医療とAI・ビッグデータ応用

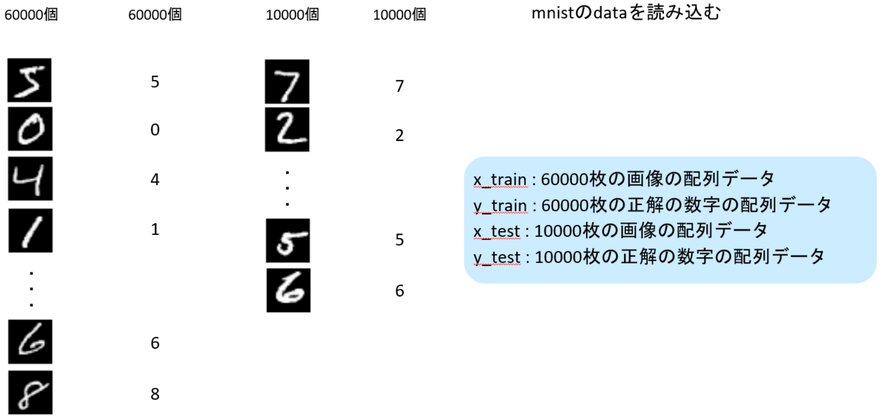
MLP①

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)
```

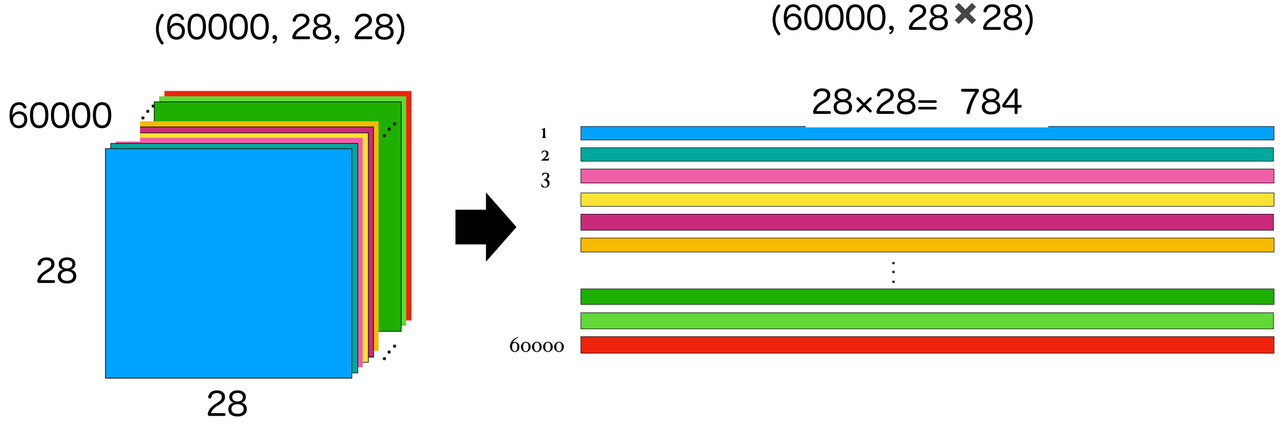
```
(60000, 784)
(10000, 784)
```

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

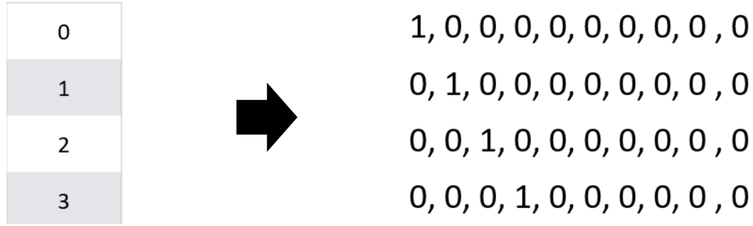
```
(60000, 10)
(10000, 10)
```

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



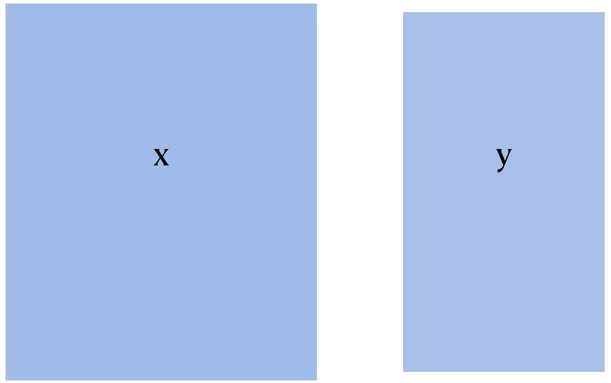
y_train, y_test : one-hot encodingで01のみの配列にする
y_train = to_categorical(y_train)



深層学習(教師あり機械学習)の復習

データを用意する

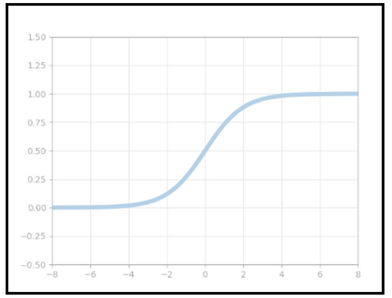
x(特徴量データ)　　y(正解データ)



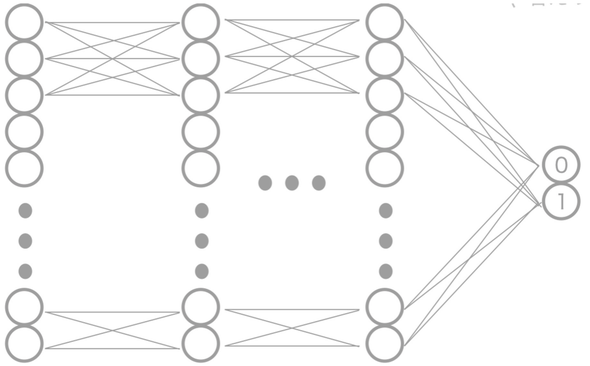
データを配列に整える

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

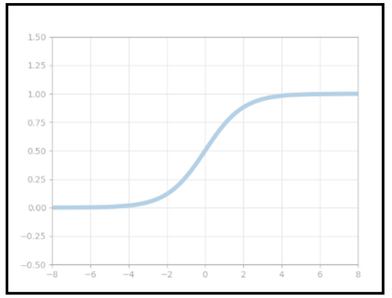
y(正解データ)

y

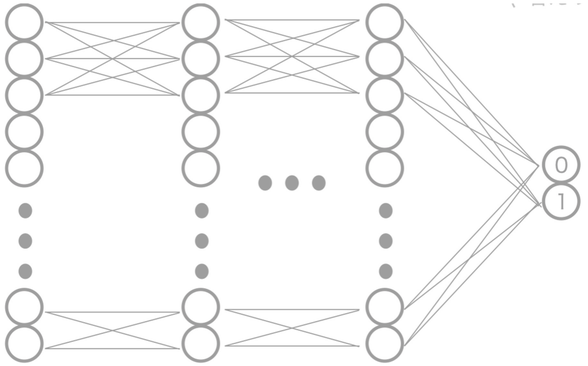
データを配列に整える

学習させる

ロジスティック回帰分析

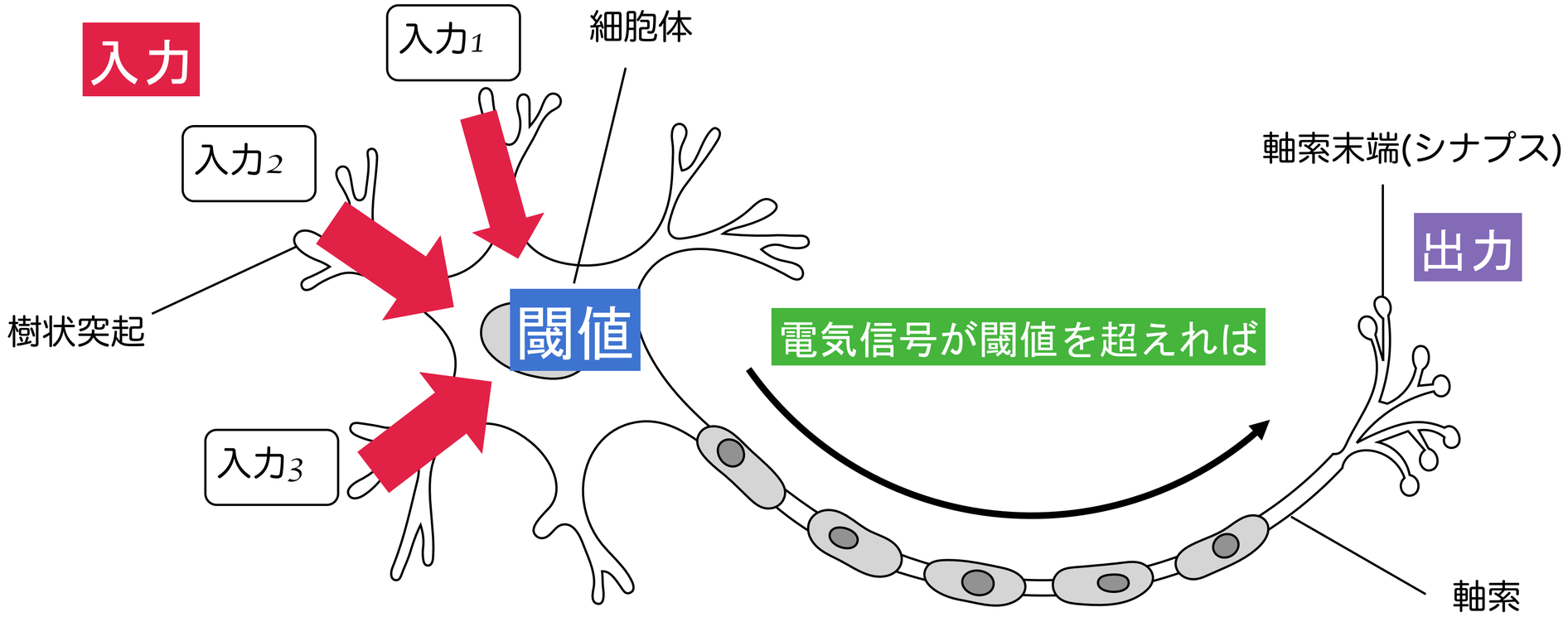


ニューラルネットワーク



