0.8673 - val_loss: 0.3926 - val_accuracy: 0.8601
Epoch 5/50 750/750 [==============================] - 1s 2ms/step - loss: 0.3628 - accuracy: 0.8721 - val_loss: 0.3776 - val_accuracy: 0.8664
                                                                .
                                                                .
Epoch 47/50 750/750 [==============================] - 1s 2ms/step - loss: 0.1602 - accuracy: 0.9409 - val_loss: 0.4712 - val_accuracy: 0.8773
Epoch 48/50 750/750 [==============================] - 1s 2ms/step - loss: 0.1564 - accuracy: 0.9427 - val_loss: 0.4850 - val_accuracy: 0.8735
Epoch 49/50 750/750 [==============================] - 1s 2ms/step - loss: 0.1570 - accuracy: 0.9425 - val_loss: 0.4780 - val_accuracy: 0.8767
Epoch 50/50 750/750 [==============================] - 1s 2ms/step - loss: 0.1542 - accuracy: 0.9434 - val_loss: 0.5010 - val_accuracy: 0.8724
```



結果の作図

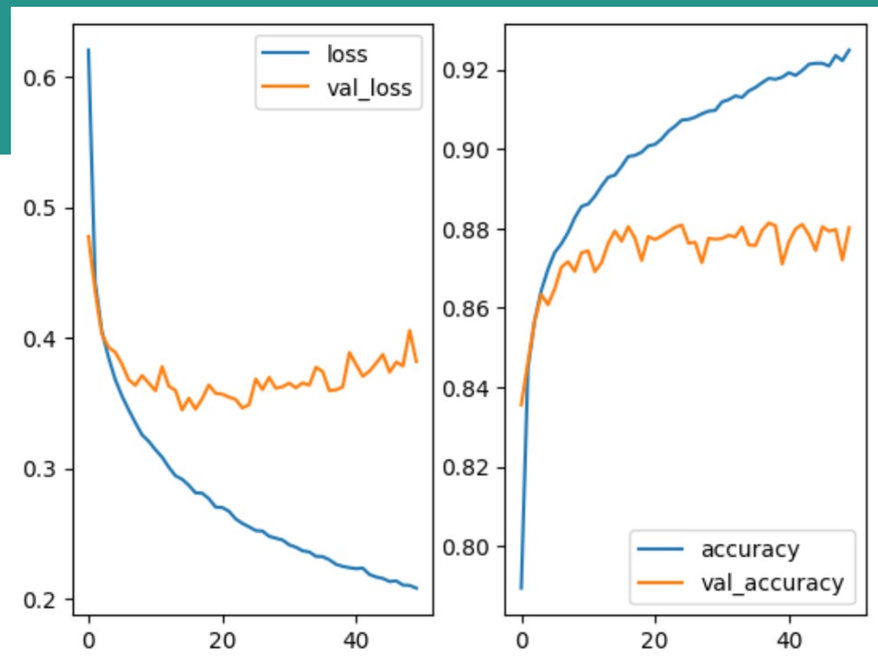
```
import matplotlib.pyplot as plt

plt.subplot(1,2,1)
plt.plot(result.history['loss'],label='loss')
plt.plot(result.history['val_loss'],label='val_loss')
plt.legend()
plt.subplot(1,2,2)
plt.plot(result.history['accuracy'],label='accuracy')
plt.plot(result.history['val_accuracy'],label='val_accuracy')
plt.legend()
plt.show()
```

縦1, 横2の1つ目

誤差(loss)の折れ線グラフ

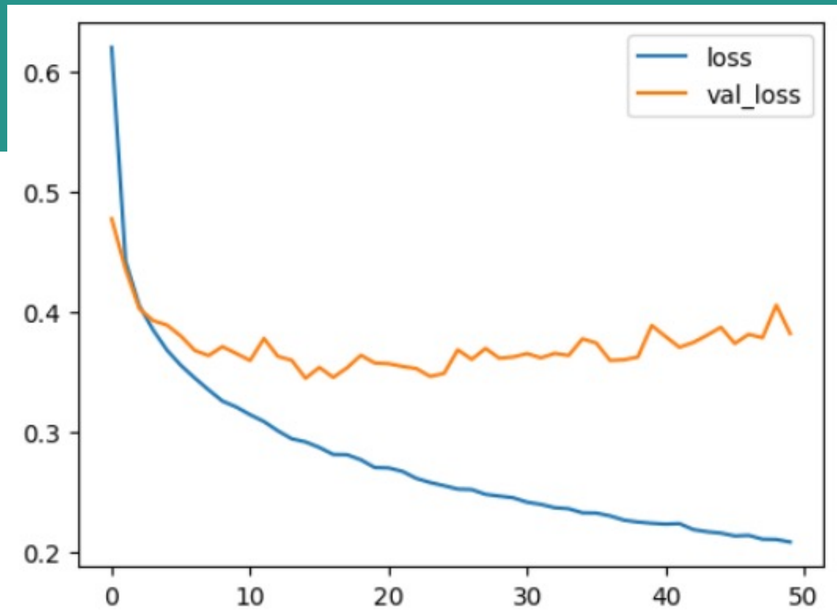
縦1, 横2の2つ目

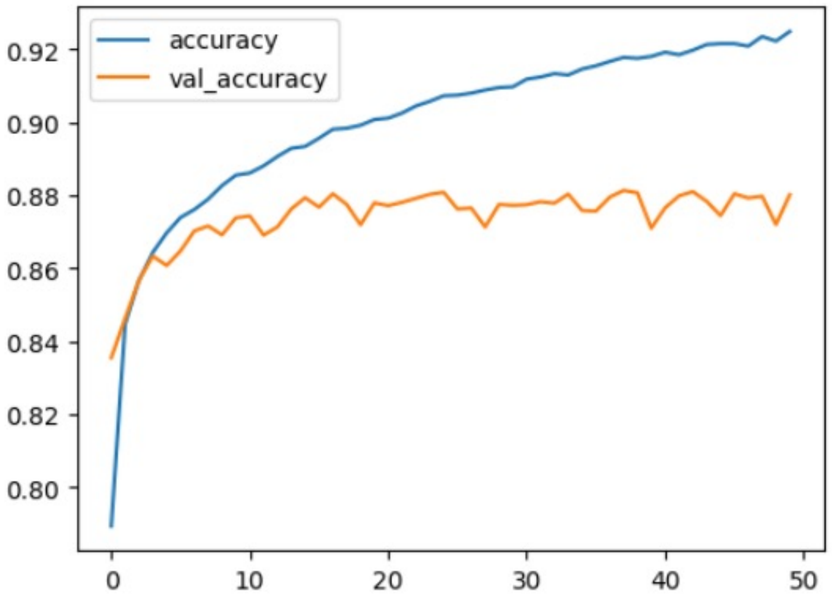
正解率(accuracy)の折れ線グラフ



結果の作図

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(result.history['loss'],label='loss')
plt.plot(result.history['val_loss'],label='val_loss')
plt.legend()
plt.subplot(1,2,2)
plt.plot(result.history['accuracy'],label='accuracy')
plt.plot(result.history['val_accuracy'],label='val_accuracy')
plt.legend()
plt.show()
```





result.historyの中身について

print(result.history)

{'loss': [0.6204796433448792, 0.4424095153808594, 0.4049777686595917, 0.38482892513275146, 0.3680422008037567, 0.3553660809993744, 0.34484514594078064, 0.3349671959877014, 0.32561713457107544, 0.3206530511379242, 0.3141988515853882, 0.30852627754211426, 0.3008130192756653, 0.29420095682144165, 0.29154443740844727, 0.2869287729263057, 0.28109729290008545, 0.28085023164749146, 0.27662160992622375, 0.2701963186264038, 0.26984351873397827, 0.2669712901154175, 0.26113253831863403, 0.25769904255867004, 0.2550542652606964, 0.2521992623806, 0.251813590526580 8, 0.24781620502471924, 0.24636663496494293, 0.24498197436332703, 0.2412831038236618, 0.2394535541534423 8, 0.2366907894611358 6, 0.2358267456293106, 0.23242957890033722, 0.2322139441967010 5, 0.229829117655754 1, 0.22631986439228058, 0.2247573286294937, 0.2237249165773391 7, 0.2230272144079208 4, 0.2234239131212234 5, 0.218465417623519 9, 0.2166386246681213 4, 0.2155087888240814 2, 0.2131433039903640 7, 0.213635131716728 2, 0.210426956415176 4, 0.210171923041343 7, 0.208052650094032 3],

'accuracy': [0.789354145526886, 0.8447291851043701, 0.8566874861717224, 0.8643541932106018, 0.8697083592414856, 0.8739166855812073, 0.8761041760444641, 0.8789583444595337, 0.8826666474342346, 0.8855208158493042, 0.886104166507721, 0.8880624771118164, 0.8906458616256714, 0.8928750157356262, 0.8933958411216736, 0.8956249952316284, 0.898104190826416, 0.8983749747276306, 0.8991875052452087, 0.9007708430290222, 0.9011458158493042, 0.9025416374206543, 0.9044791460037231, 0.9057499766349792, 0.9072708487510681, 0.9074375033378601, 0.9079999923706055, 0.9088541865348816, 0.9095208048820496, 0.9097291827201843, 0.9118333458900452, 0.9124166369438171, 0.9133541584014893, 0.9129791855812073, 0.9146041870117188, 0.9154791831970215, 0.9166874885559082, 0.9177916646003723, 0.9175416827201843, 0.9180625081062317, 0.9192083477973938, 0.9185208082199097, 0.9197708368301392, 0.9213333129882812, 0.9215624928474426, 0.9215624928474426, 0.9208750128746033, 0.9235208630561829, 0.922249972820282, 0.9248958230018616],

'val_loss': [0.4776163697242737, 0.4357101321220398, 0.40258854627609253, 0.39271479845046997, 0.3889164924621582, 0.3798101842403412, 0.3678809702396393, 0.3634991049766540 5, 0.3710480630397796 6, 0.3652504086494446, 0.3594720363616943 4, 0.3778232038021087 6, 0.3629121482372284, 0.3596421778202057, 0.3446278870105743 4, 0.3536724448204040 5, 0.3453534245491028, 0.3533513247966766 4, 0.3638331294059753 4, 0.3573807775974273 7, 0.3567837774753570 6, 0.3544683754444122 3, 0.3527243733406067, 0.3461835086345672 6, 0.3486511111259460 4, 0.3683689832687378, 0.3603420555591583, 0.3695347905158996 6, 0.3612982034683227 5, 0.3623996675014496, 0.3652012944221496 6, 0.3616231977939605 7, 0.3653631210327148 4, 0.3636190593242645, 0.3774882853031158 4, 0.3739925920963287 4, 0.3594782650470733 6, 0.3600547611713409 4, 0.3623170256614685, 0.3886697292327881, 0.3794552385807037 4, 0.3704958558082580 6, 0.3743937611579895, 0.3805387318134308, 0.3871587216854095 5, 0.373552829027175 9, 0.3813273906707763 7, 0.3784124851226806 6, 0.4056456685066223, 0.3818448781967163],

'val_accuracy': [0.8355000019073486, 0.8462499976158142, 0.8567500114440918, 0.8633333444595337, 0.8607500195503235, 0.8647500276565552, 0.8702499866485596, 0.8715833425521851, 0.8691666722297668, 0.8738333582878113, 0.8743333220481873, 0.8690833449363708, 0.8713333606719971, 0.8762500286102295, 0.8793333172798157, 0.8767499923706055, 0.8804166913032532, 0.8774999976158142, 0.871916651725769, 0.877916693687439, 0.8771666884422302, 0.878083348274231, 0.8791666626930237, 0.8802499771118164, 0.8808333277702332, 0.8762500286102295, 0.8765000104904175, 0.8713333606719971, 0.8774999976158142, 0.8772500157356262, 0.8774166703224182, 0.878250002861023, 0.8778333067893982, 0.8803333044052124, 0.8758333325386047, 0.875666677951812 7, 0.8794999718666077, 0.8813333511352539, 0.8806666731834412, 0.8709999918937683, 0.8765833377838135, 0.8798333406448364, 0.8809999823570251, 0.878333333015441 9, 0.8744166493415833, 0.8804166913032532, 0.8792499899864197, 0.8797500133514404, 0.871999979019165, 0.8801666498184204]}

result.historyの中身について

print(result.history)　　エポック50回分の誤差と正解率

{'loss': [0.6204796433448792, 0.4424095153808594, 0.4049777686595917, 0.3848289251327514 6, 0.3680422008037567, 0.3553660809993744, 0.34484514594078064, 0.3349671959877014, 0.32561713457107544, 0.3206530511379242, 0.3141988515853882, 0.30852627754211426, 0.3008130192756653, 0.2942009568214416 5, 0.29154443740844727, 0.2869287729263305 7, 0.28109729290008545, 0.2808502316474914 6, 0.27662160992622375, 0.2701963186264038, 0.2698435187339782 7, 0.2669712901115417 5, 0.26113253831863403, 0.2576990425586700 4, 0.2550542652606964, 0.2521992623806, 0.2518135905265808, 0.2478162050247192 4, 0.24636663496494293, 0.2449819743633270 3, 0.2412831038236618, 0.2394535541534423 8, 0.2366907894611358 6, 0.2358267456293106, 0.2324295789003372 2, 0.2322139441967010 5, 0.229829117655754 1, 0.2263198643922805 8, 0.2247573286294937, 0.2237249165773391 7, 0.2230272144079208 4, 0.2234239131212234 5, 0.218465417623519 9, 0.2166386246681213 4, 0.2155087888240814 2, 0.2131433039903640 7, 0.2136351317167282, 0.210426956415176 4, 0.2101719230413437, 0.2080526500940323],

'accuracy': [0.789354145526886, 0.8447291851043701, 0.8566874861717224, 0.8643541932106018, 0.8697083592414856, 0.8739166855812073, 0.8761041760444641, 0.8789583444595337, 0.8826666474342346, 0.8855208158493042, 0.886104166507721, 0.8880624771118164, 0.8906458616256714, 0.8928750157356262, 0.8933958411216736, 0.8956249952316284, 0.898104190826416, 0.8983749747276306, 0.8991875052452087, 0.9007708430290222, 0.9011458158493042, 0.9025416374206543, 0.9044791460037231, 0.9057499766349792, 0.9072708487510681, 0.9074375033378601, 0.9079999923706055, 0.9088541865348816, 0.9095208048820496, 0.9097291827201843, 0.9118333458900452, 0.9124166369438171, 0.9133541584014893, 0.9129791855812073, 0.9146041870117188, 0.9154791831970215, 0.9166874885559082, 0.9177916646003723, 0.9175416827201843, 0.9180625081062317, 0.9192083477973938, 0.9185208082199097, 0.9197708368301392, 0.9213333129882812, 0.9215624928474426, 0.9215624928474426, 0.9208750128746033, 0.9235208630561829, 0.922249972820282, 0.9248958230018616],

'val_loss': [0.4776163697242737, 0.4357101321220398, 0.40258854627609253, 0.39271479845046997, 0.3889164924621582, 0.3798101842403412, 0.3678809702396393, 0.3634991049766540 5, 0.37104806303977966, 0.3652504086494446, 0.3594720363616943 4, 0.3778232038021087 6, 0.3629121482372284, 0.3596421778202057, 0.3446278870105743 4, 0.3536724448204040 5, 0.3453534245491028, 0.3533513247966766 4, 0.3638331294059753 4, 0.3573807775974273 7, 0.3567837774753570 6, 0.3544683754444122 3, 0.3527243733406067, 0.3461835086345672 6, 0.3486511111259460 4, 0.3683689832687378, 0.3603420555591583, 0.3695347905158996 6, 0.3612982034683227 5, 0.3623996675014496, 0.3652012944221496 6, 0.3616231977939605 7, 0.3653631210327148 4, 0.3636190593242645, 0.3774882853031158 4, 0.3739925920963287 4, 0.3594782650470733 6, 0.3600547611713409 4, 0.3623170256614685, 0.3886697292327881, 0.3794552385807037 4, 0.3704958558082580 6, 0.3743937611579895, 0.3805387318134308, 0.3871587216854095 5, 0.3735528290271759, 0.3813273906707763 7, 0.3784124851226806 6, 0.4056456685066223, 0.3818448781967163],

'val_accuracy': [0.8355000019073486, 0.8462499976158142, 0.8567500114440918, 0.8633333444595337, 0.8607500195503235, 0.8647500276565552, 0.8702499866485596, 0.8715833425521851, 0.8691666722297668, 0.873833358287811 3, 0.8743333220481873, 0.8690833449363708, 0.8713333606719971, 0.8762500286102295, 0.8793333172798157, 0.8767499923706055, 0.8804166913032532, 0.8774999976158142, 0.871916651725769, 0.877916693687439, 0.8771666884422302, 0.878083348274231, 0.8791666626930237, 0.8802499771118164, 0.8808333277702332, 0.8762500286102295, 0.8765000104904175, 0.8713333606719971, 0.8774999976158142, 0.8772500157356262, 0.8774166703224182, 0.878250002861023, 0.8778333067893982, 0.8803333044052124, 0.8758333325386047, 0.8756666779518127, 0.8794999718666077, 0.8813333511352539, 0.8806666731834412, 0.8709999918937683, 0.8765833377838135, 0.8798333406448364, 0.8809999823570251, 0.878333330154419, 0.8744166493415833, 0.8804166913032532, 0.8792499899864197, 0.8797500133514404, 0.871999979019165, 0.8801666498184204]}

{‘loss’:[1回目の学習用データの損失,2回目の学習用データの損失,...,50回目の学習用データの損失],
‘accuracy’:[1回目の学習用データの正解率,2回目の学習用データの正解率,...,50回目の学習用データの正解率],
‘val_loss’:[1回目の検証用データの損失,2回目の検証用データの損失,...,50回目の検証用データの損失],
‘val_accuracy’:[1回目の検証用データの正解率,2回目の検証用データの正解率,...,50回目の検証用データの正解率]}

辞書型のデータの取得方法

print(result.history['loss'])

[0.6204796433448792, 0.4424095153808594, 0.4049777686595917, 0.38482892513275146, 0.3680422008037567, 0.3553660809993744, 0.344845145940780 64, 0.3349671959877014, 0.32561713457107544, 0.320653051137924 2, 0.3141988515853882, 0.30852627754211426, 0.3008130192756653, 0.2942009568214 4165, 0.29154443740844727, 0.28692877292633057, 0.28109729290008545, 0.28085023164749146, 0.27662160992622375, 0.2701963186264038, 0.26984351873397827, 0.2669712901115 4175, 0.26113253831863403, 0.25769904255867004, 0.2550542652606964, 0.2521992623806, 0.25181359052658 08, 0.24781620502471924, 0.24636663496494293, 0.24498197436332703, 0.2412831038236618, 0.2394535541534 4238, 0.23669078946113586, 0.2358267456293106, 0.2324295789003 3722, 0.2322139441967 0105, 0.2298291176557541, 0.2263198643922 8058, 0.2247573286294937, 0.2237249165773 3917, 0.2230272144079 2084, 0.2234239131212 2345, 0.2184654176235199, 0.2166386246681 2134, 0.2155087888240 8142, 0.2131433039903 6407, 0.2136351317167282, 0.2104269564151764, 0.2101719230413437, 0.2080526500940323]

出力結果は各エポックの誤差がリスト になっている

x = [要素,要素,...,要素]

a = [1,1,3,3,3]
これはリスト型

x = {key:value,key:value,...,key:value}

b = {'name':'sudo','age':36}
これは辞書型

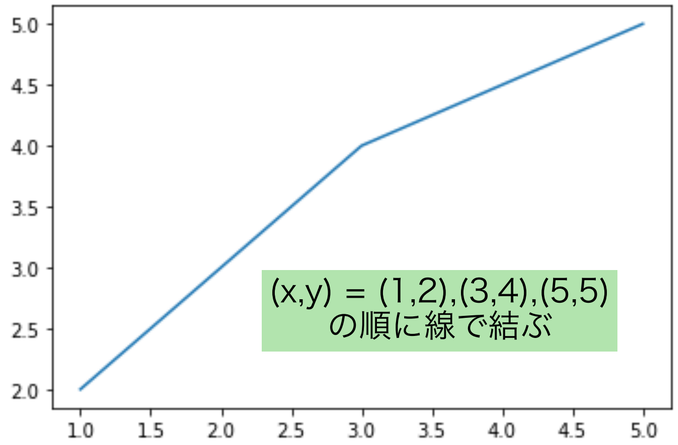
```
b = {'name':'sudo','age':36}
print(type(b))
print(b['name'])
print(b['age'])
```

```
<class 'dict'>
sudo
36
```

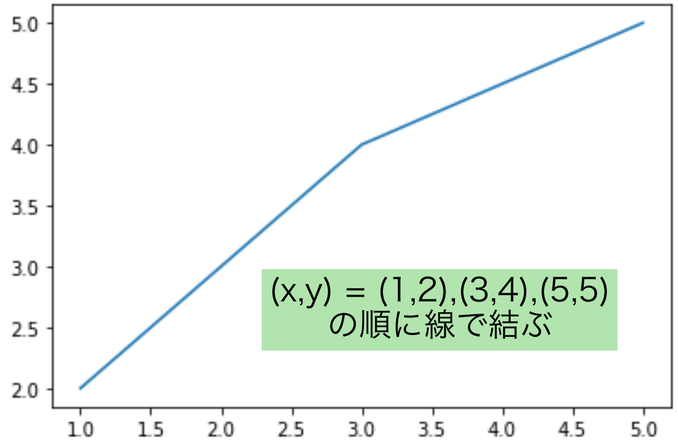
辞書型は変数名[key]で
valueを取り出せる!

result.history['loss']で{'loss':[〜〜], ...}の[〜〜]を取り出している

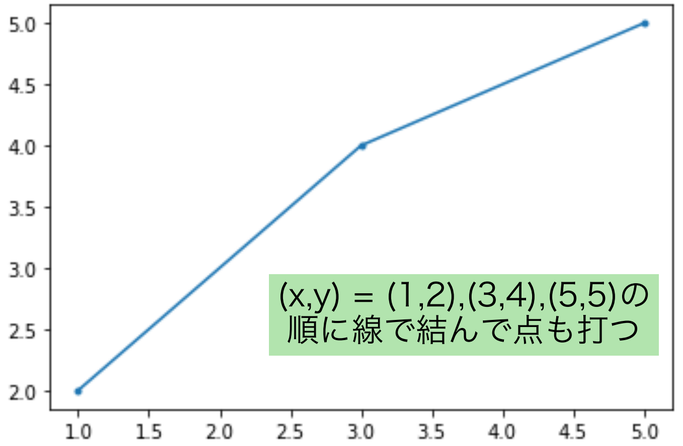
```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y)
plt.show()
```



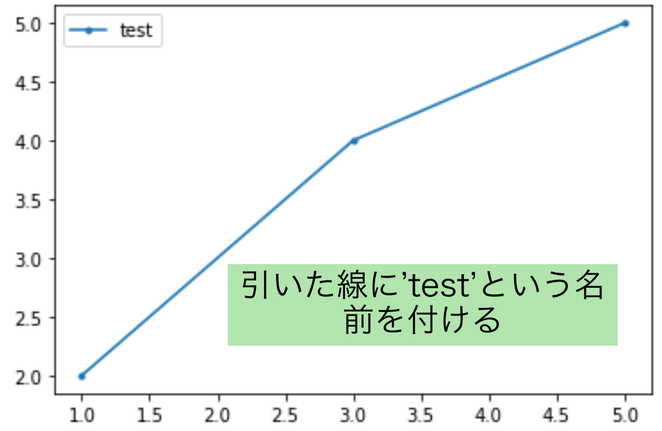
```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y)
plt.show()
```



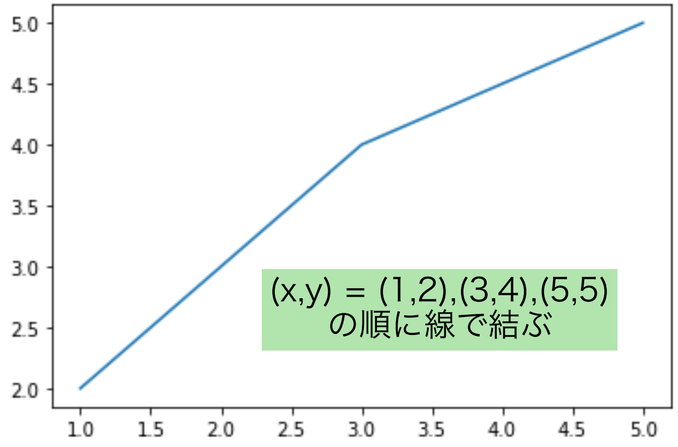

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.')
plt.show()
```



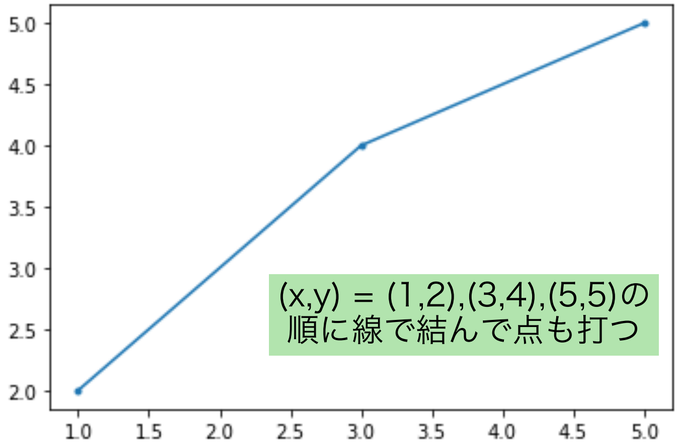

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.',label='test')
plt.legend()
plt.show()
```



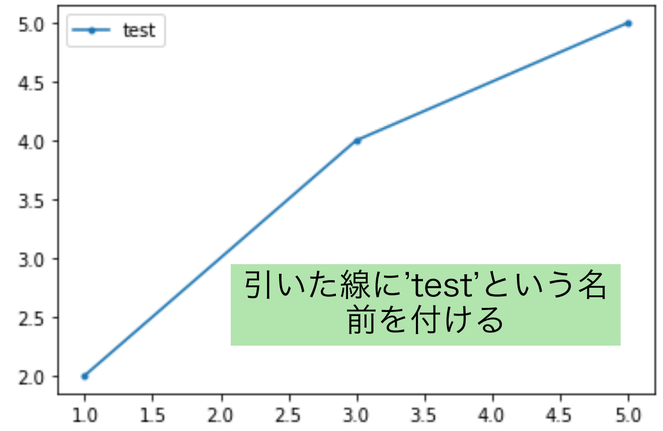
```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y)
plt.show()
```