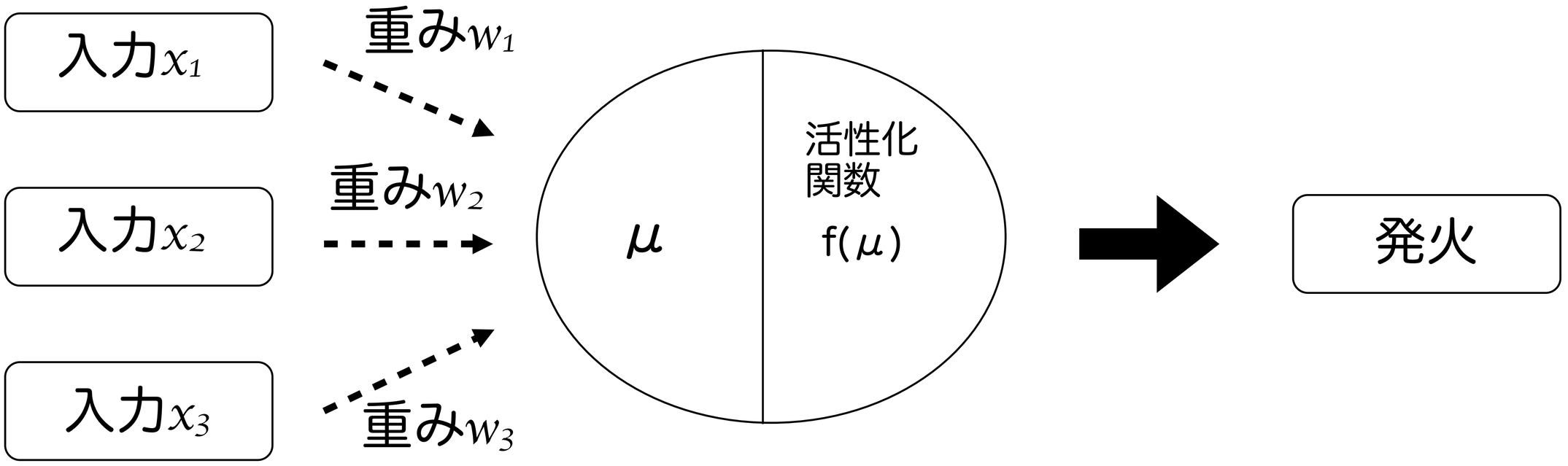
ニューロンとパーセプトロン

・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

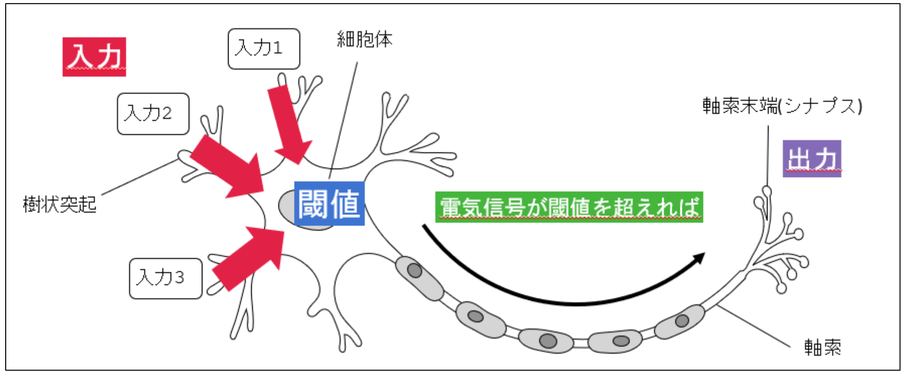
単一の人工ニューロンはこのようなモデルで表すことができる。

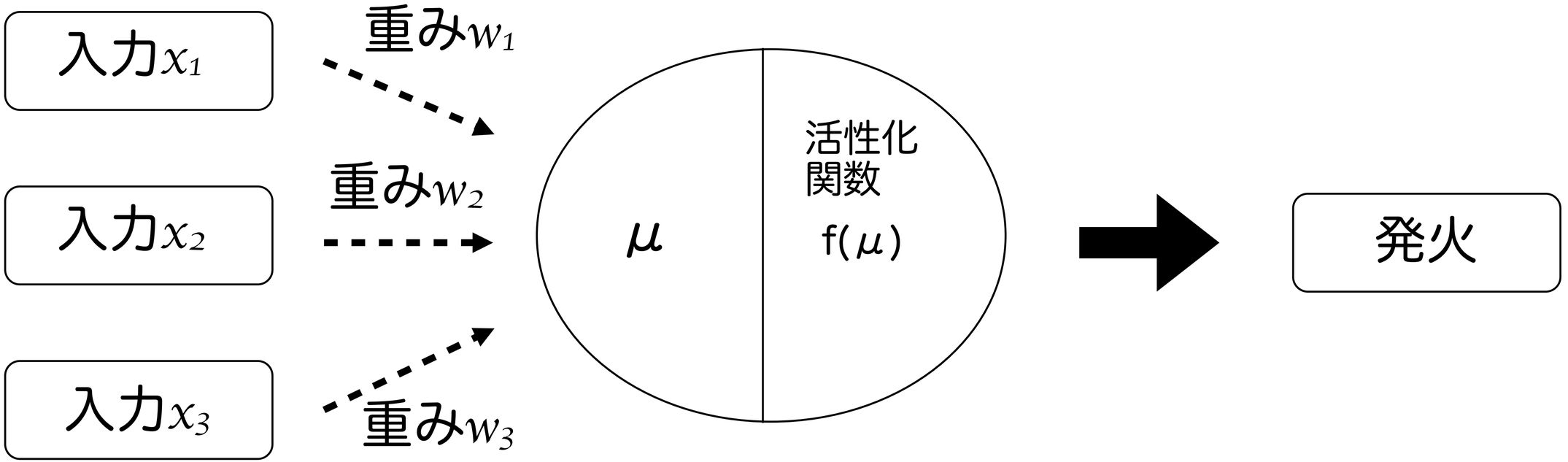


ニューロンとパーセプトロン

(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



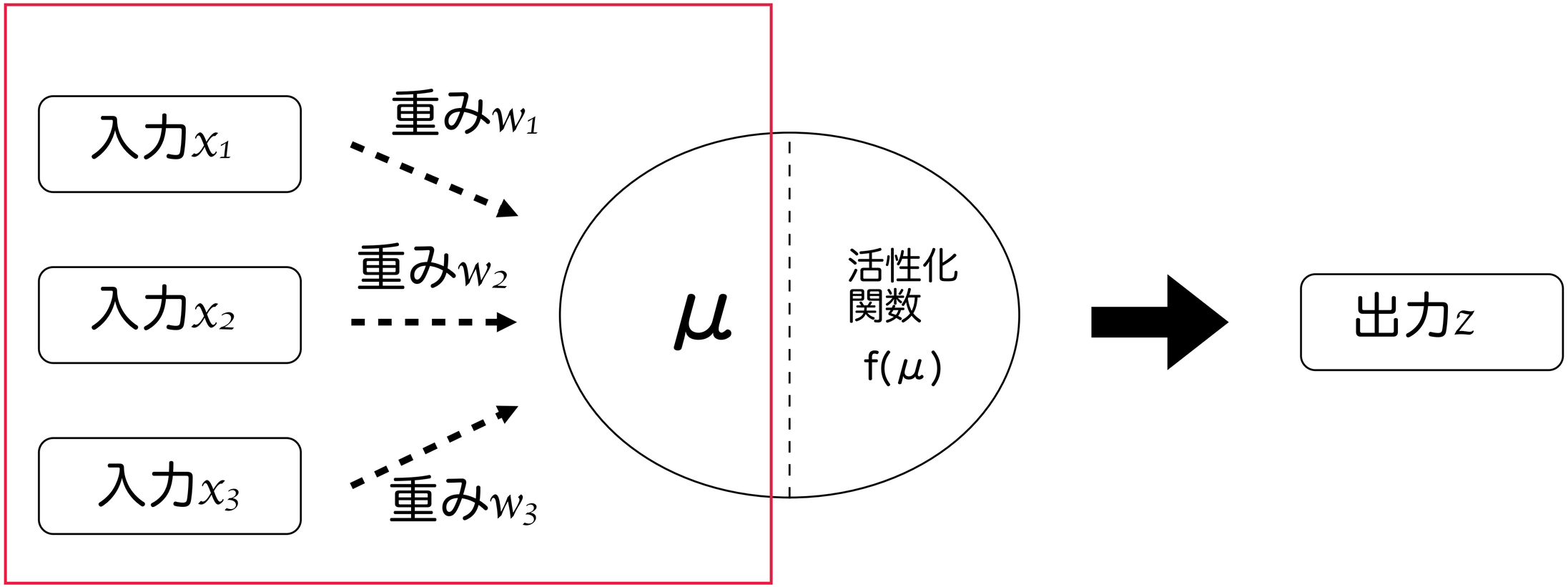


樹状突起　　細胞体　　軸索

ニューロンとパーセプトロン

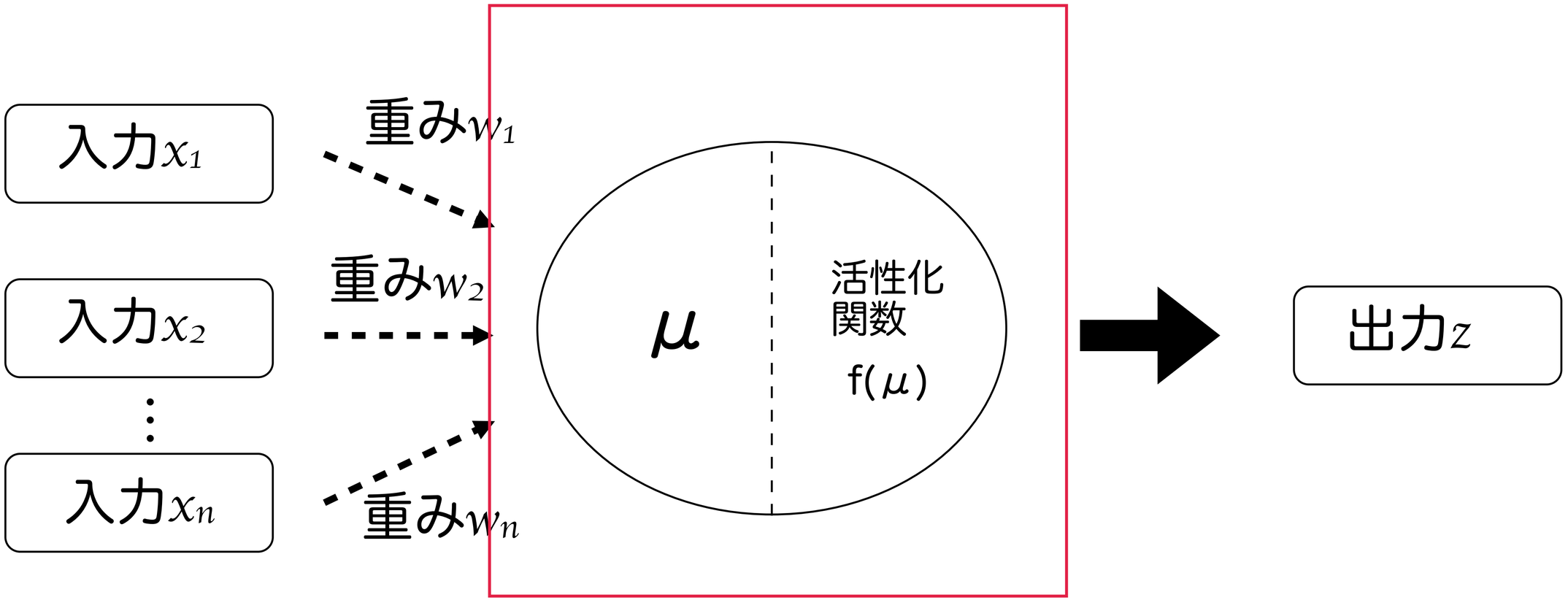
μは入力値に重みを掛け合わせた合計で計算される

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



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

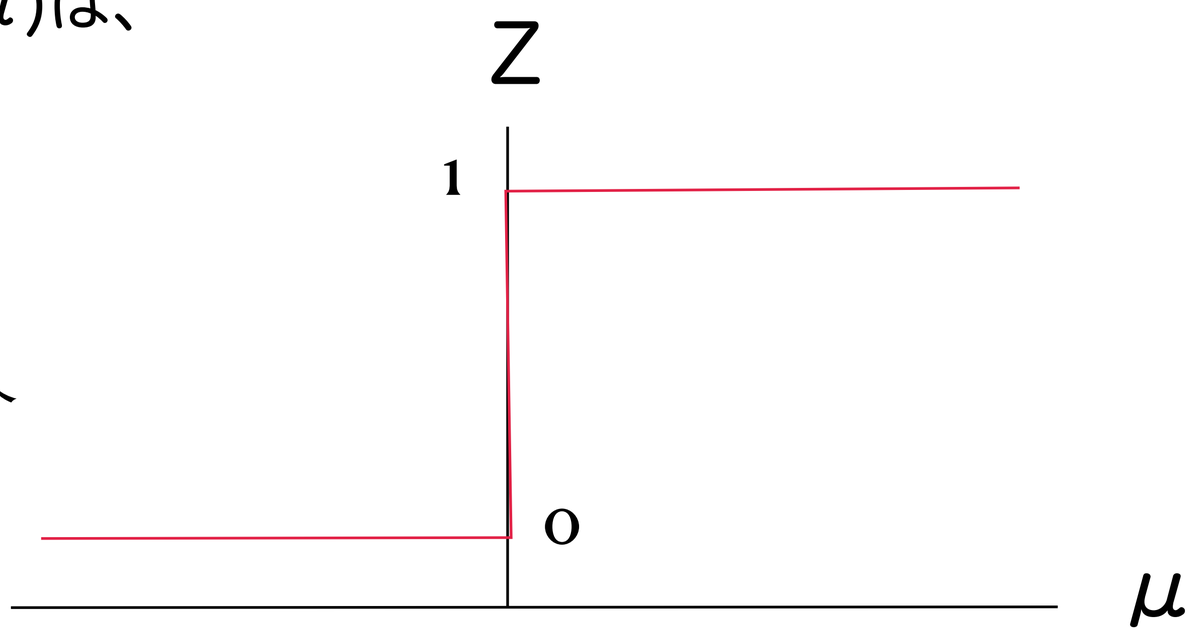
- ステップ関数
- 恒等関数
- シグモイド関数
- tanh関数
- ReLU関数
- ソフトプラス関数
- Leaky ReLU
- ソフトマックス関数
- PReLU / Parametric ReLU
- ELU
- SELU
- Swish関数
- Mish関数

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、

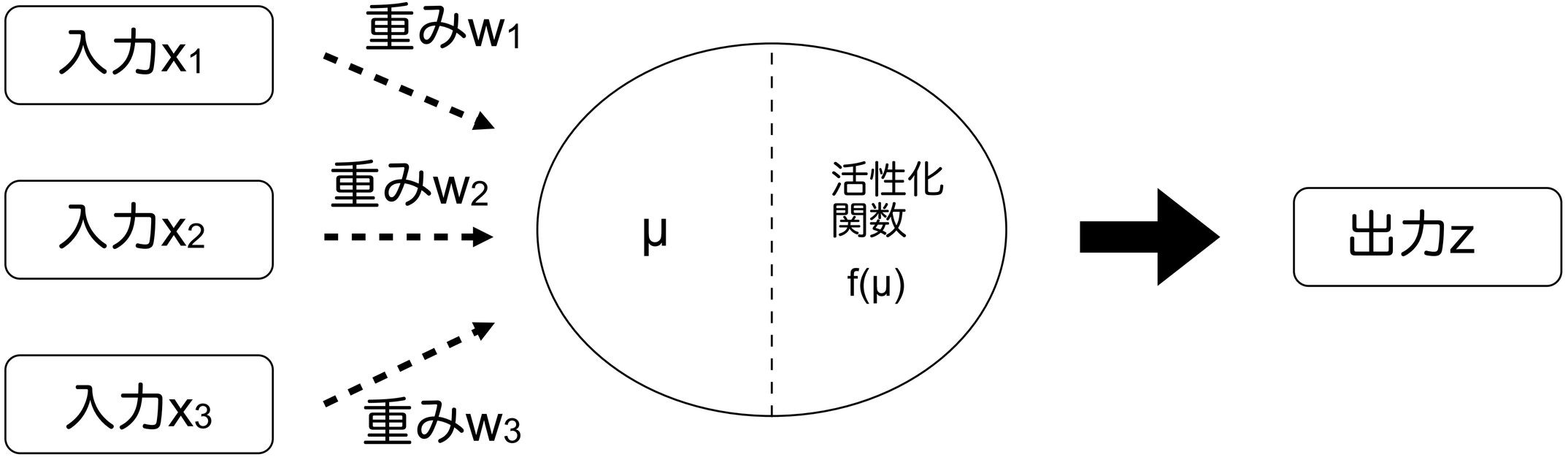
$$Z = f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



μがどんな値でも出力は
0か1のいずれかになる！

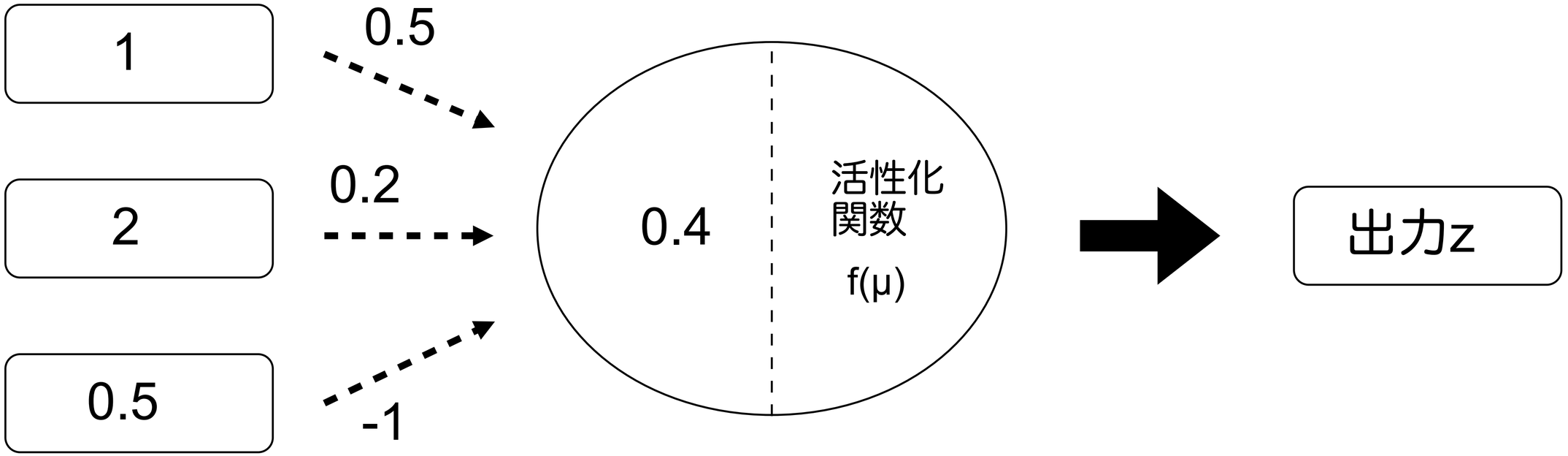
閾値を0とすると、μが0より大きければzは1となり(発火)、
0以下であれば0となる(発火しない)

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？

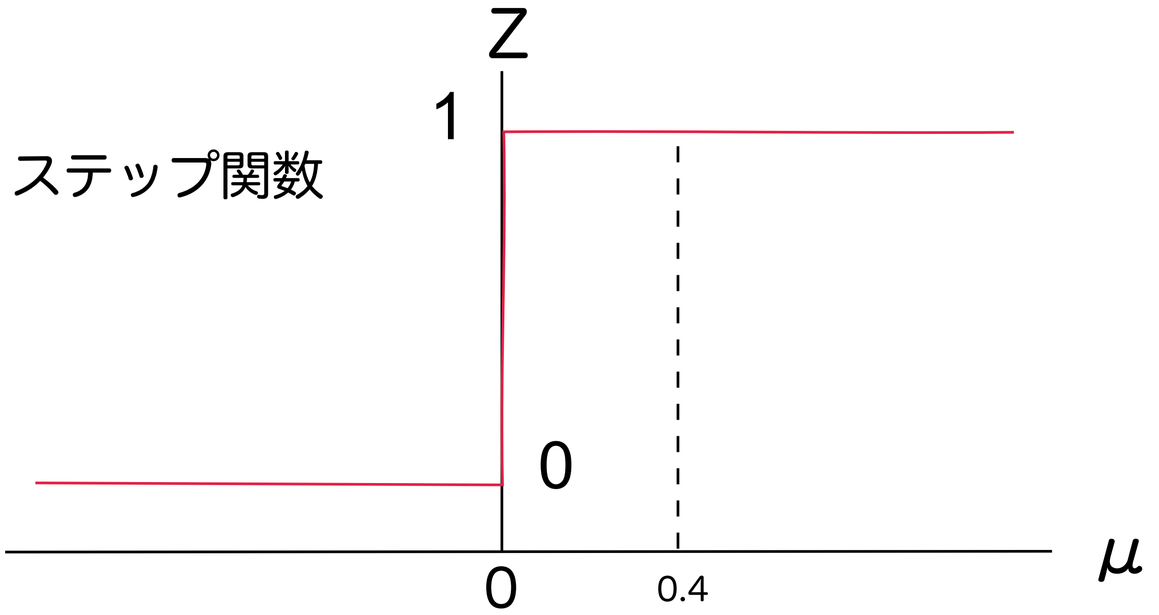


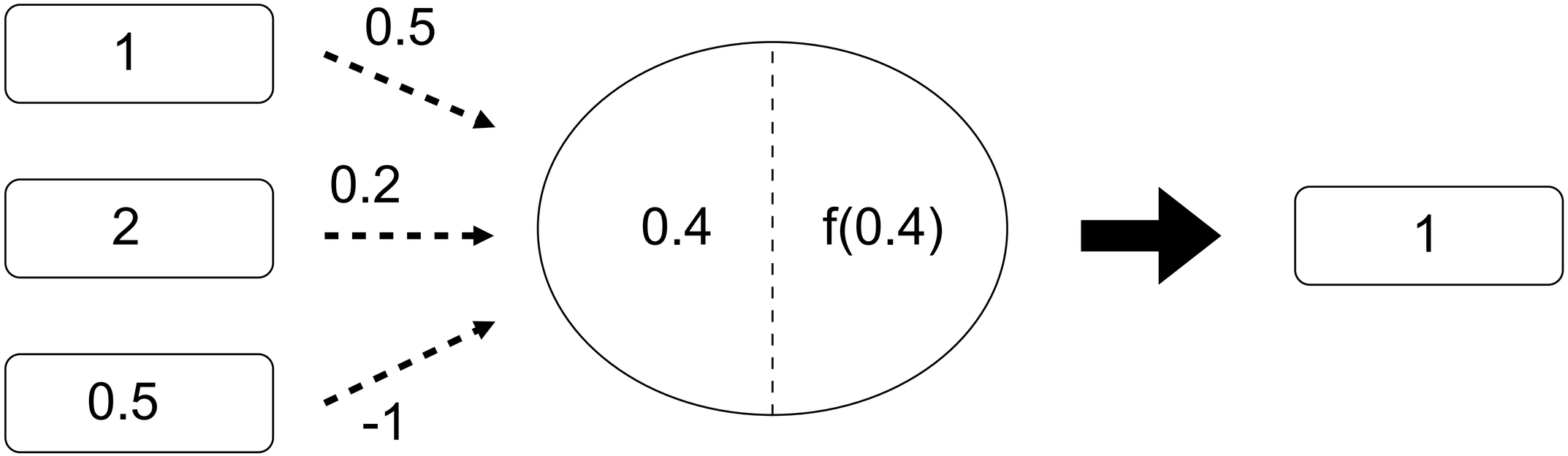
例えばx_1=1、x_2=2、x_3=0.5、w_1=0.5、w_2=0.2、w_3=-1の時は？

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

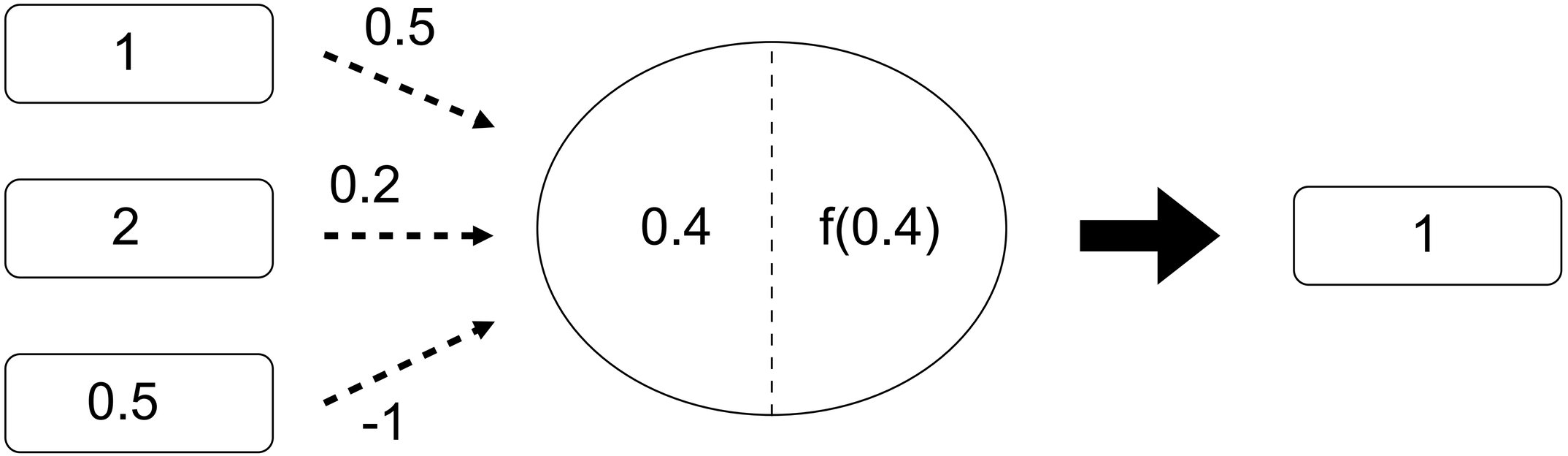


$$z = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



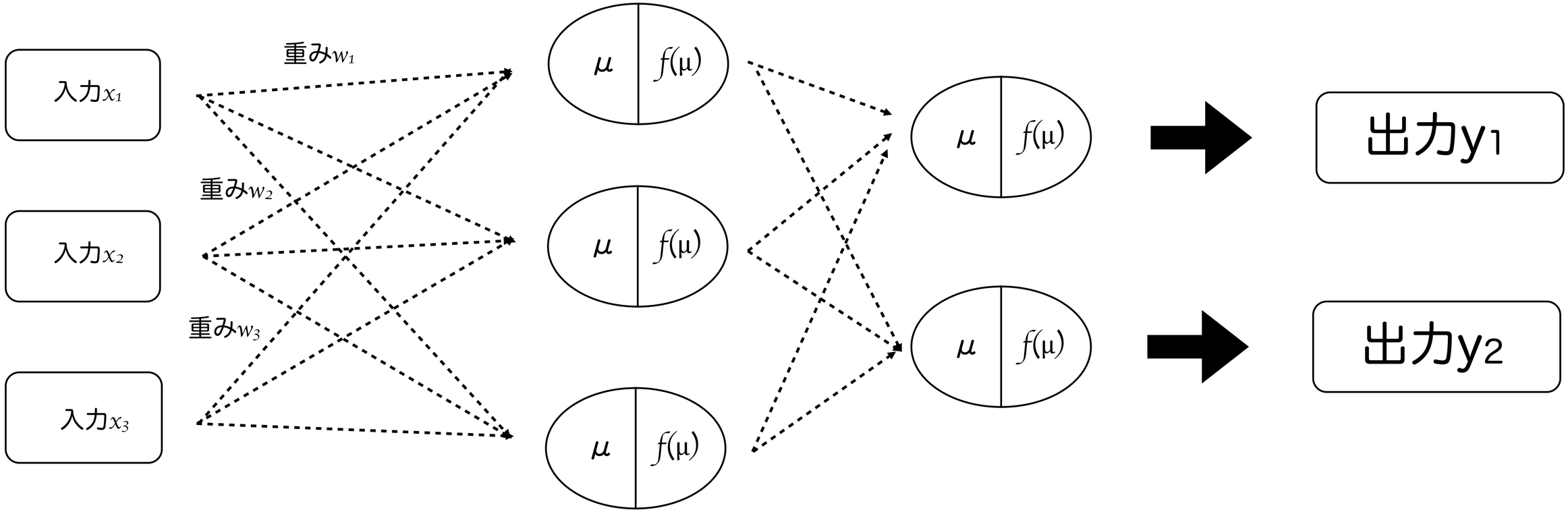


このような重みw_iや閾値を調整することできる
人工ニューラルネットワークで学習する仕組みをパーセプトロンと呼ぶ。

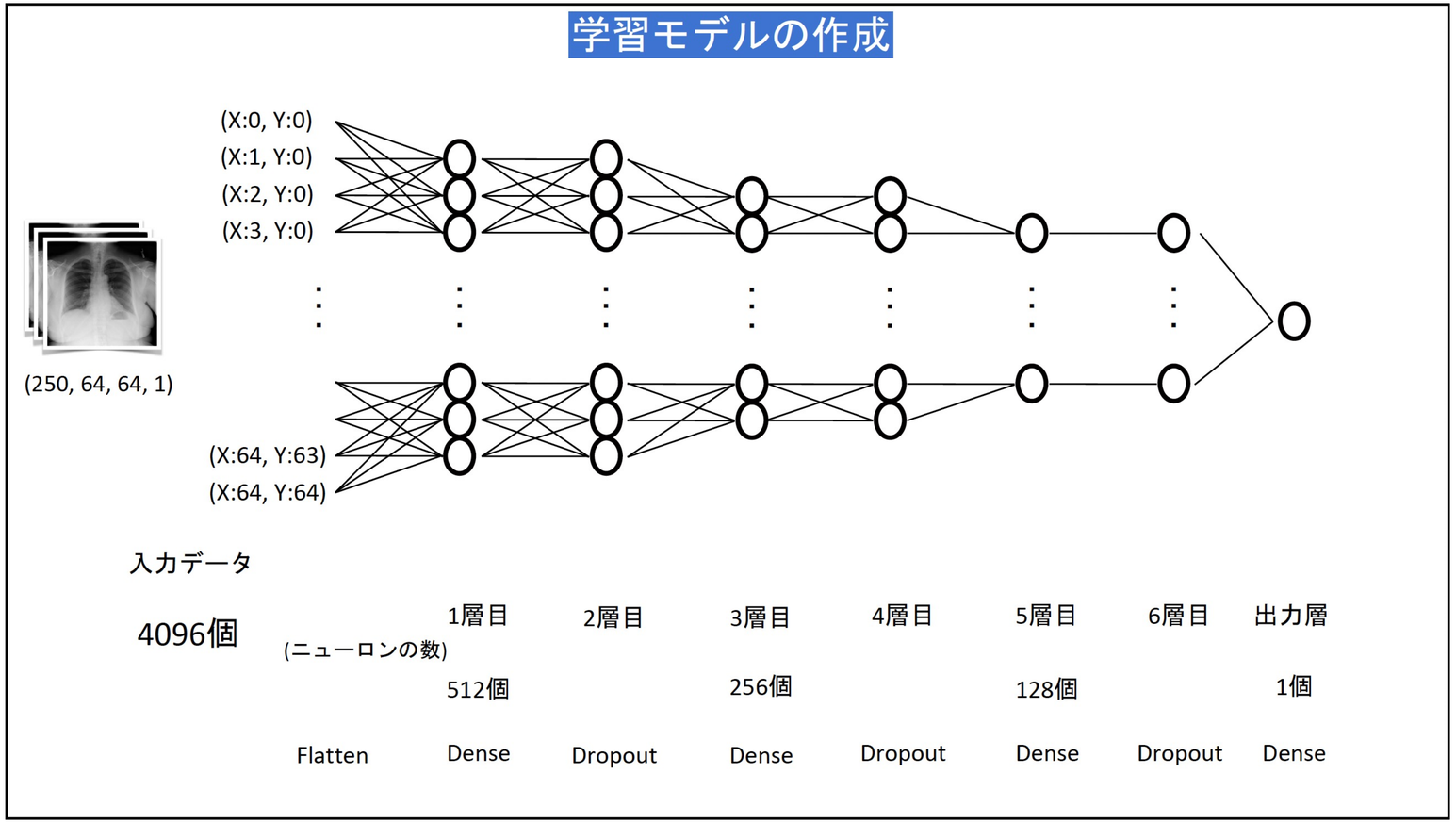


多層パーセプトロン（MLP：Multi Layer Perceptron）

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



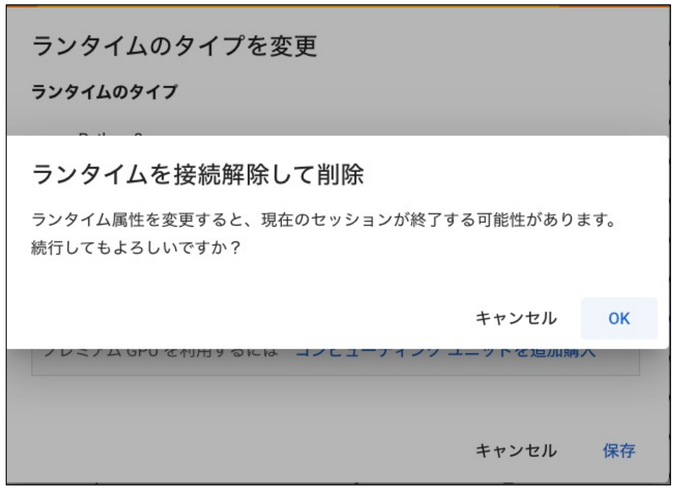
これは実はMLP(Multi Layer Perceptron)

webclassの20240509用事前資料.ipynbをダウンロードしてアップロードしましょう.

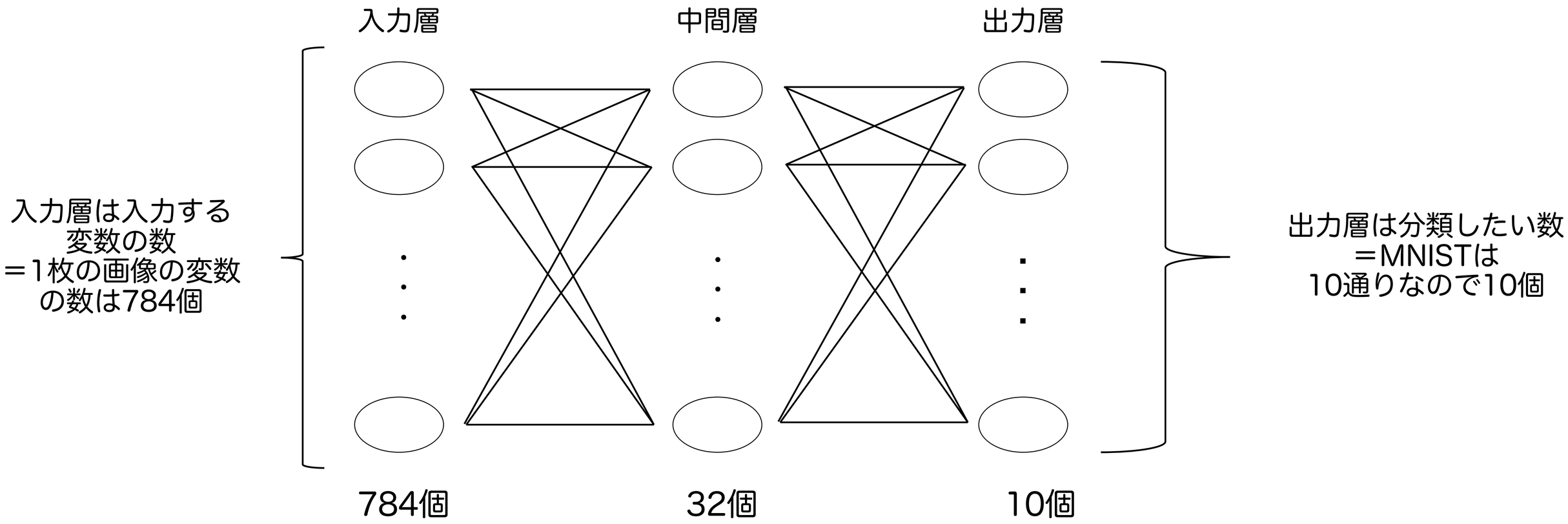
```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