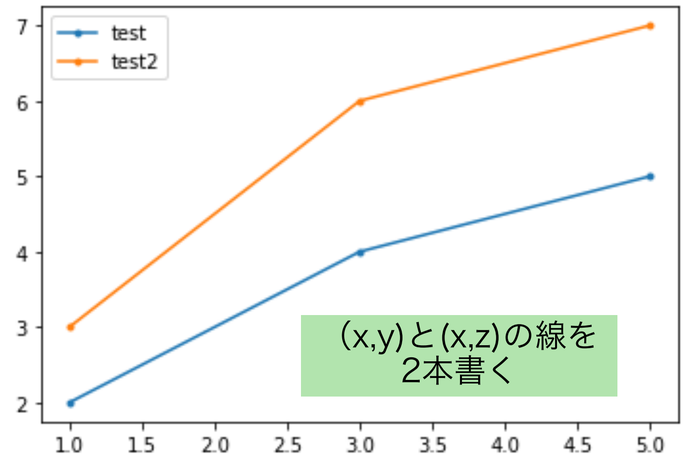

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.')
plt.show()
```



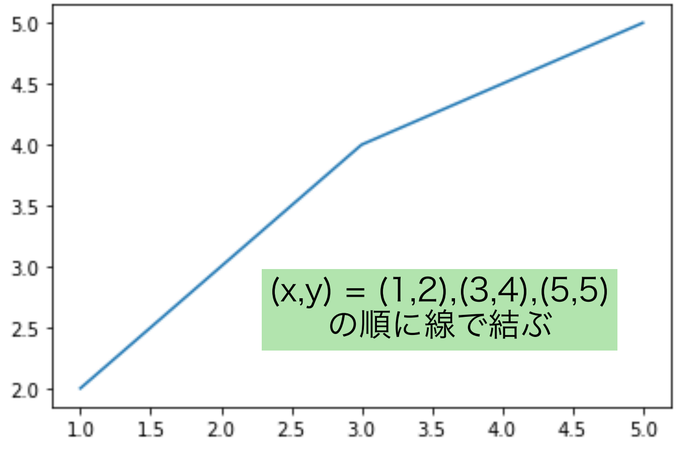

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.',label='test')
plt.legend()
plt.show()
```



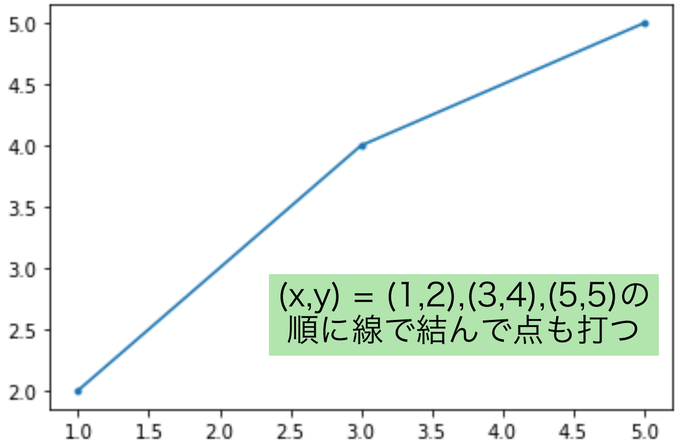

```
x = [1,3,5]
y = [2,4,5]
z = [3,6,7]
plt.plot(x,y,marker='.',label='test')
plt.plot(x,z,marker='.',label='test2')
plt.legend()
plt.show()
```



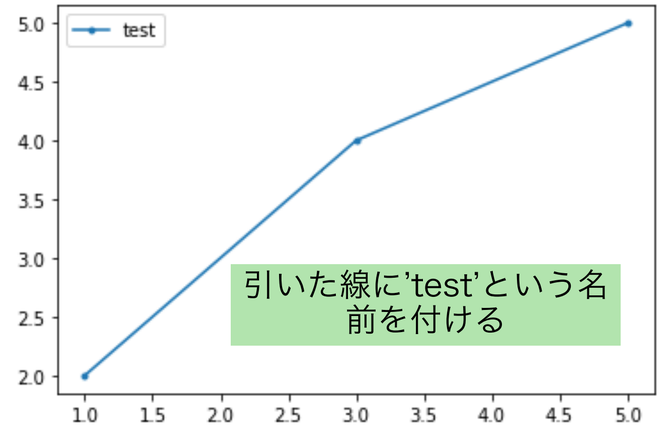
```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y)
plt.show()
```



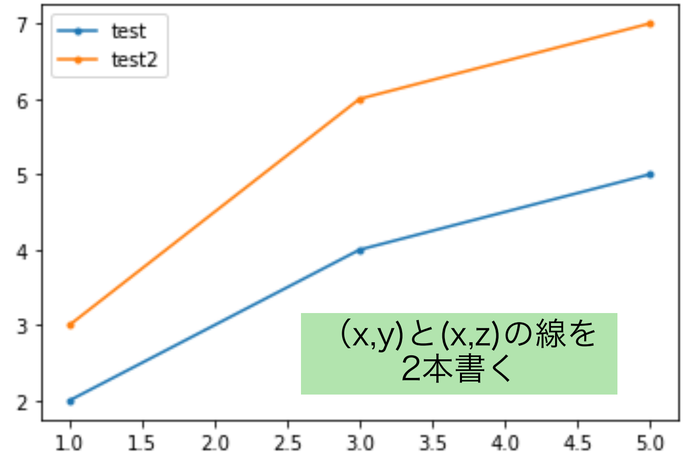

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.')
plt.show()
```



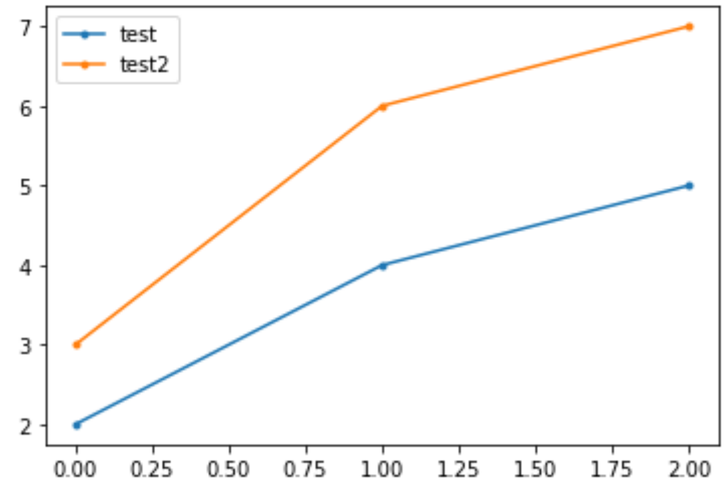

```
x = [1,3,5]
y = [2,4,5]
plt.plot(x,y,marker='.',label='test')
plt.legend()
plt.show()
```




```
x = [1,3,5]
y = [2,4,5]
z = [3,6,7]
plt.plot(x,y,marker='.',label='test')
plt.plot(x,z,marker='.',label='test2')
plt.legend()
plt.show()
```




```
y = [2,4,5]
z = [3,6,7]
plt.plot(y,marker='.',label='test')
plt.plot(z,marker='.',label='test2')
plt.legend()
plt.show()
```



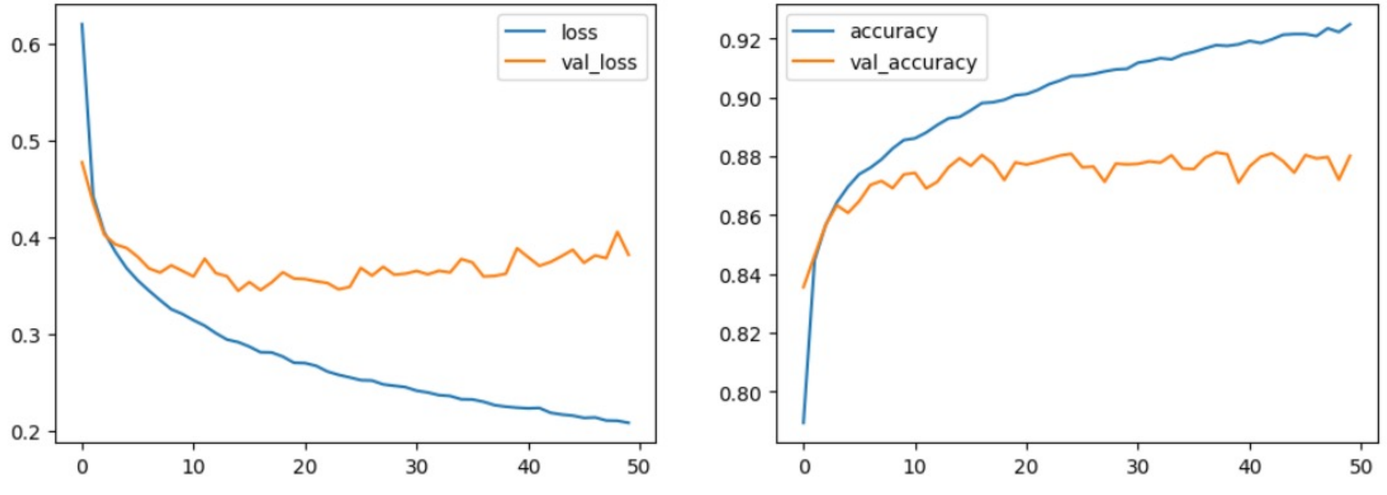
x軸の変数が与えられない時はy軸の個数だけ順に0,1,2,3,..と与えられる。

```
plt.plot(result.history['loss'],label='loss')
plt.plot(result.history['val_loss'],label='val_loss')
```

'loss': [0.6204796433448792, 0.4424095153808594, 0.4049777686595917, 0.3848289251327514 6, 0.3680422008037567, 0.3553660809993744, 0.34484514594078064, 0.3349671959877014, 0.3256171345710754 4, 0.3206530511379242, 0.3141988515853882, 0.30852627754211426, 0.3008130192756653, 0.2942009568214416 5, 0.29154443740844727, 0.28692877292633057, 0.28109729290008545, 0.280850231647491 46, 0.27662160992622375, 0.2701963186264038, 0.26984351873397827, 0.2669712901115417 5, 0.26113253831863403, 0.2576990425586700 4, 0.2550542652606964, 0.2521992623806, 0.2518135905265808, 0.24781620502471924, 0.2463666349649429 3, 0.2449819743633270 3, 0.2412831038236618, 0.23945355415344238, 0.2366907894611358 6, 0.2358267456293106, 0.2324295789003372 2, 0.2322139441967010 5, 0.2298291176557541, 0.2263198643922805 8, 0.2247573286294937, 0.2237249165773391 7, 0.2230272144079208 4, 0.2234239131212234 5, 0.218465417623519 9, 0.2166386246681213 4, 0.2155087888240814 2, 0.2131433039903640 7, 0.2136351317167282, 0.2104269564151764, 0.2101719230413437, 0.2080526500940323],

'val_loss': [0.4776163697242737, 0.4357101321220398, 0.4025885462760925 3, 0.3927147984504699 7, 0.388916492462158 2, 0.3798101842403412, 0.367880970239639 3, 0.3634991049766540 5, 0.3710480630397796 6, 0.3652504086494446, 0.3594720363616943 4, 0.3778232038021087 6, 0.362912148237228 4, 0.3596421778202057, 0.3446278870105743 4, 0.3536724448204040 5, 0.3453534245491028, 0.3533513247966766 4, 0.3638331294059753 4, 0.3573807775974273 7, 0.3567837774753570 6, 0.3544683754444122 3, 0.3527243733406067, 0.3461835086345672 6, 0.3486511111259460 4, 0.3683689832687378, 0.3603420555591583, 0.3695347905158996 6, 0.3612982034683227 5, 0.3623996675014496, 0.3652012944221496 6, 0.3616231977939605 7, 0.3653631210327148 4, 0.3636190593242645, 0.3774882853031158 4, 0.3739925920963287 4, 0.3594782650470733 6, 0.3600547611713409 4, 0.362317025661468 5, 0.3886697292327881, 0.3794552385807037 4, 0.3704958558082580 6, 0.3743937611579895, 0.3805387318134308, 0.3871587216854095 5, 0.3735528290271759, 0.3813273906707763 7, 0.3784124851226806 6, 0.4056456685066223, 0.3818448781967163],

どちらも要素の数が50個のリスト
xは[0,1,2,...,49]が省略されている



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

model.evaluate()で評価

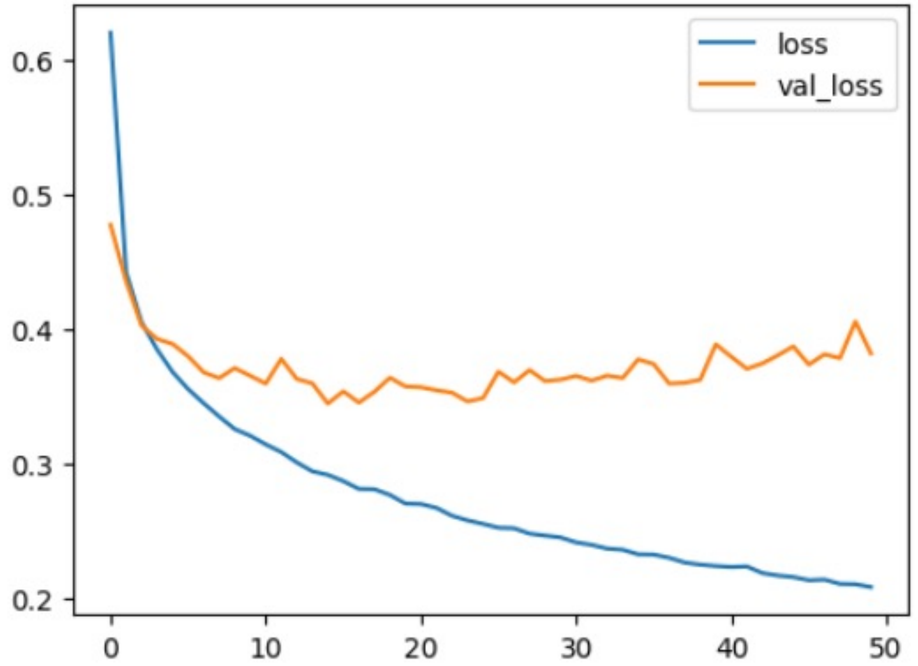
出来上がったモデルを別で用意している10000枚の画像で評価する

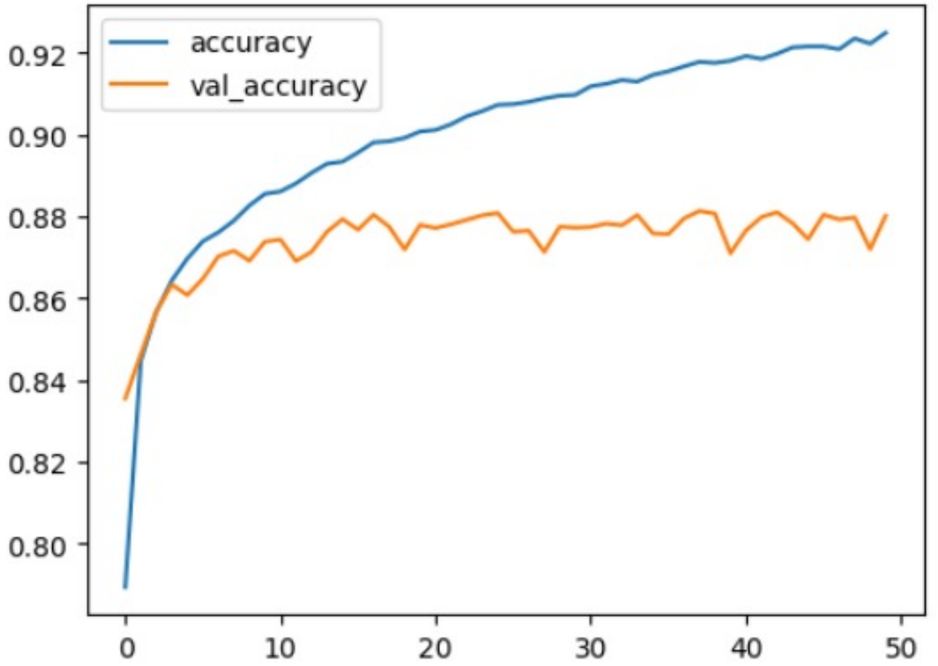
Test loss: 0.40565115213394165
Test accuracy: 0.8694000244140625

score = model.evaluate(特徴量,正解)
で、scoreにはmodelを用いた予測結果の損失と正解率が代入される
score[0]で損失、score[1]で正解率が得られる。