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)
```

前回のファイルを開いてもらっても大丈夫です

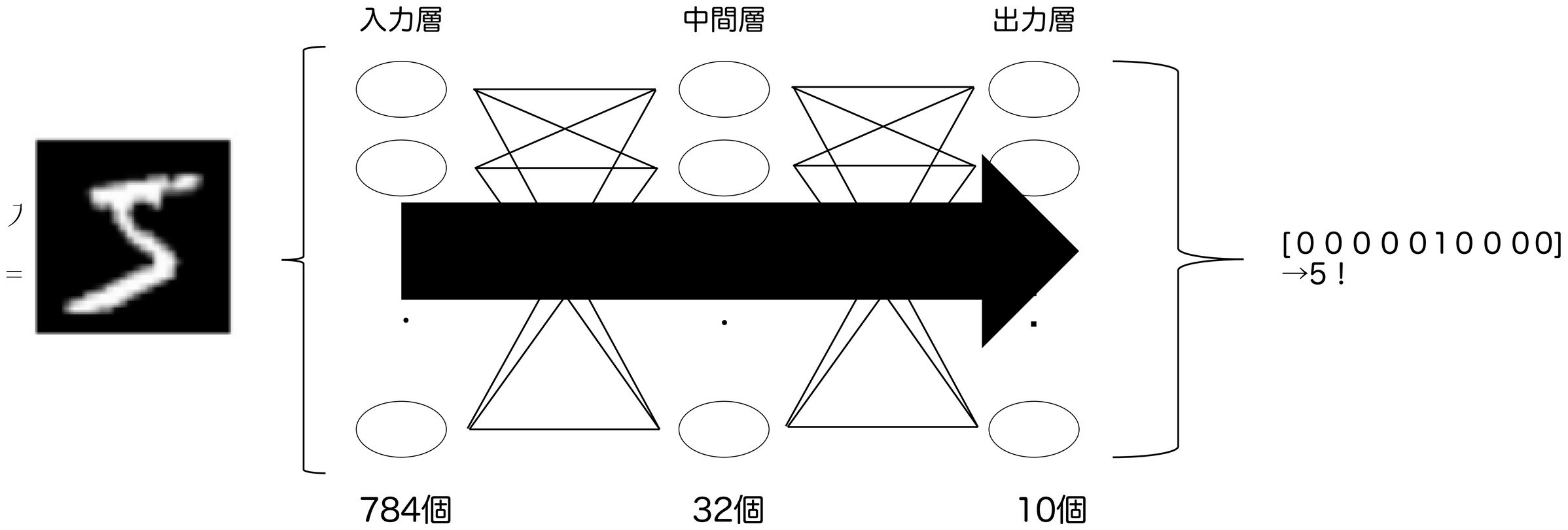
実行前に今日からGPUに切り替えて実行します
「ランタイム」→「ランタイムのタイプを変更」
ハードウェアタイプをCPUからT4 GPUに変更



```
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()
```

784個

0.53

0.24

0.88

0.34

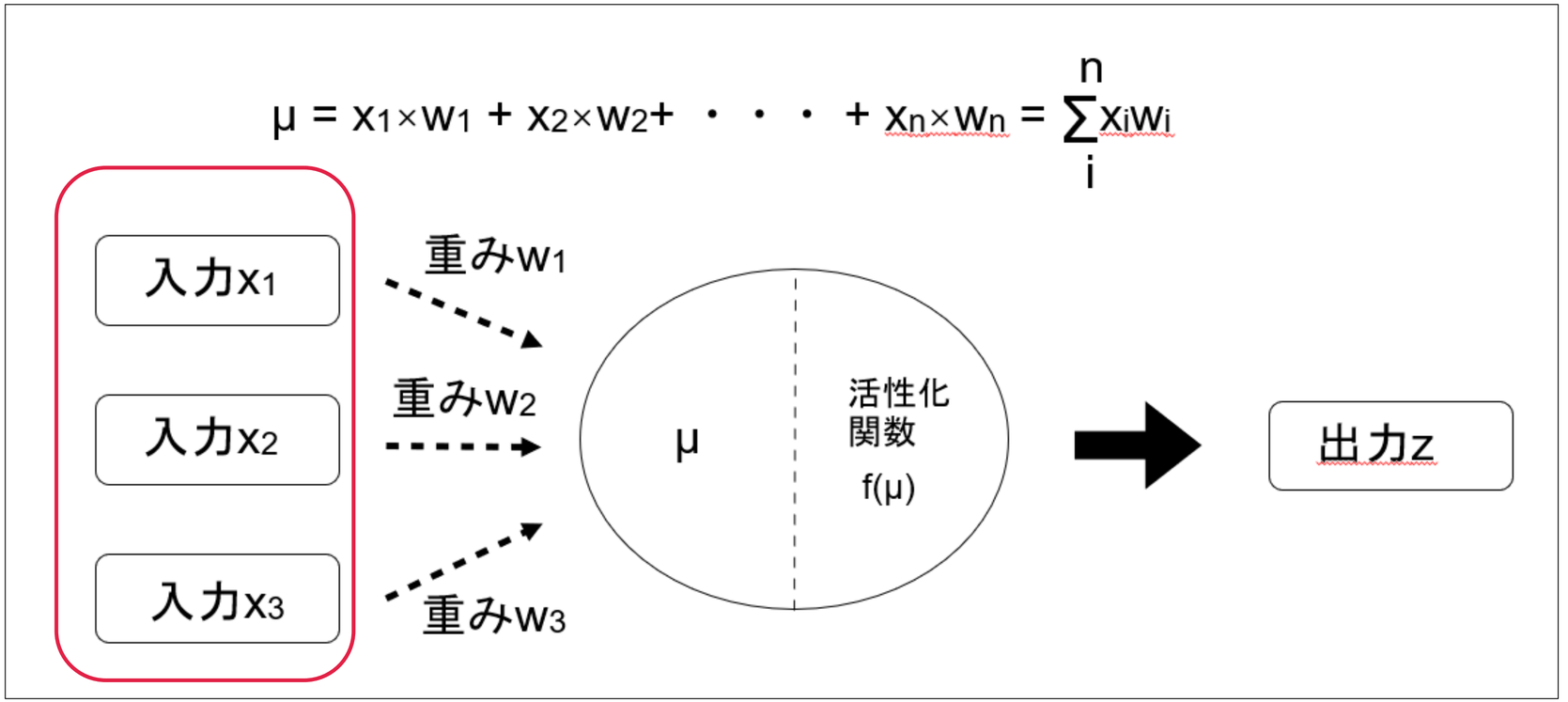
0.11

0.91

0.57

(バイアス項): b

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

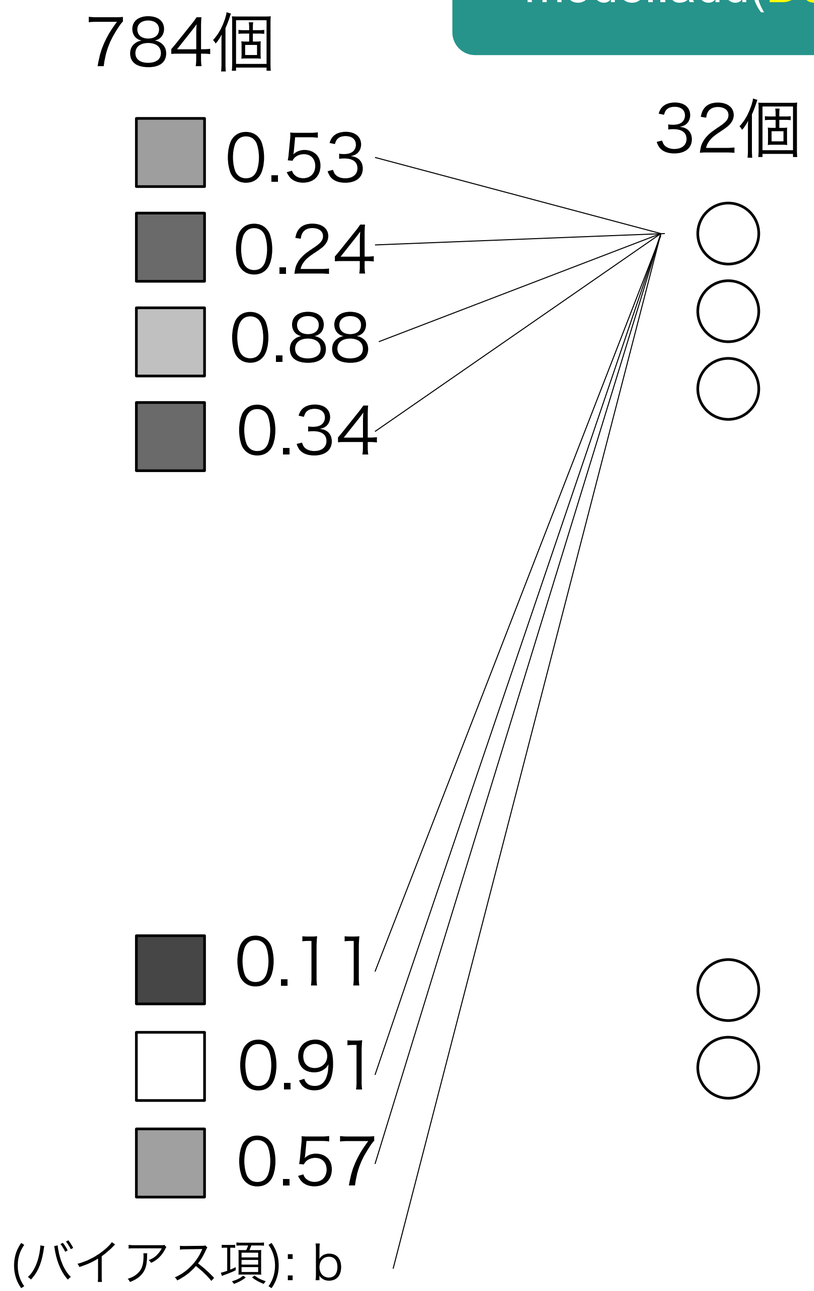


MLPでは１枚ずつモデルに入力する

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

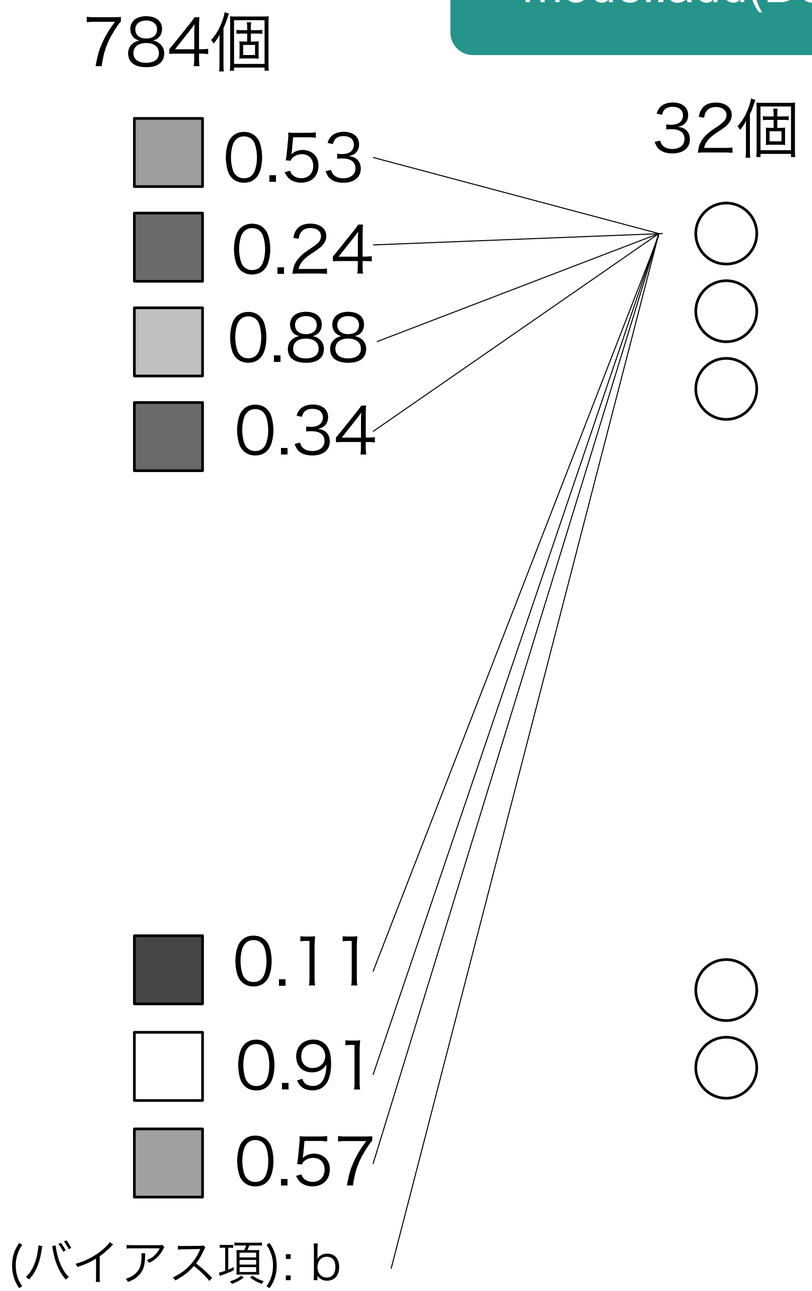
model.add(Dense(32, input_shape=(784,), activation='relu'))



model.add()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'))

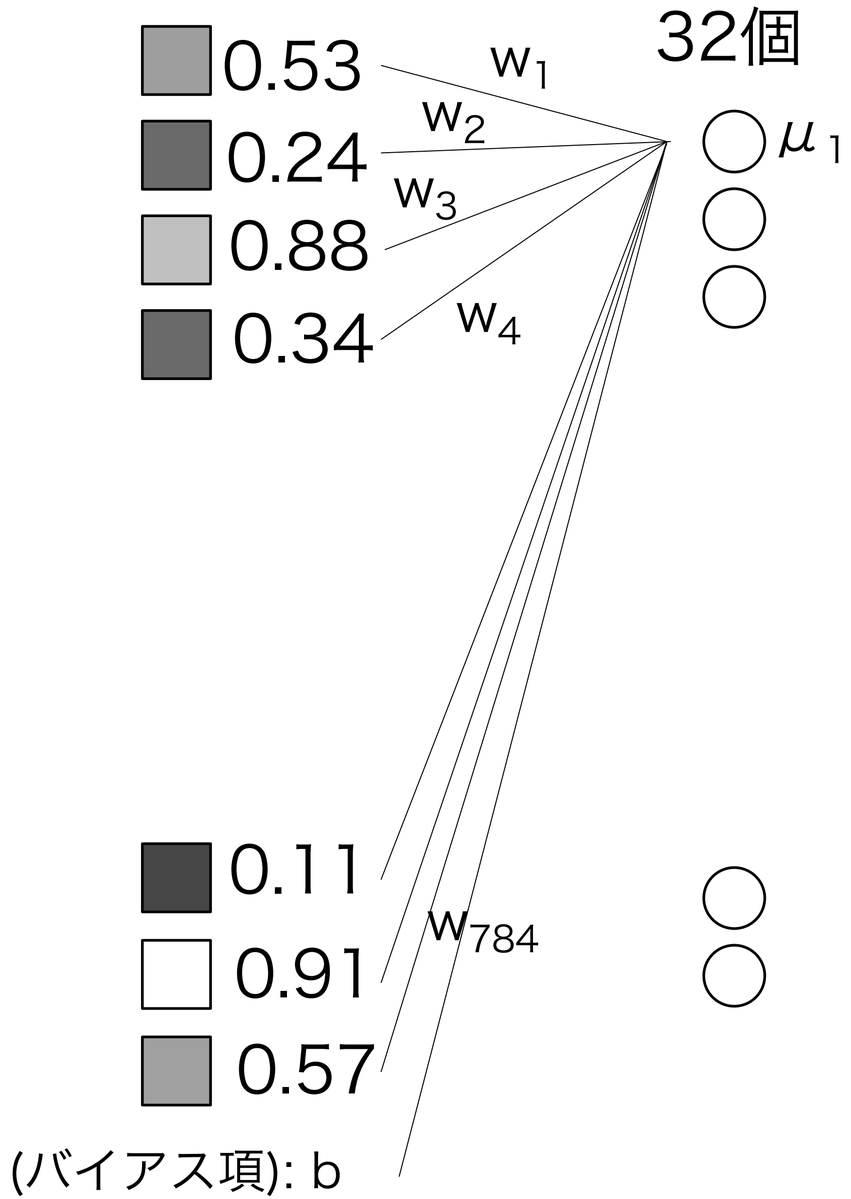


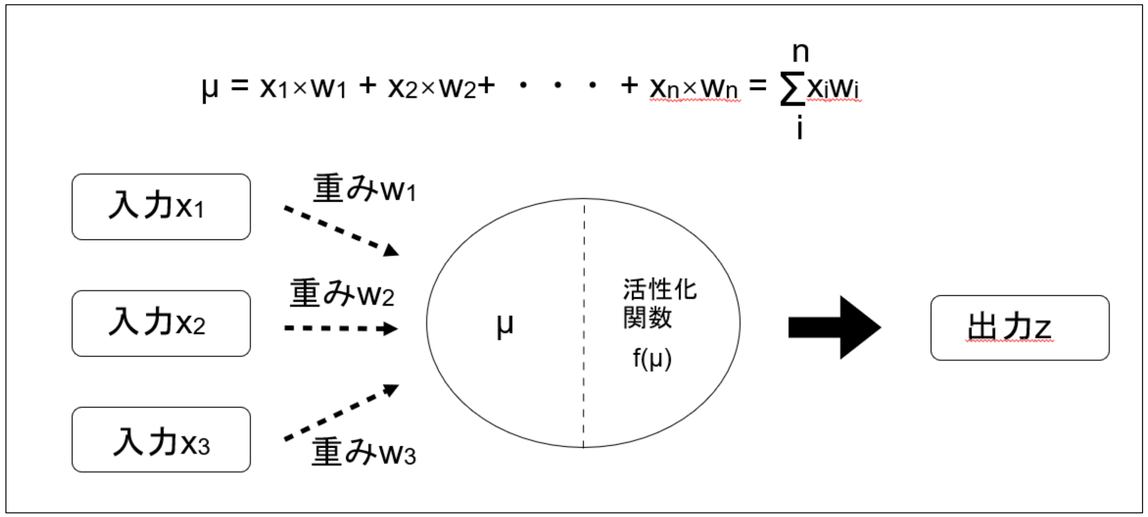
model.add()で層を追加する

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'))

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'))

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

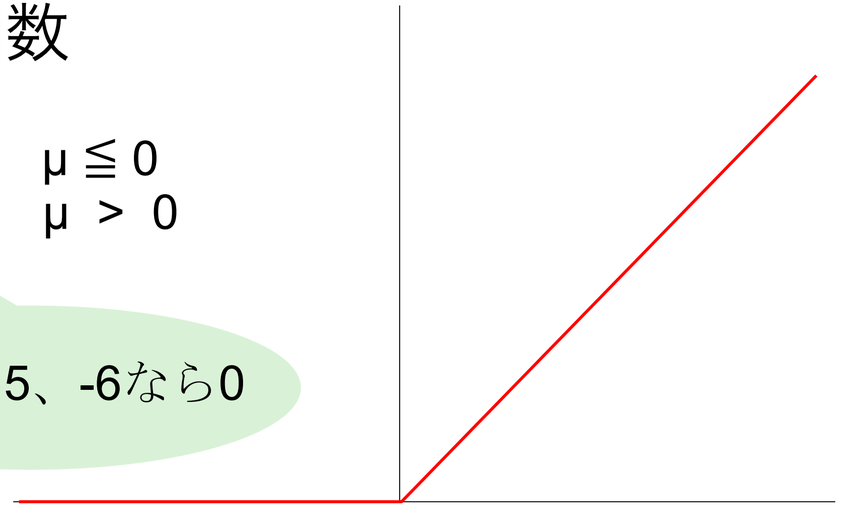
32個

(バイアス項): b

ReLU関数

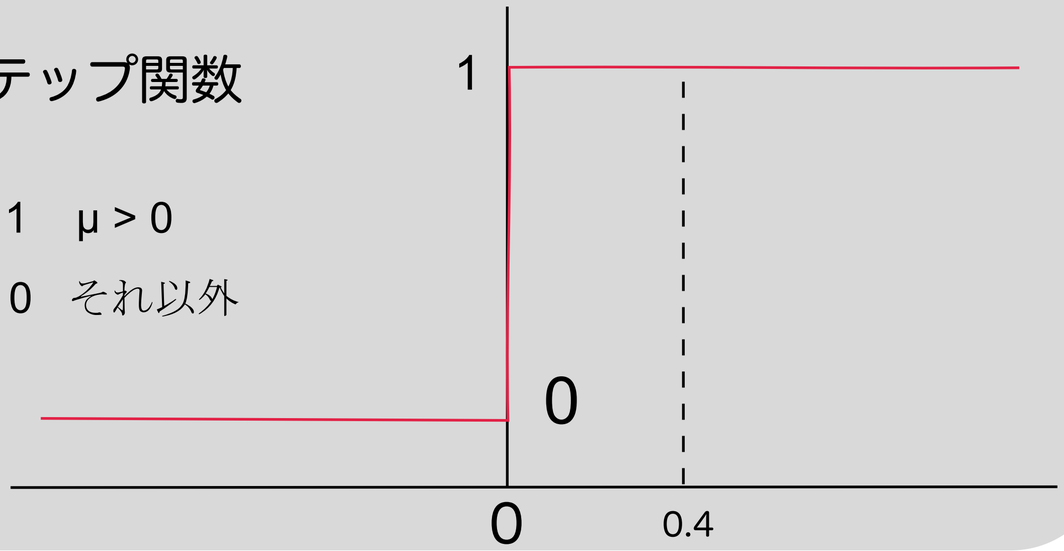
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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'))

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

32個

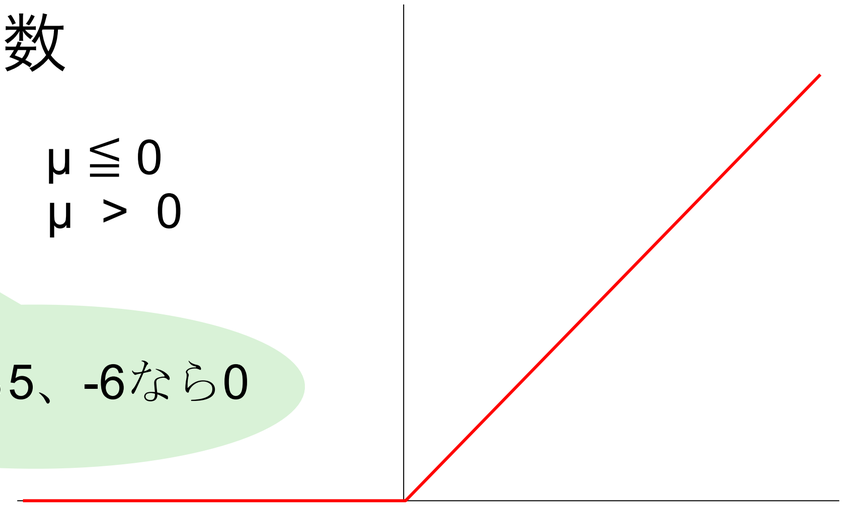
3.23

(バイアス項): b

ReLU関数

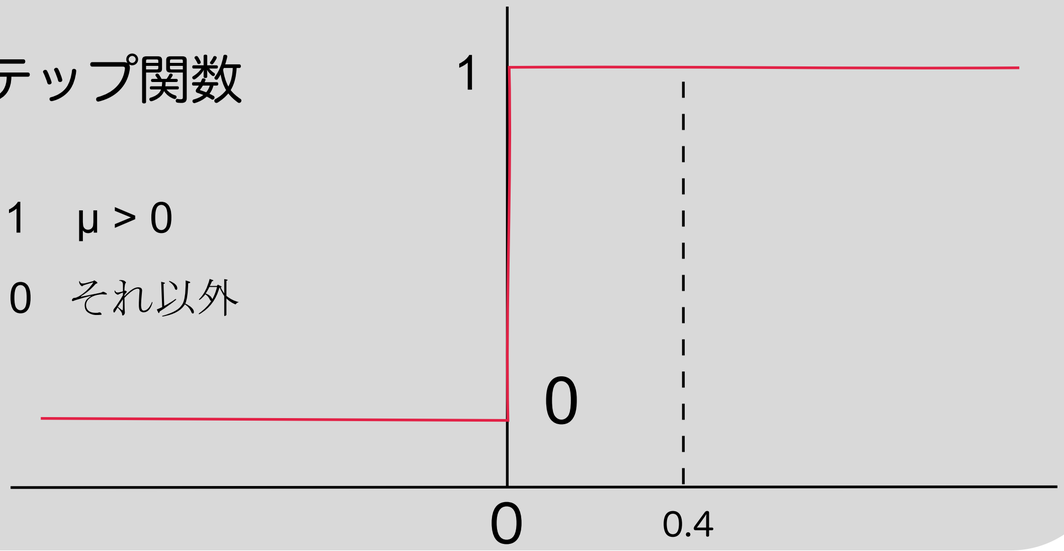
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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'))

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

(バイアス項): b