```
a = 5
print(a)          出力 5
print(‘aは’)      出力 aは
print(‘aは’,a)    出力 aは5
```

今回のモデルで学習した結果





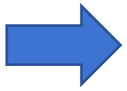
Test loss: 0.40565115213394165
Test accuracy: 0.8694000244140625

もっと精度をあげたい

ニューロンの数を増やす

```
model = Sequential()
model.add(Dense(32,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 32)                25120
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

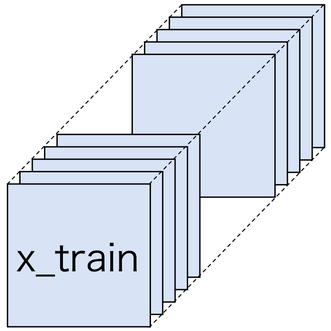


```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

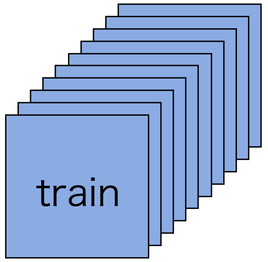
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

結果の分析

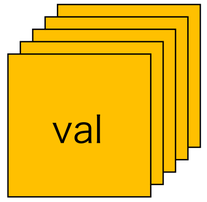
60000枚



48000枚

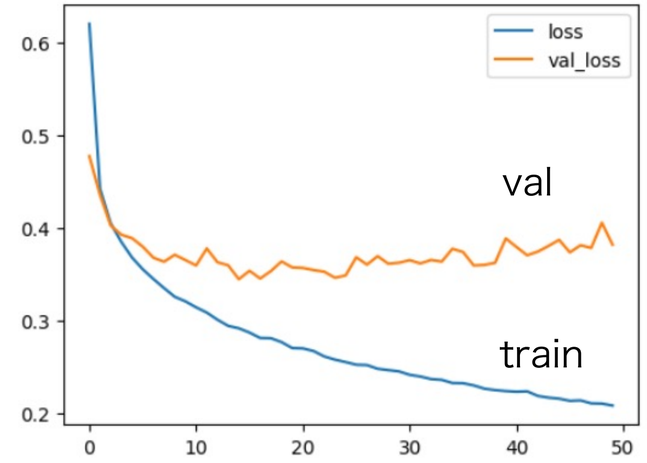


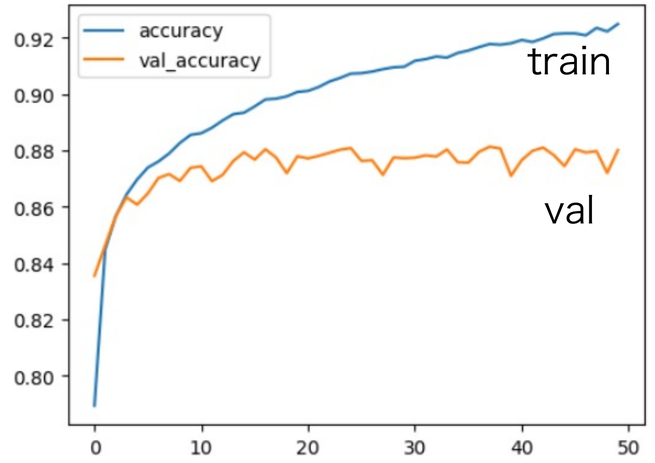
12000枚



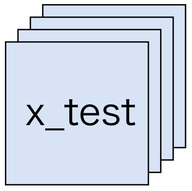
①trainを64枚ずつランダムに取り出して、学習、更新(750回/epoch)

②1epoch終了後、trainとval両方でlossとaccuracyを算出(50epoch)





10000枚



③最後にtestで最終評価
(lossとaccuracy)

Test loss: 0.40565115213394165
Test accuracy: 0.8694000244140625

result.historyの中身について

print(result.history)

{'loss': [0.6204796433448792, 0.4424095153808594, 0.4049777686595917, 0.38482892513275146, 0.3680422008037567, 0.3553660809993744, 0.34484514594078064, 0.3349671959877014, 0.32561713457107544, 0.3206530511379242, 0.3141988515853882, 0.30852627754211426, 0.3008130192756653, 0.29420095682144165, 0.29154443740844727, 0.2869287729263057, 0.28109729290008545, 0.28085023164749146, 0.27662160992622375, 0.2701963186264038, 0.26984351873397827, 0.2669712901154175, 0.26113253831863403, 0.25769904255867004, 0.2550542652606964, 0.2521992623806, 0.251813590526580 8, 0.24781620502471924, 0.24636663496494293, 0.24498197436332703, 0.2412831038236618, 0.2394535541534423 8, 0.2366907894611358 6, 0.2358267456293106, 0.23242957890033722, 0.2322139441967010 5, 0.229829117655754 1, 0.22631986439228058, 0.2247573286294937, 0.2237249165773391 7, 0.2230272144079208 4, 0.2234239131212234 5, 0.218465417623519 9, 0.2166386246681213 4, 0.2155087888240814 2, 0.2131433039903640 7, 0.213635131716728 2, 0.210426956415176 4, 0.210171923041343 7, 0.2080526500940323],