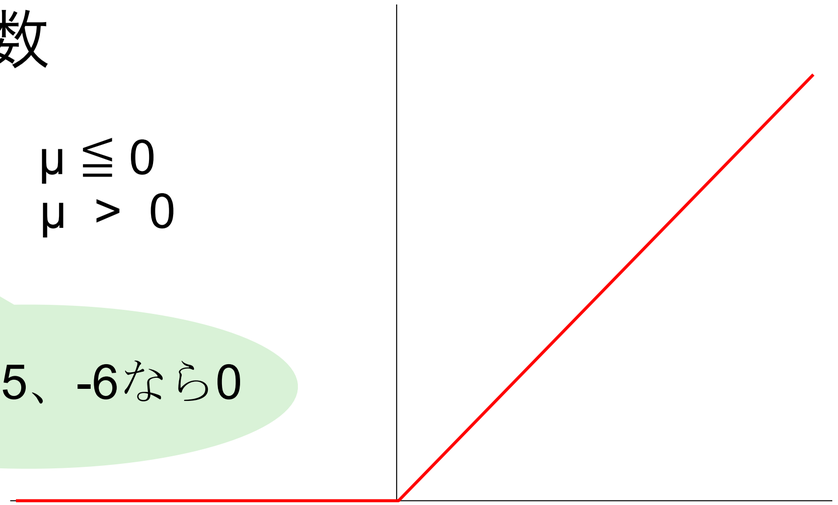
32個

3.23
0
8.45

ReLU関数

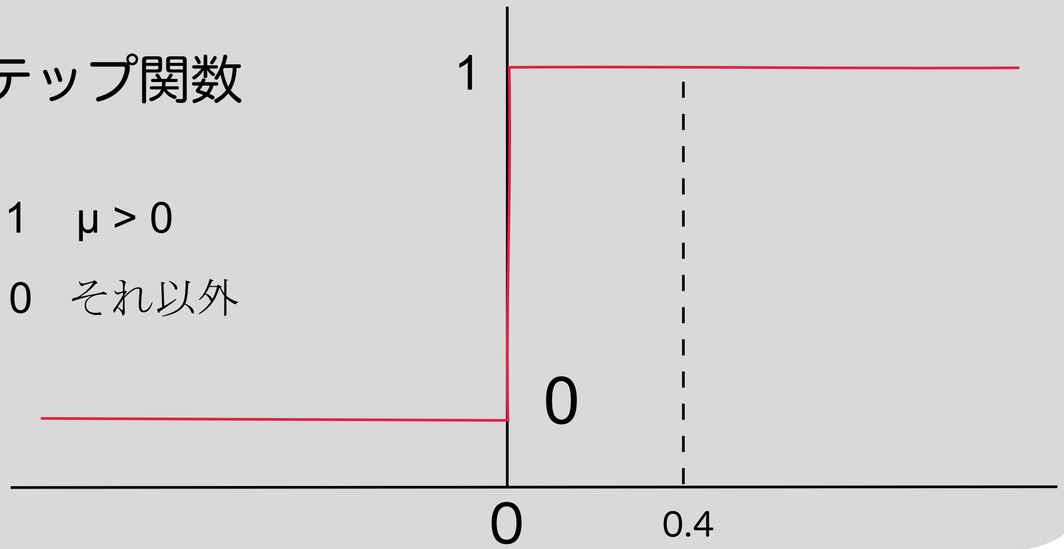
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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'))

784個

0.53
0.24
0.88
0.34
0.11
0.91
0.57

(バイアス項): b

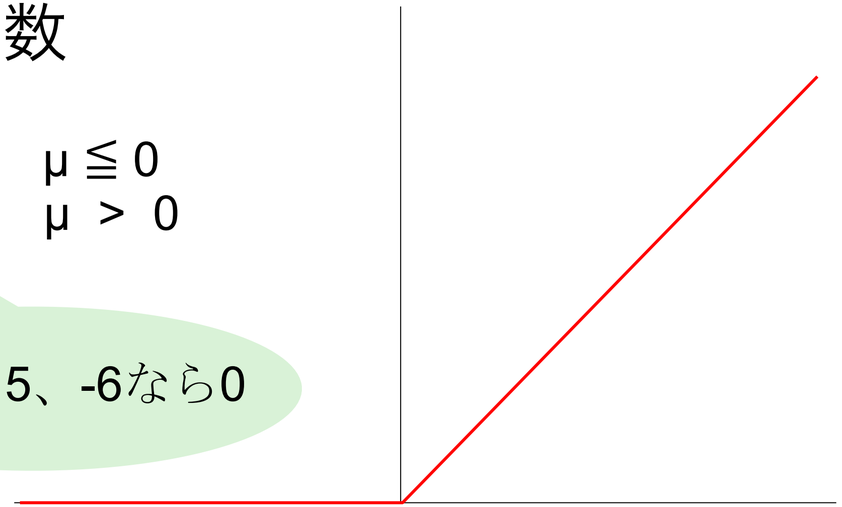
32個

3.23
0
8.45
0
4.33

ReLU関数

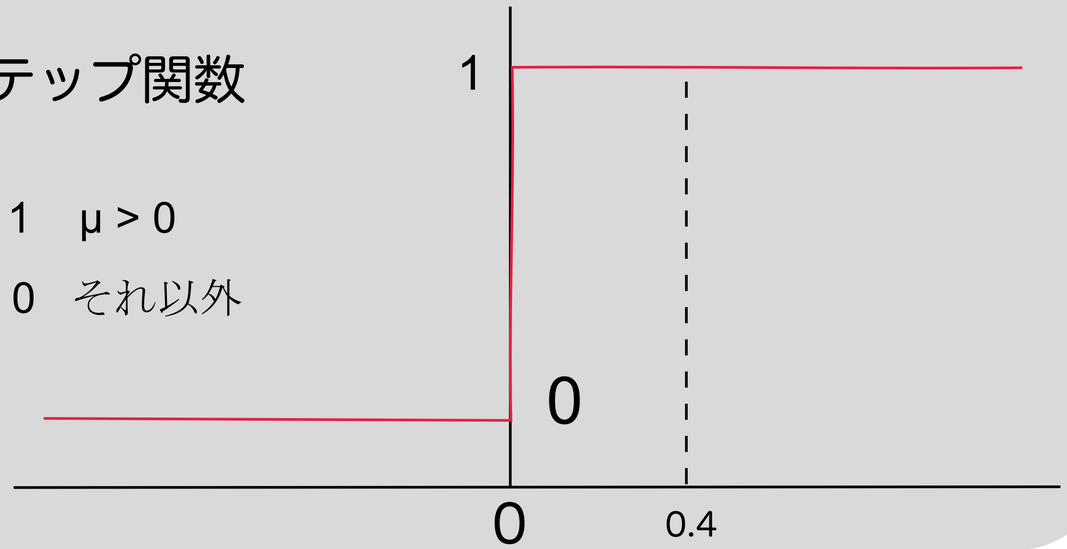
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0

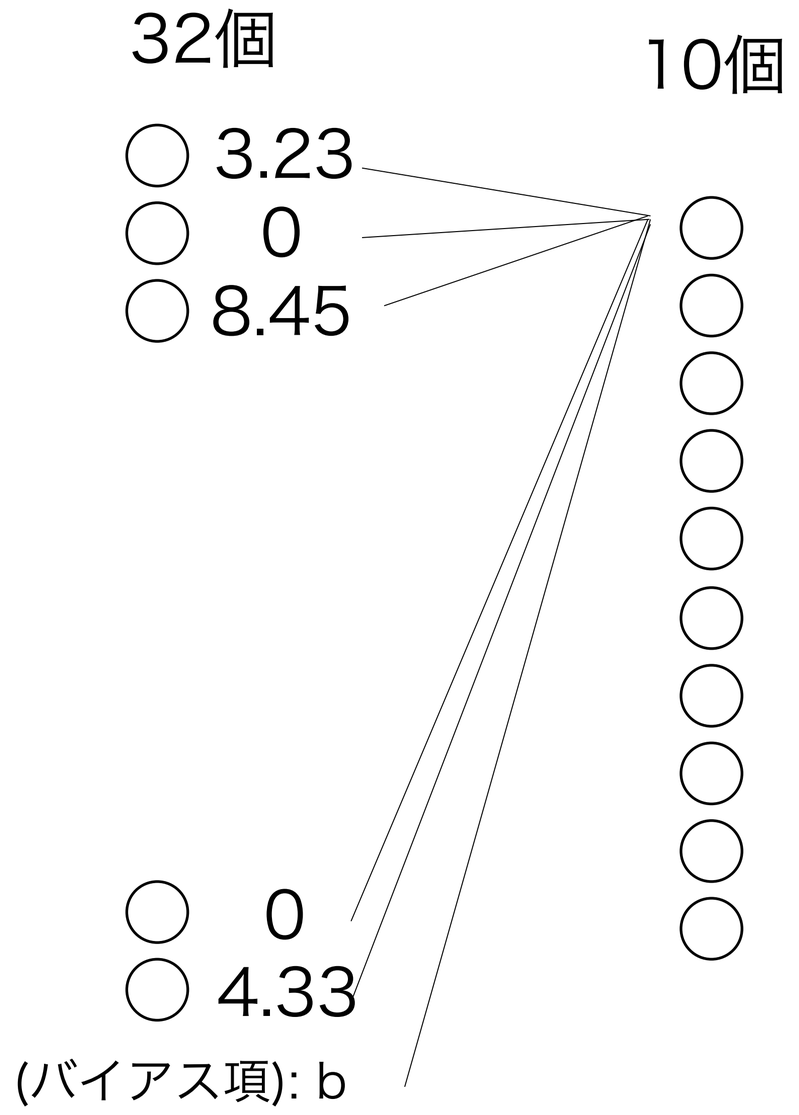


ステップ関数

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



model.add(Dense(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

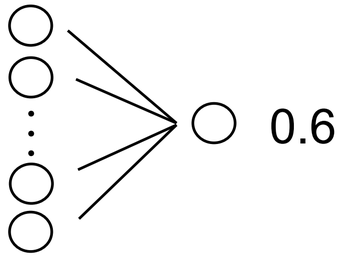
model.add(Dense(10, activation='softmax'))

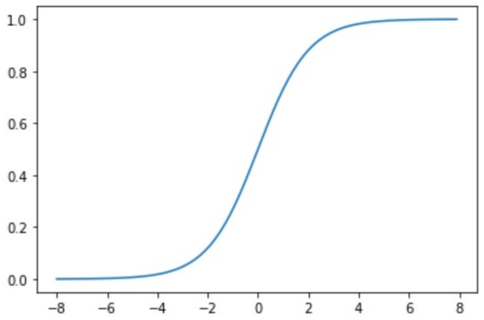
2値分類

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

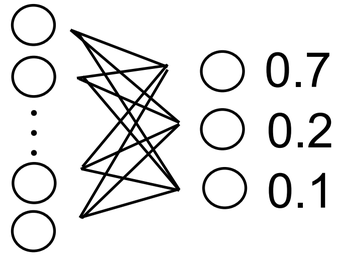
ex) 猫である確率が0.6 (=犬である確率は0.4)

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

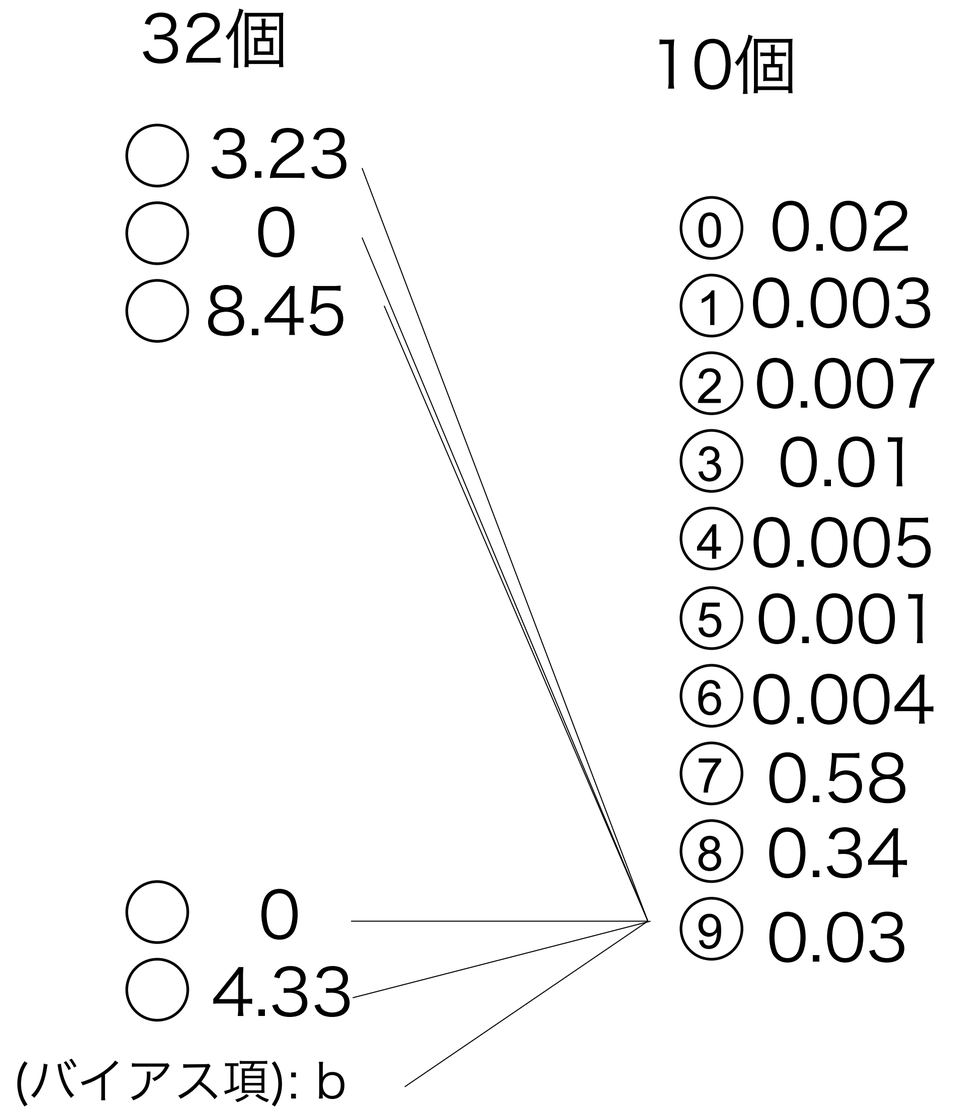
softmax関数
(activation='softmax')



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

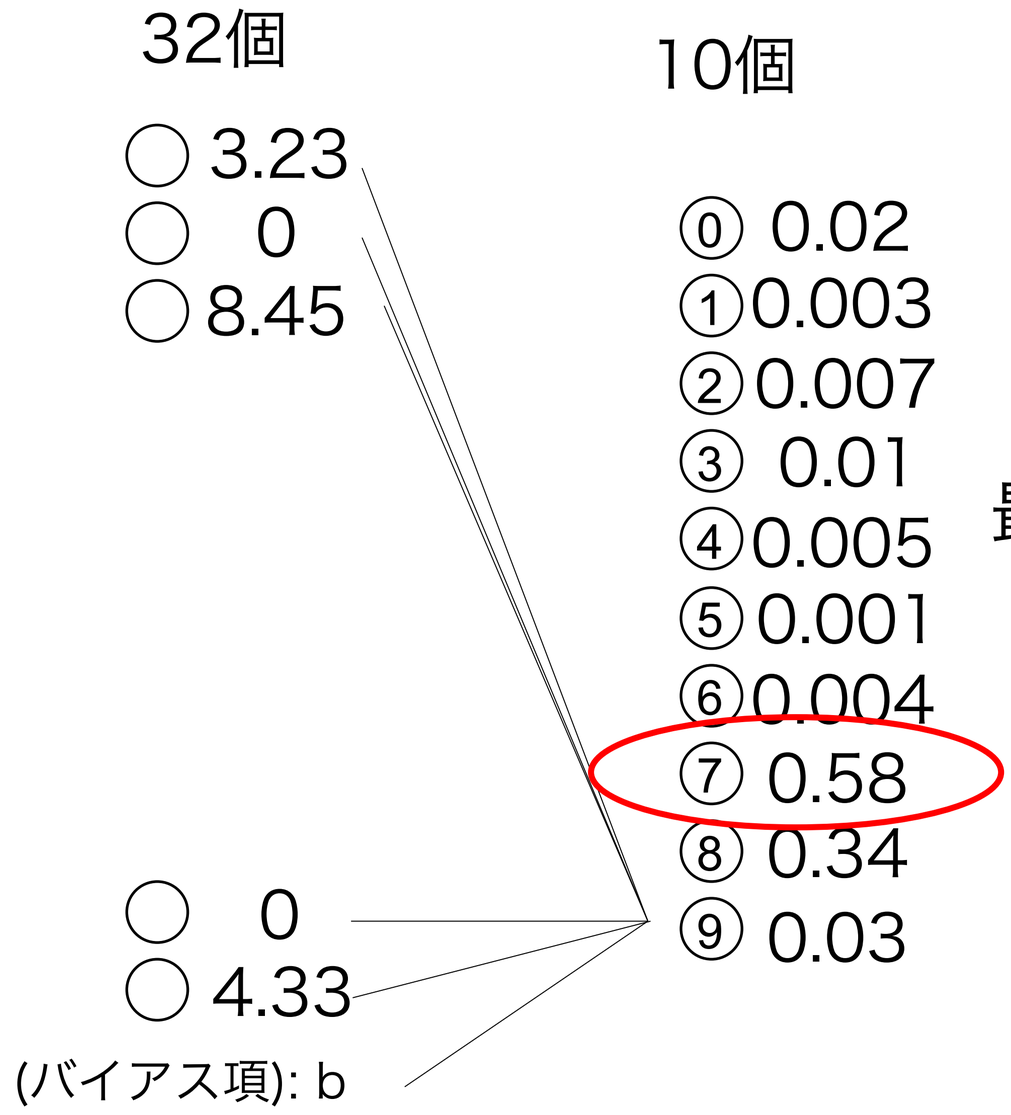
```
model.add(Dense(10, activation='softmax'))
```



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(10, activation='softmax'))



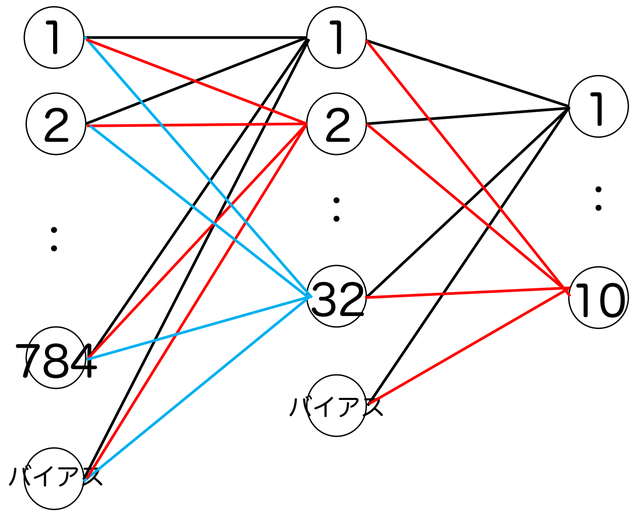
最大の確率が予測される値

例えばこの場合は7と予測

model.summary()

作ったモデルの要約を表示する

```
_________________________________________________________________
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
_________________________________________________________________
```



paramsはパラメーター(変数)
のことでwとbの数

(784+1) × 32 = 25120

(32+1) × 10 = 330

25120 + 330 = 25450

まずはそのまま予測してみる

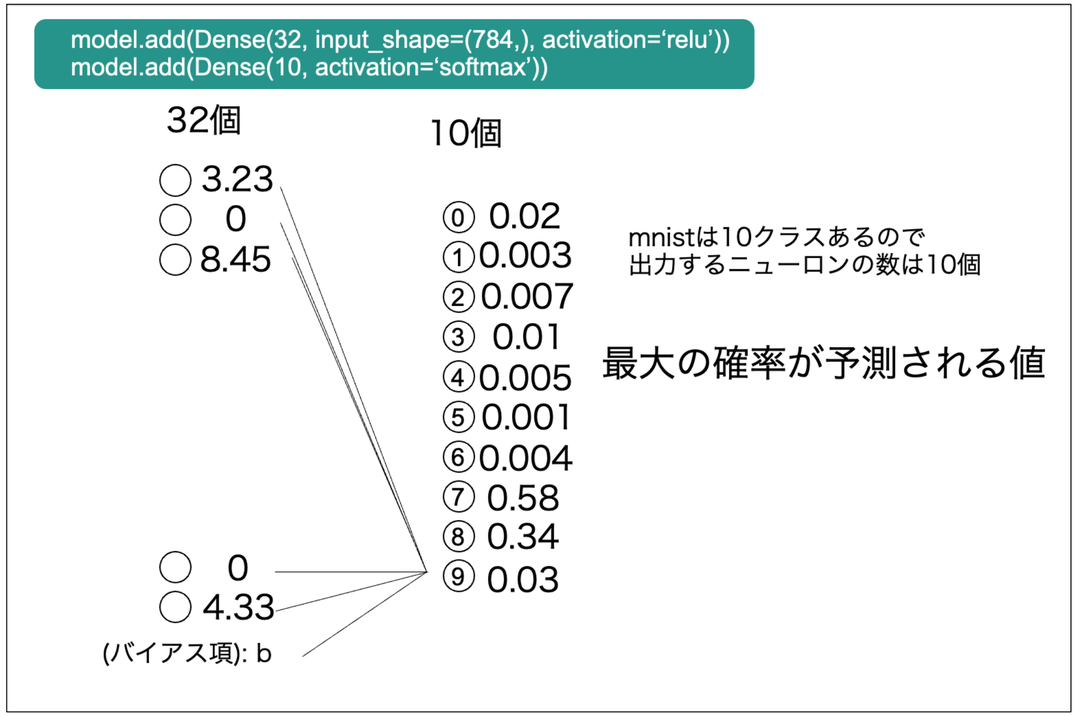
予測はmodel.predict()

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、

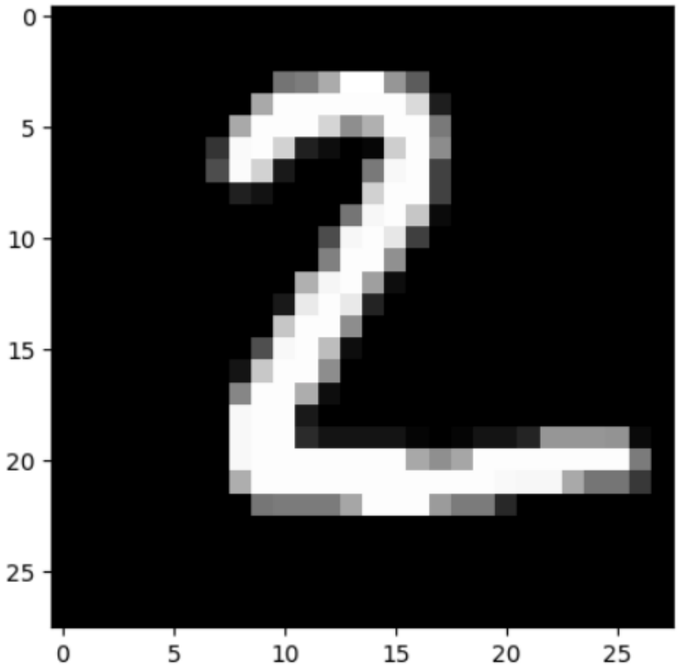
```
print(test[1])
print(y_test[1])
```