'accuracy': [0.789354145526886, 0.8447291851043701, 0.8566874861717224, 0.8643541932106018, 0.8697083592414856, 0.8739166855812073, 0.8761041760444641, 0.8789583444595337, 0.8826666474342346, 0.8855208158493042, 0.886104166507721, 0.8880624771118164, 0.8906458616256714, 0.8928750157356262, 0.8933958411216736, 0.8956249952316284, 0.898104190826416, 0.8983749747276306, 0.8991875052452087, 0.9007708430290222, 0.9011458158493042, 0.9025416374206543, 0.9044791460037231, 0.9057499766349792, 0.9072708487510681, 0.9074375033378601, 0.9079999923706055, 0.9088541865348816, 0.9095208048820496, 0.9097291827201843, 0.9118333458900452, 0.9124166369438171, 0.9133541584014893, 0.9129791855812073, 0.9146041870117188, 0.9154791831970215, 0.9166874885559082, 0.9177916646003723, 0.9175416827201843, 0.9180625081062317, 0.9192083477973938, 0.9185208082199097, 0.9197708368301392, 0.9213333129882812, 0.9215624928474426, 0.9215624928474426, 0.9208750128746033, 0.9235208630561829, 0.922249972820282, 0.9248958230018616],

'val_loss': [0.4776163697242737, 0.4357101321220398, 0.40258854627609253, 0.39271479845046997, 0.3889164924621582, 0.3798101842403412, 0.3678809702396393, 0.3634991049766540 5, 0.3710480630397796 6, 0.3652504086494446, 0.3594720363616943 4, 0.3778232038021087 6, 0.3629121482372284, 0.3596421778202057, 0.3446278870105743 4, 0.3536724448204040 5, 0.3453534245491028, 0.3533513247966766 4, 0.3638331294059753 4, 0.3573807775974273 7, 0.3567837774753570 6, 0.3544683754444122 3, 0.3527243733406067, 0.3461835086345672 6, 0.3486511111259460 4, 0.3683689832687378, 0.3603420555591583, 0.3695347905158996 6, 0.3612982034683227 5, 0.3623996675014496, 0.3652012944221496 6, 0.3616231977939605 7, 0.3653631210327148 4, 0.3636190593242645, 0.3774882853031158 4, 0.3739925920963287 4, 0.3594782650470733 6, 0.3600547611713409 4, 0.3623170256614685, 0.3886697292327881, 0.3794552385807037 4, 0.3704958558082580 6, 0.3743937611579895, 0.3805387318134308, 0.3871587216854095 5, 0.373552829027175 9, 0.3813273906707763 7, 0.3784124851226806 6, 0.4056456685066223, 0.3818448781967163],

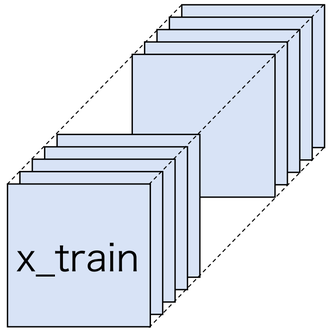
'val_accuracy': [0.8355000019073486, 0.8462499976158142, 0.8567500114440918, 0.8633333444595337, 0.8607500195503235, 0.8647500276565552, 0.8702499866485596, 0.8715833425521851, 0.8691666722297668, 0.8738333582878113, 0.8743333220481873, 0.8690833449363708, 0.8713333606719971, 0.8762500286102295, 0.8793333172798157, 0.8767499923706055, 0.8804166913032532, 0.8774999976158142, 0.871916651725769, 0.877916693687439, 0.8771666884422302, 0.878083348274231, 0.8791666626930237, 0.8802499771118164, 0.8808333277702332, 0.8762500286102295, 0.8765000104904175, 0.8713333606719971, 0.8774999976158142, 0.8772500157356262, 0.8774166703224182, 0.878250002861023, 0.8778333067893982, 0.8803333044052124, 0.8758333325386047, 0.875666677951812 7, 0.879499971866607 7, 0.8813333511352539, 0.8806666731834412, 0.8709999918937683, 0.8765833377838135, 0.8798333406448364, 0.8809999823570251, 0.878333333015441 9, 0.8744166493415833, 0.8804166913032532, 0.8792499899864197, 0.8797500133514404, 0.871999979019165, 0.8801666498184204]}

'loss': 0.2080526500940323,
'accuracy': 0.9248958230018616,

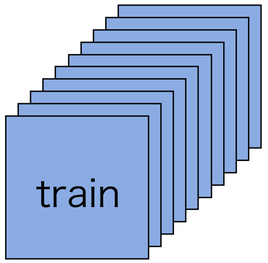
'val_loss': 0.3818448781967163,
'val_accuracy': 0.8801666498184204

結果の分析

60000枚



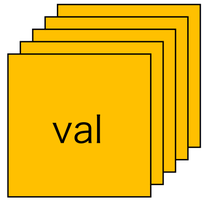
48000枚



①trainを64枚ずつランダムに取り出して、学習、更新(750回/epoch)

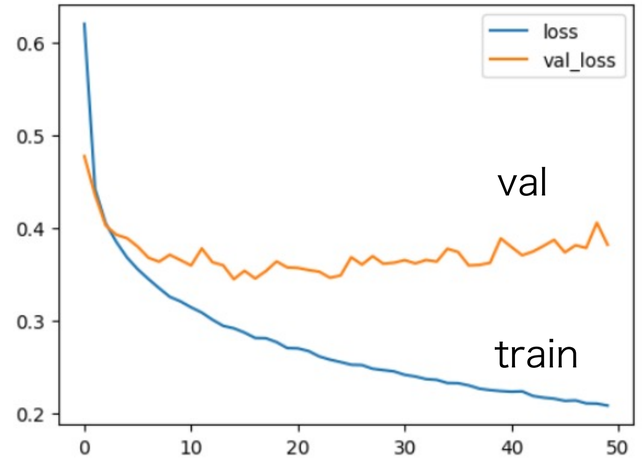
'loss': 0.21
'accuracy': 0.92

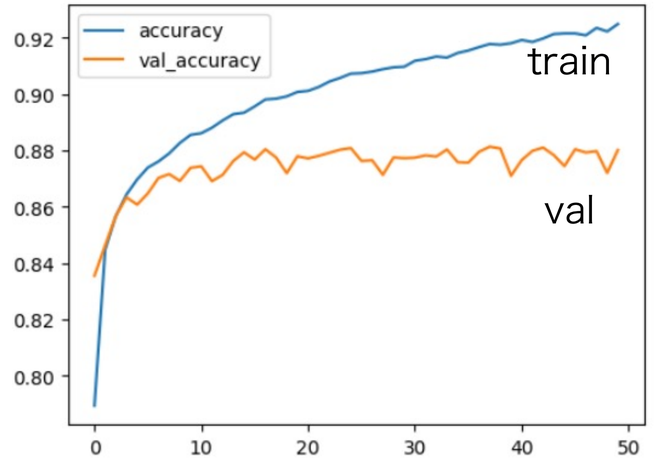
12000枚



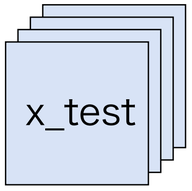
②1epoch終了後、trainとval両方でlossとaccuracyを算出(50epoch)

'val_loss': 0.38
'val_accuracy': 0.88





10000枚

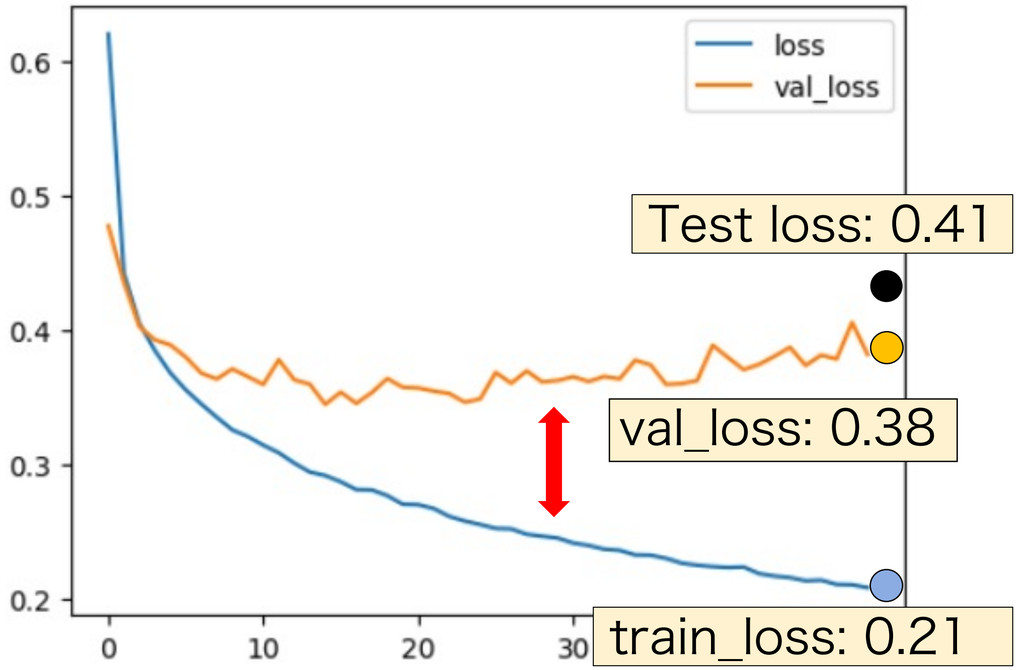


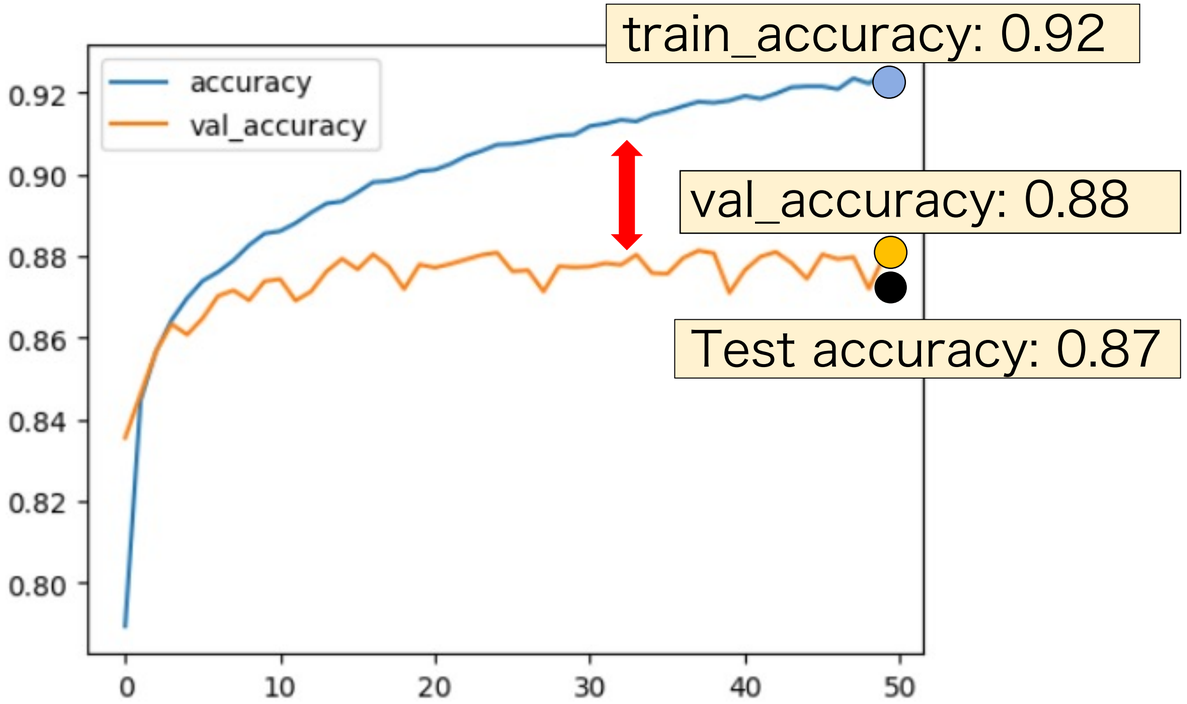
③最後にtestで最終評価
(lossとaccuracy)

Test loss: 0.41
Test accuracy: 0.87

Test loss: 0.40565115213394165
Test accuracy: 0.8694000244140625

trainはlossも小さくaccuracyも高いのに、valは精度が悪い (testも悪い)



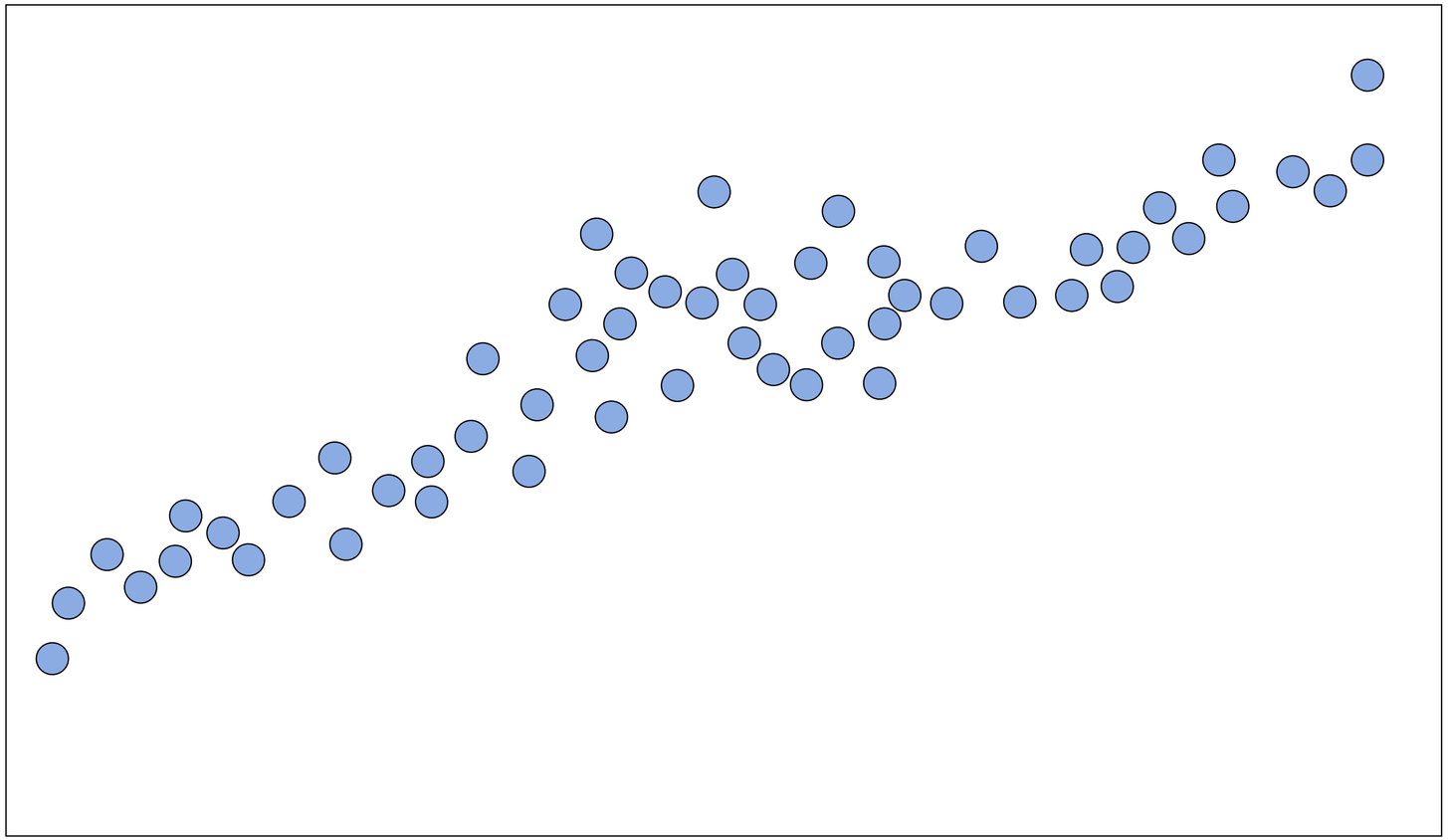


→ trainはうまく学習出来ているのに、valは出来ていない
→ trainのデータに合わせて学習し過ぎ

この状態を過学習と言う

学習のイメージ

学習はデータから当てはまりの良い関数(係数)を導く作業です



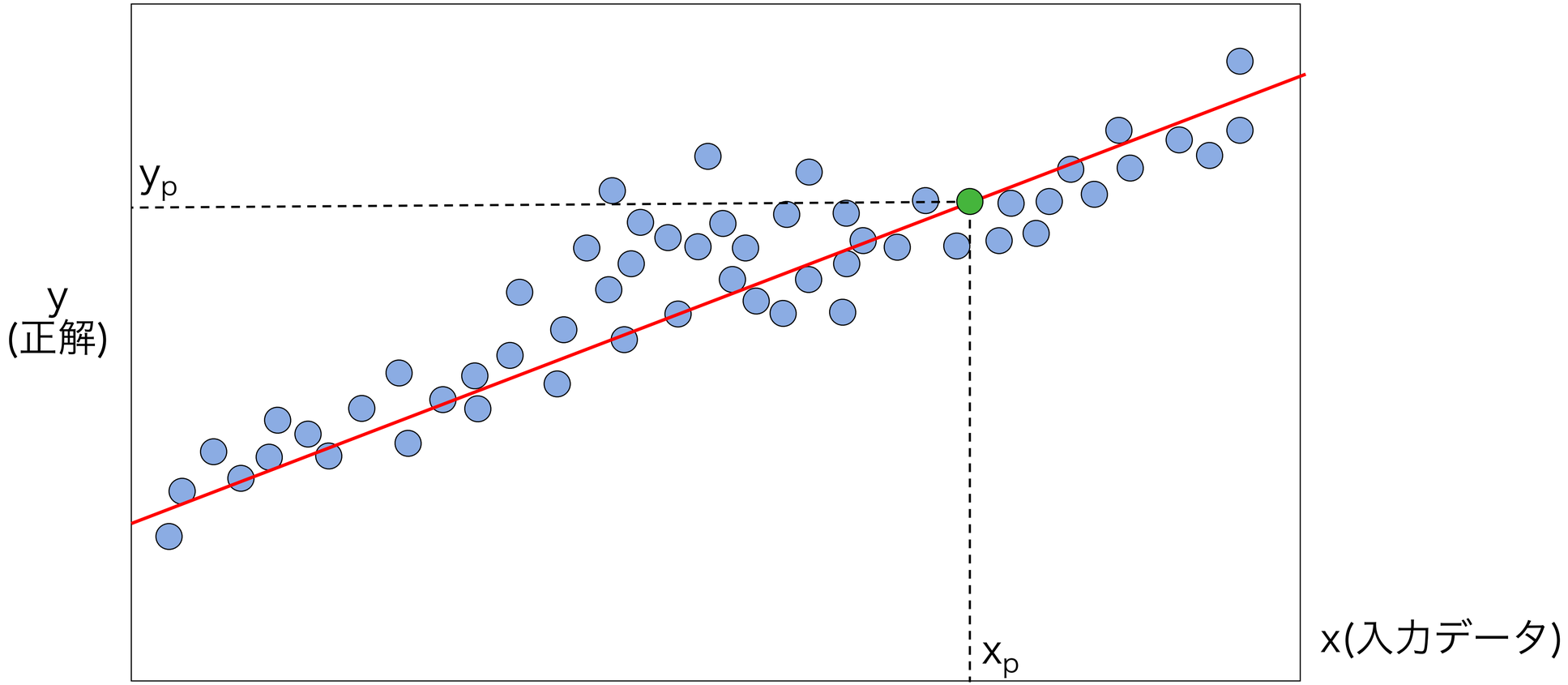
一番シンプルな線形回帰 $f_1(x)$ を想定します(xが増えるとyが一定の割合で増える)

$y = f_1(x) = a_1 x + a_2$　となり、このa_1とa_2を探します

深層学習のwがa_1、bがa_2に相当します

学習のイメージ

学習はデータから当てはまりの良い関数(係数)を導く作業です



a1とa2が求まると、図のような直線が求まるので、

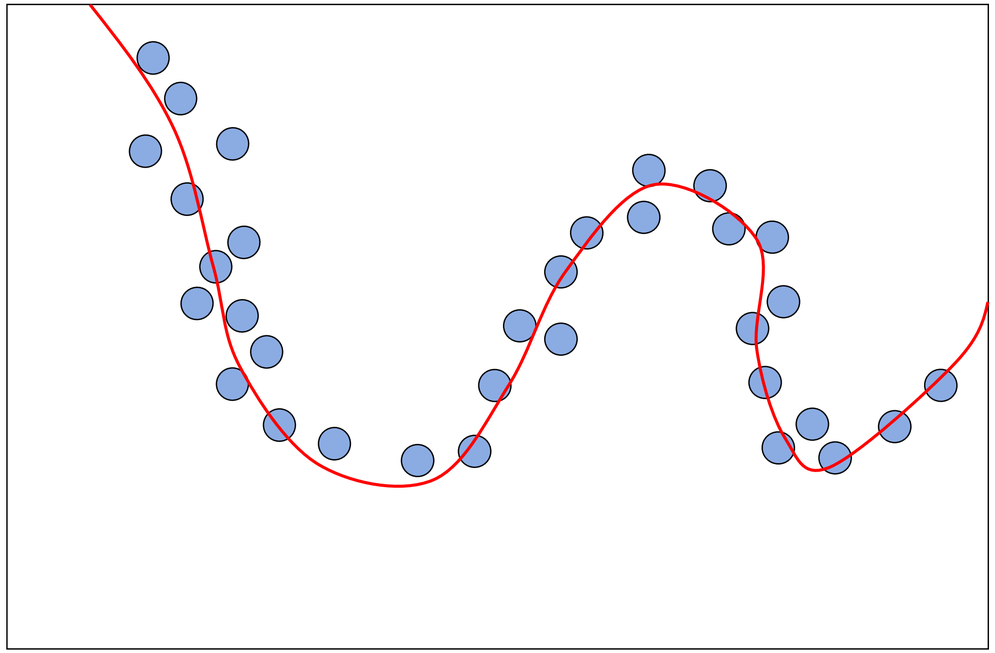
例えば x_pの時の予測結果y_pは、

$y_p = f(x_p) = a_1 x_p + a_2$ で予測することが出来ます

学習のイメージ

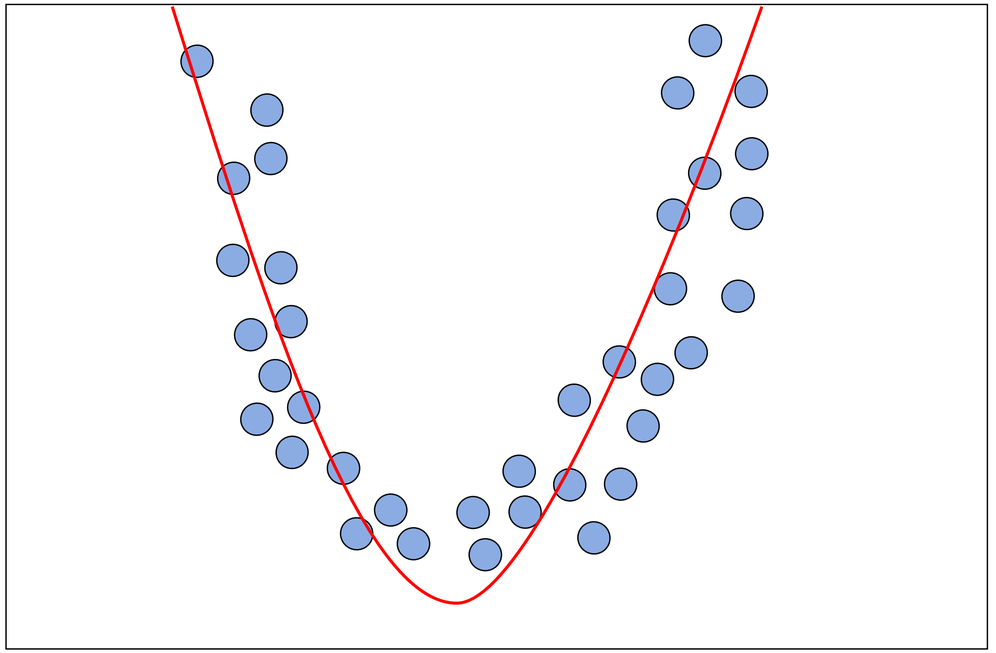
学習はデータから当てはまりの良い関数(係数)を導く作業です

y
(正解)



x(入力データ)

y
(正解)

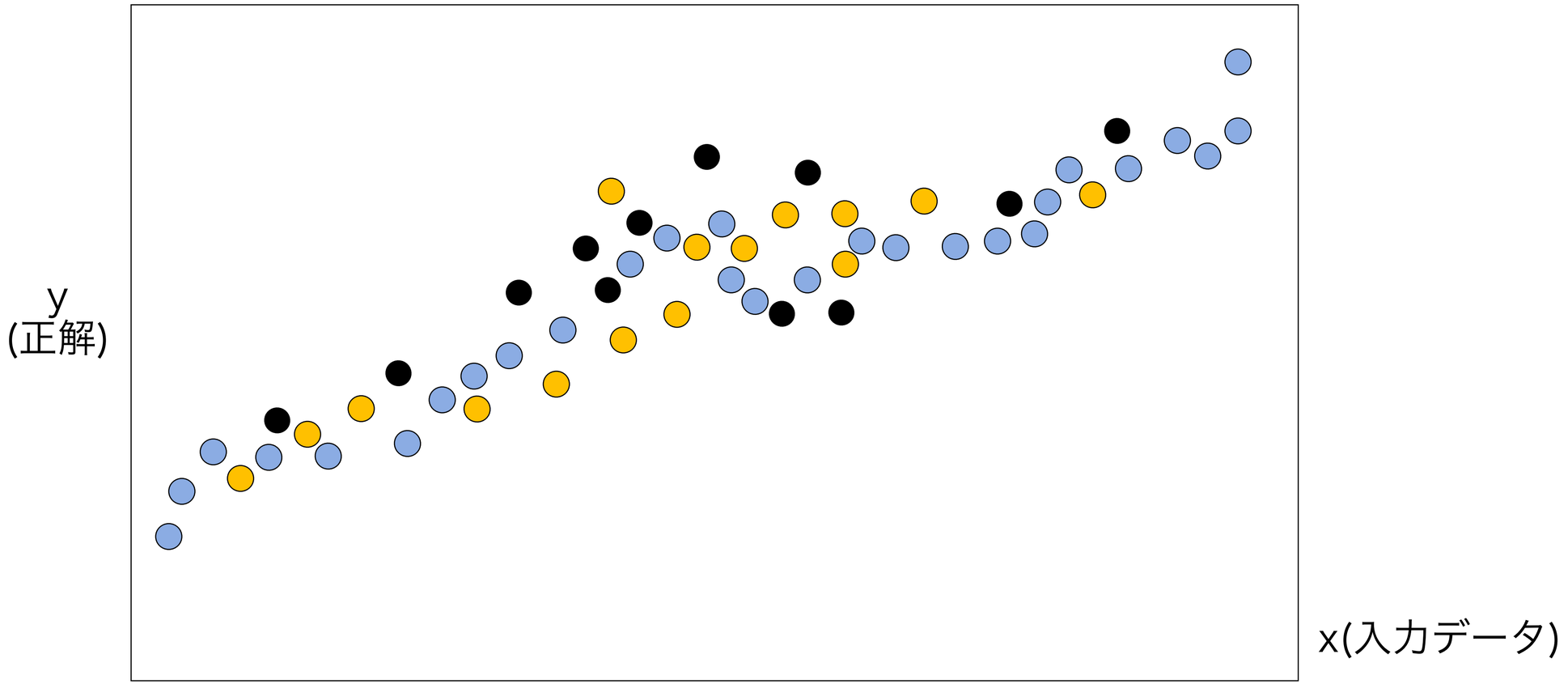


x(入力データ)

$y = a_1x + a_2$では(どんなa_1、a_2を与えても)直線にしかならない

曲線や3次元空間、4次元以上で表現出来るデータも適切な関数を設定すれば学習出来る

過学習のイメージ

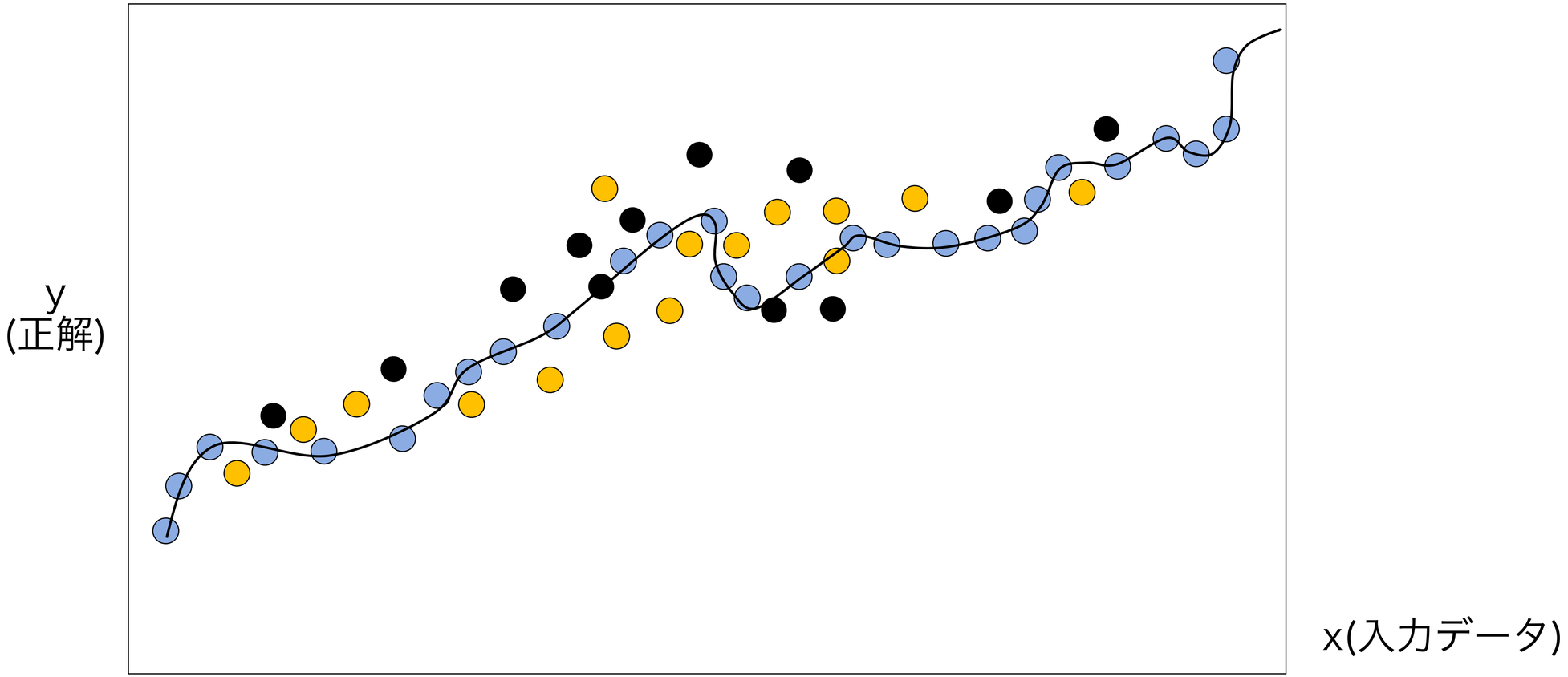


仮に深層学習のデータをまとめて2次元で表したとする
(本当は高次元で作図が難しい)

学習用データ　検証用データ　テスト用データ

過学習のイメージ

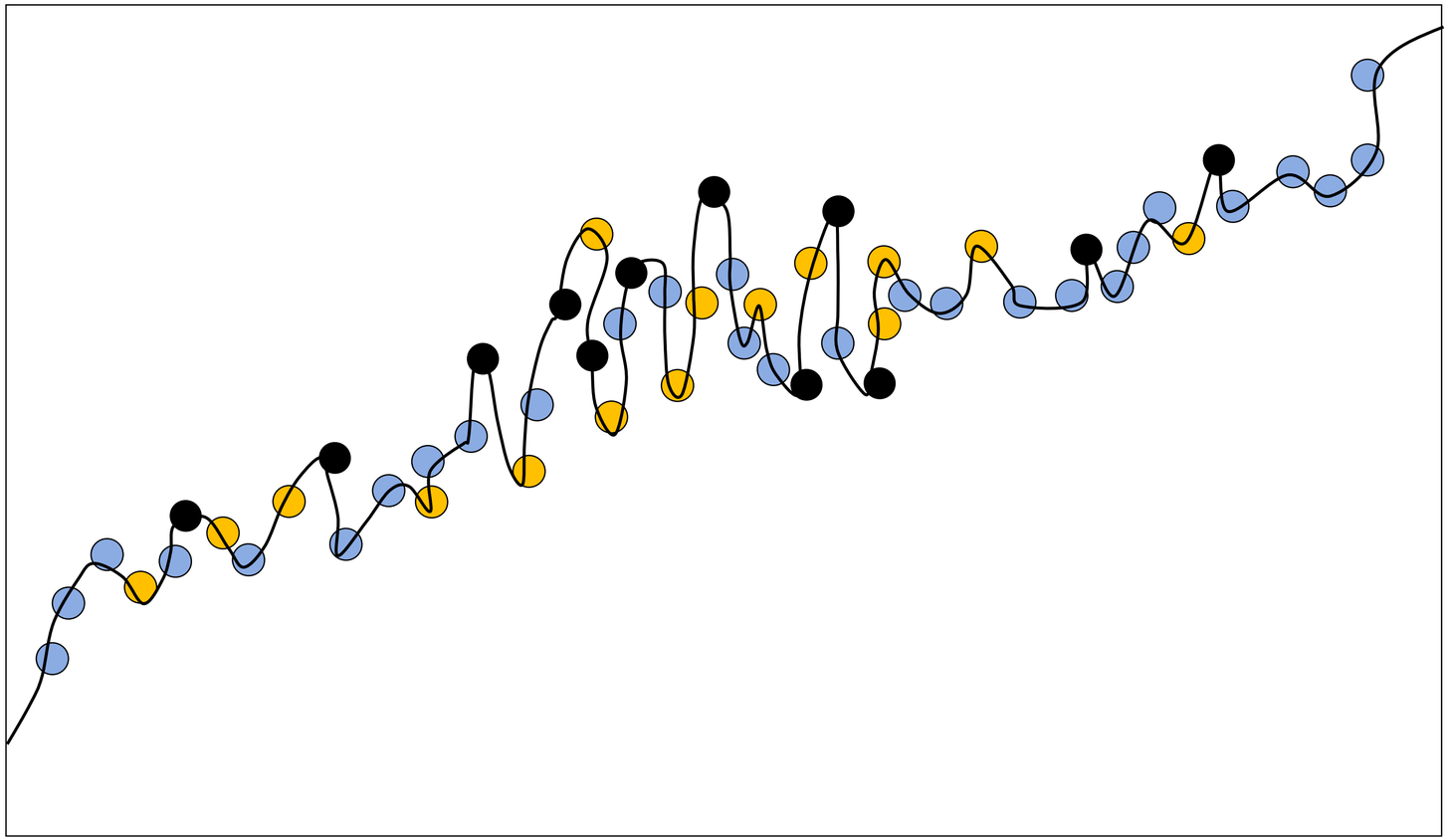
trainはlossも小さくaccuracyも高いのに、valは精度が悪い (testも悪い)



過学習はtrainのデータに合わせて学習し過ぎてしまった状態
この図だとtrainでは精度100%近いが、valやtestでは精度が落ちる

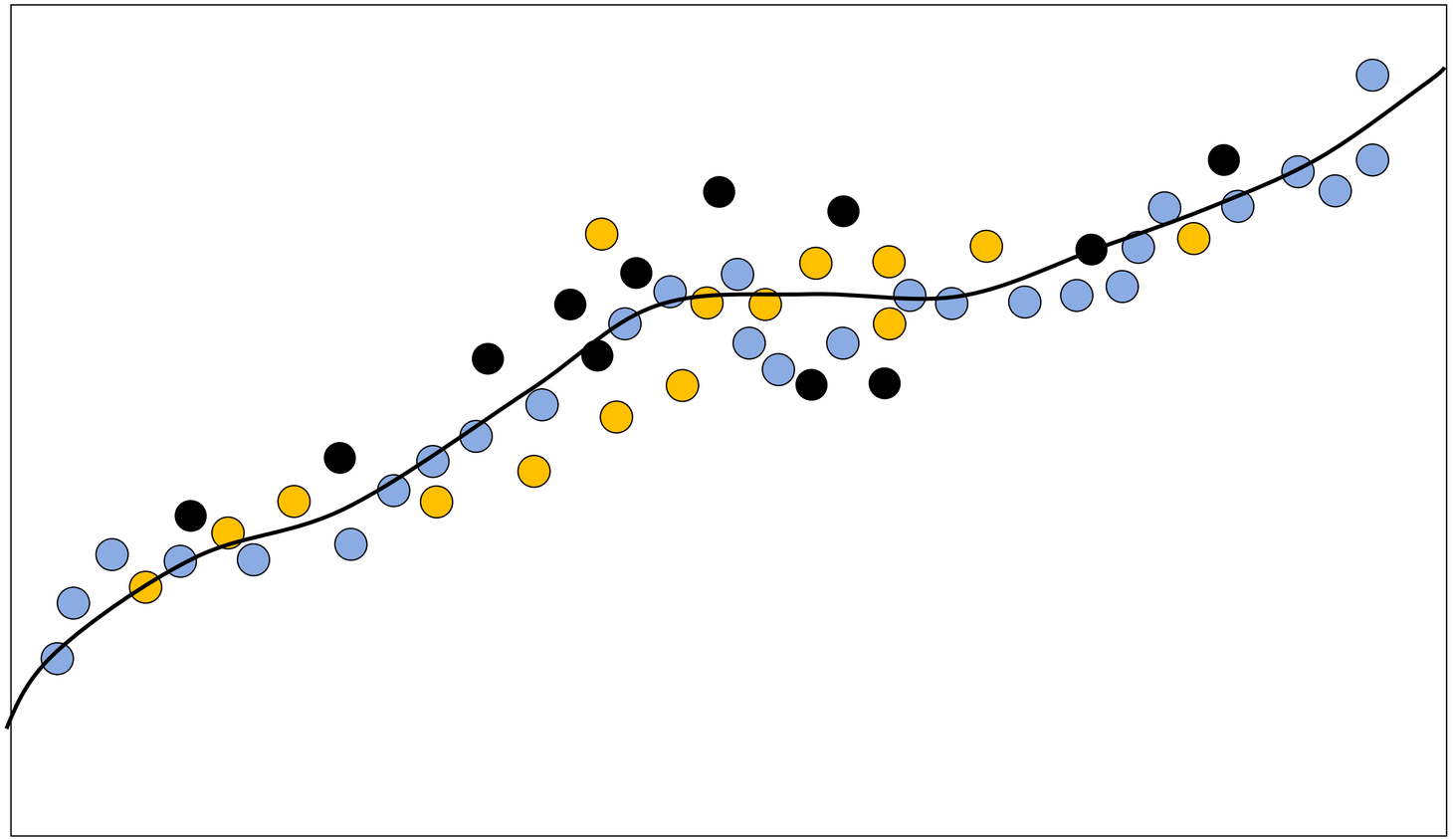
学習用データ　検証用データ　テスト用データ

完全に適合しているイメージ



実際はこのようにはなりません

過学習のイメージ



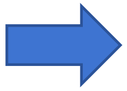
未知のデータでも精度が高くなるようなパラメータを探す作業

モデルの改変や学習用データに偏りがないかなど

ニューロンの数を増やす

```
model = Sequential()
model.add(Dense(32,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 32)                25120
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```



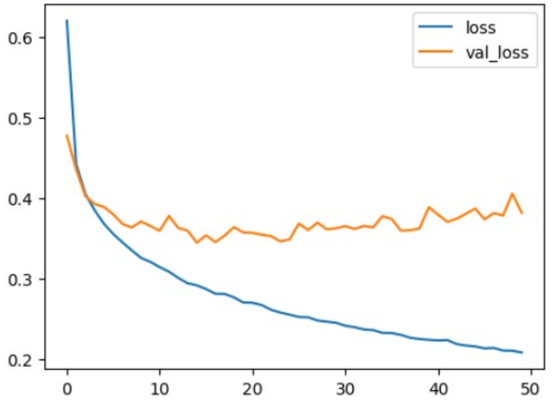
```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

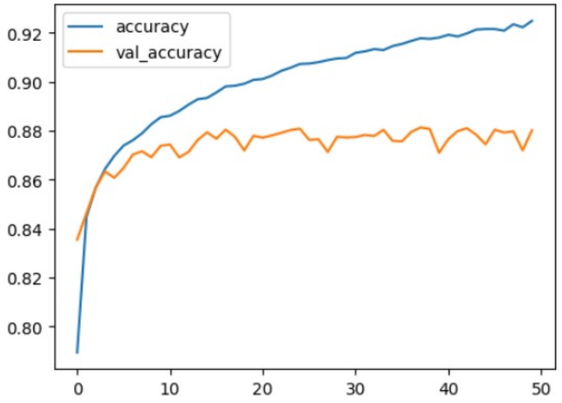
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

ニューロンの数を増やす

```
model = Sequential()
model.add(Dense(32,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy',
                    optimizer='Adam',metrics=['accuracy'])
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 32)                25120
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

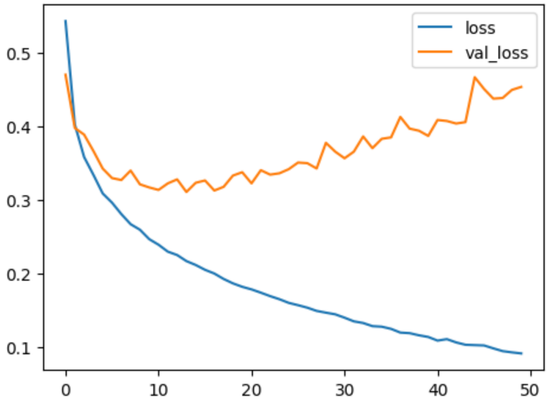


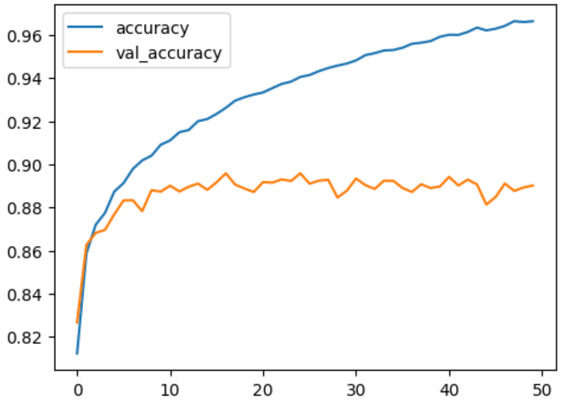


Test loss: 0.40565115213394165
Test accuracy: 0.8694000244140625

```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy',
                    optimizer='Adam',metrics=['accuracy'])
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```





Test loss: 0.4939178228378296
Test accuracy: 0.8852000236511123

層を追加してみよう

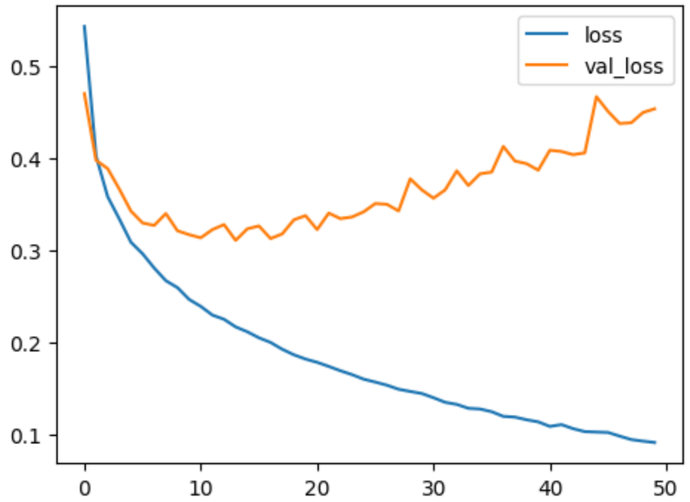
```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

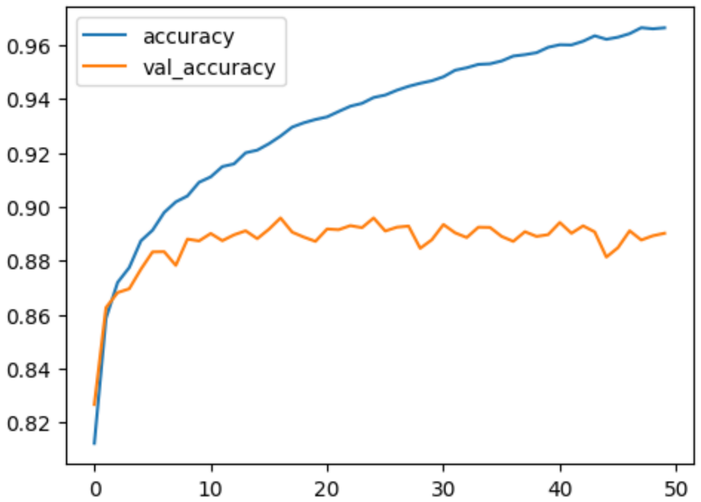
```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu’))
model.add(Dense(64,activation='relu’))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy’,
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

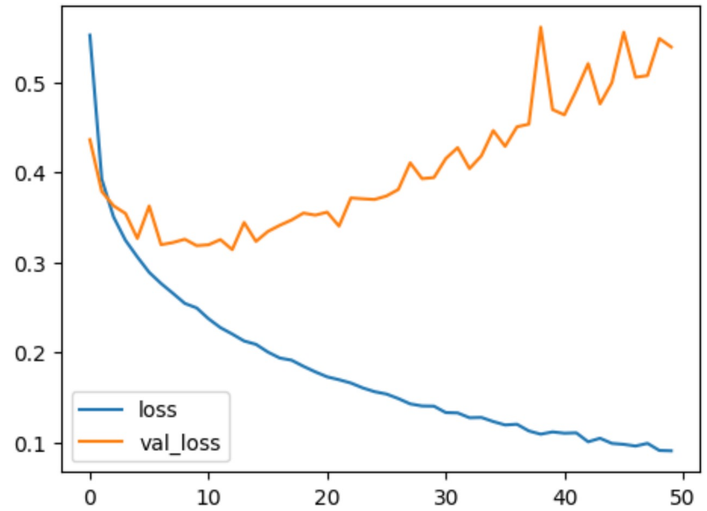
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_5 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_6 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
_________________________________________________________________
```

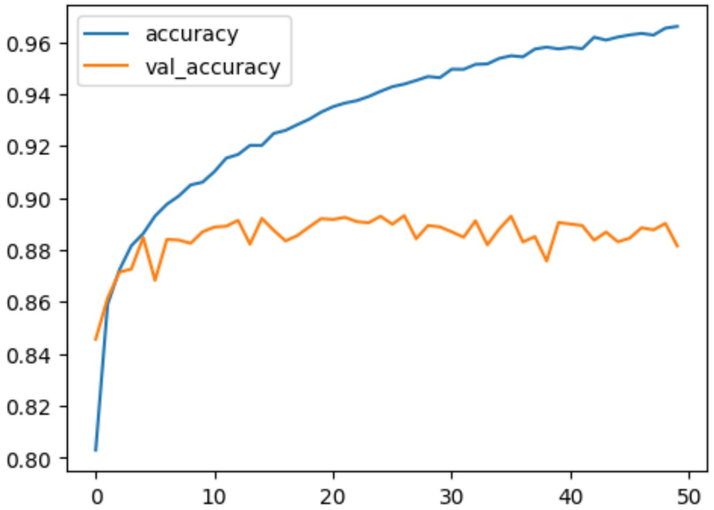
層を増やしても今回のデータでは精度あまり上がっていない





Test loss: 0.4939178228378296
Test accuracy: 0.885200023651123





Test loss: 0.5994181036949158
Test accuracy 0.877399810218811

なぜ過学習が起きるのか

モデルの複雑さ： モデルが非常に複雑である場合（例えば、パラメータが多すぎる）、そのモデルは訓練データのノイズまで学習してしまう。モデルがデータの真のパターンよりも、データに含まれるランダムな誤差や無関係な特徴を学習してしまう。

データの不足： 訓練データが不十分である場合、モデルは利用可能なデータに過剰に適合してしまう。十分なバリエーションのデータがなければ、モデルが一般化するための「良い」パターンを学ぶことができない。

トレーニングの長さ： トレーニングを長く続けすぎると、モデルが訓練データセットの特異性を学習し、新しいデータに対してうまく一般化できなくなる。

データの品質： データに偏りがあったり、誤った情報が含まれていたりすると、モデルが誤ったパターンを学習する原因となる。

Dropoutを加えてみよう

```
model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dense(64,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy',
                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

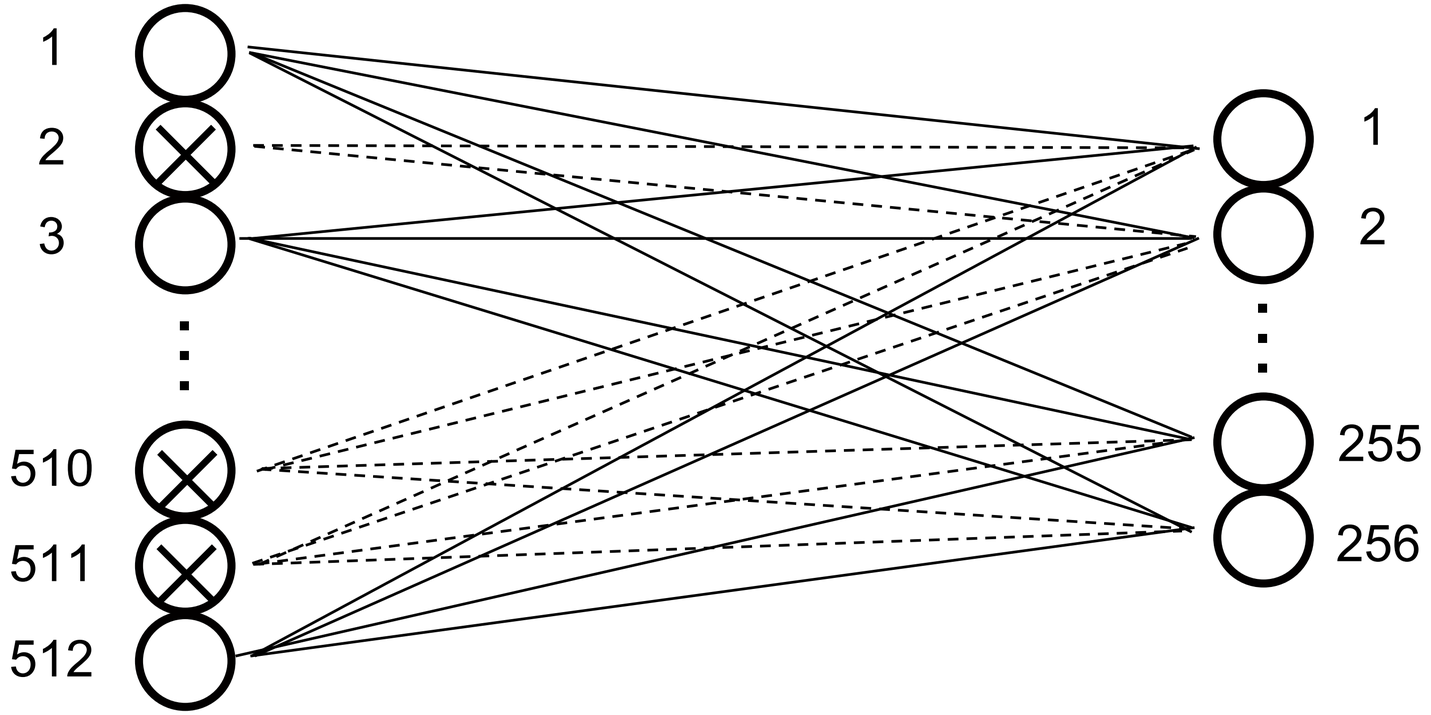
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_5 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_6 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
_________________________________________________________________
```

```
from keras.layers import Dropout

model = Sequential()
model.add(Dense(128,input_shape=(784,),activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(loss='categorical_crossentropy',

                        optimizer='Adam',metrics=['accuracy'])
model.summary()
```

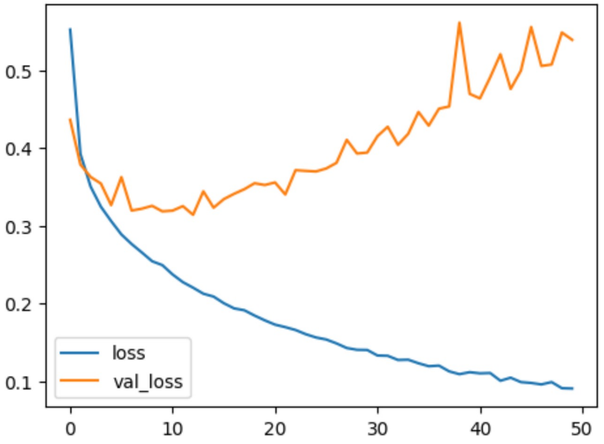
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_10 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_11 (Dense)             (None, 10)                330
=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
_________________________________________________________________
```

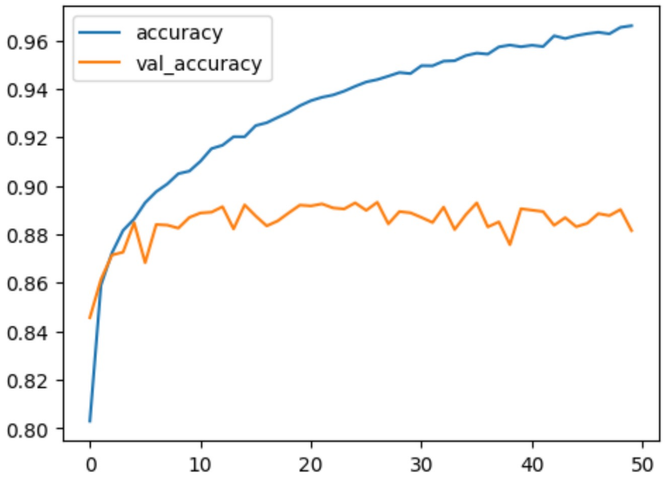
Dropout

過学習を防ぐための対策の1つで、学習時に設定した確率で出力を０にする手法。
推論時には何も行わず、学習時にのみ行われる。これにより、特定のニューロン
の評価だけに依存し過ぎることを避けて、より頑健な(=データの本質的な構造を捉えた)
特徴を学習することを促す。

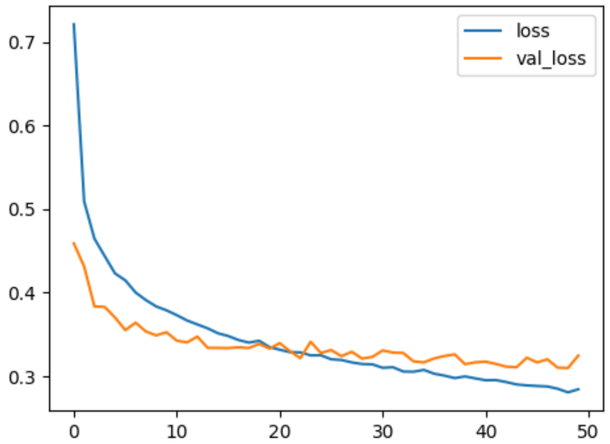


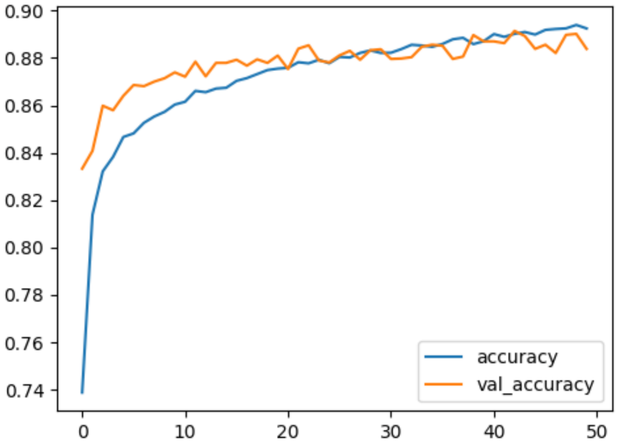
Dropoutを加えると過学習を抑制できる





Test loss: 0.5994181036949158
Test accuracy 0.8773999810218811





Test loss: 0.3330557644367218
Test accuracy: 0.8855000138282776

実際は過学習を抑えつつ精度をどれだけ上げられるかを検討する

層はいくらでも増やすことが出来ます。

色々試してみましょう。

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 256)	200960
dropout_1 (Dropout)	(None, 256)	0
dense_13 (Dense)	(None, 64)	16448
dense_14 (Dense)	(None, 128)	8320
dropout_2 (Dropout)	(None, 128)	0
dense_15 (Dense)	(None, 64)	8256
dense_16 (Dense)	(None, 32)	2080
dense_17 (Dense)	(None, 10)	330

Total params: 236,394
Trainable params: 236,394
Non-trainable params: 0

Test loss:
0.34160017967224121
Test accuracy:
0.88550001382827759

今回のデータの量、質だと
MLPではパラメータを増やし
ても90%以上の精度は
なかなか出にくいようです。

課題

・WebClassにある”kadai5.ipynb”をやってみましょう
・実行したら”学籍番号_名前_5.ipynb”という名前で保存して提出して下さい。

締め切りは2週間後の11/23の23:59です。
締め切りを過ぎた課題は受け取らないので注意して下さい