```
[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

これが学習するとどうなるか

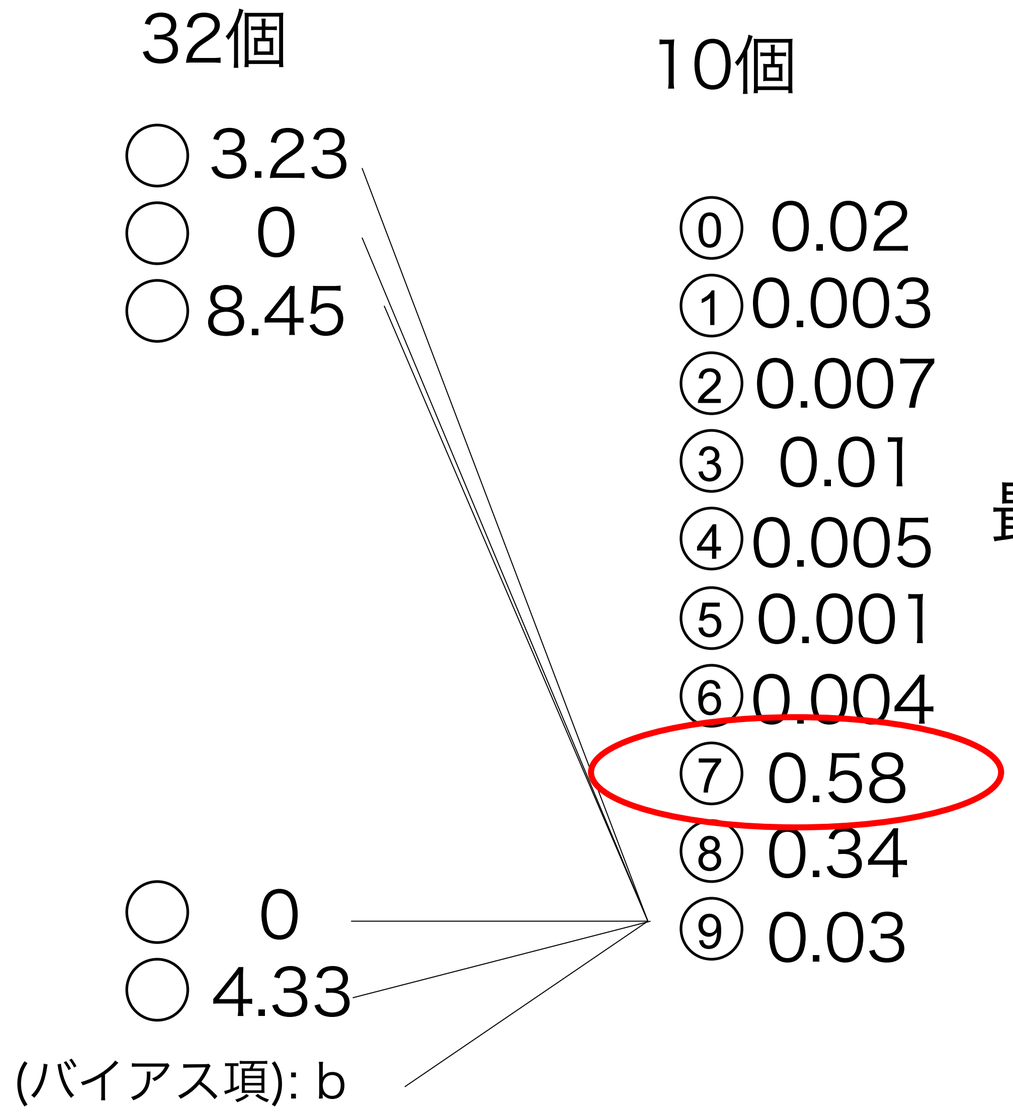
```
import matplotlib.pyplot as plt
plt.imshow(x_test[1],'gray')
plt.show()
```




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

```
2
```

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

最初は全ての重みとバイアスはランダムなので全然正解にならない

重みとバイアスを変えるために学習させる！

model.add(Dense(10, activation='softmax'))

32個

10個

◯ 3.23

◯ 0

◯ 0

◯ 4.33

(バイアス項): b

⓪ 0.02

⑥ 0.004

⑦ 0.58

⑧ 0.34

⑨ 0.03

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

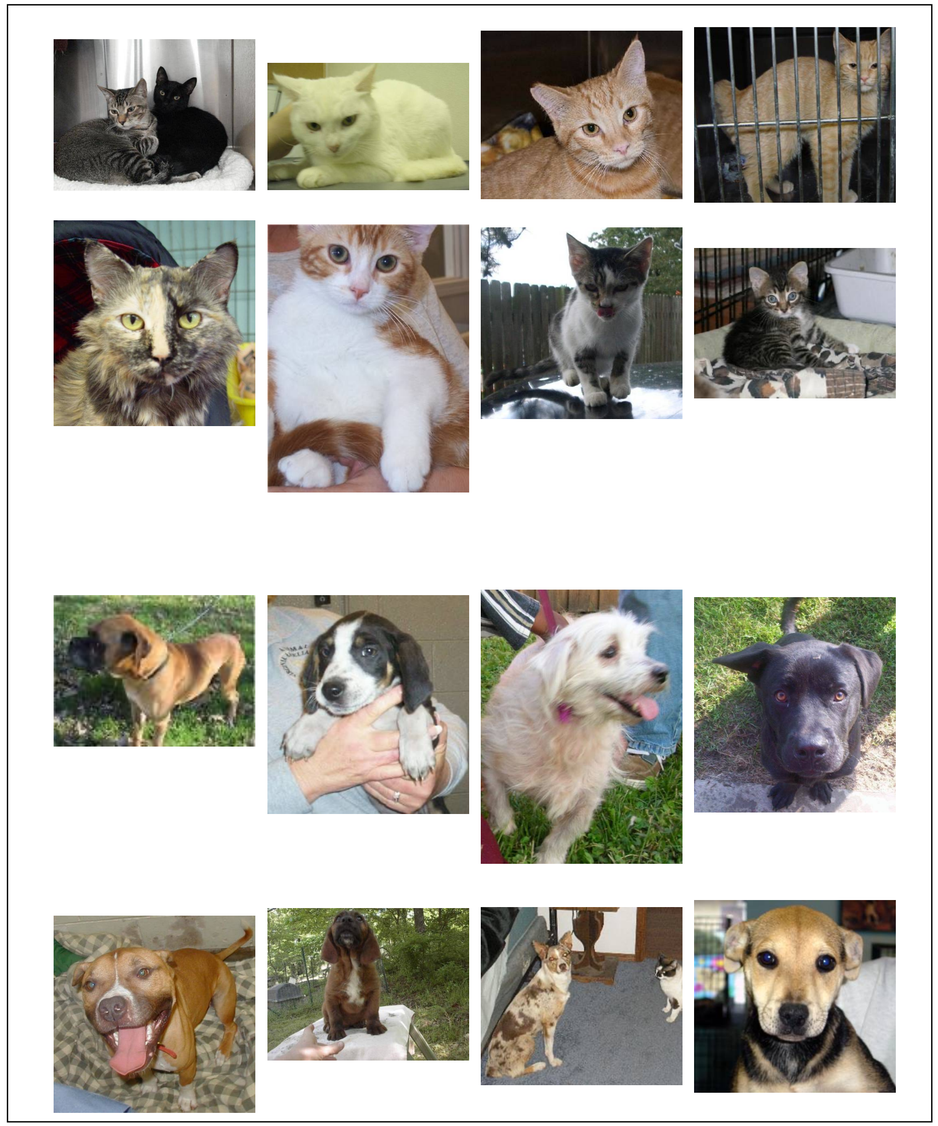
例えばこの場合は7と予測

最初は全ての重みとバイアスはランダムなので全然正解にならない

重みとバイアスを変えるために学習させる！

学習の仕組みの概要

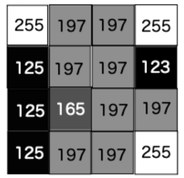
仮に猫と犬の画像の2値分類で考えると、



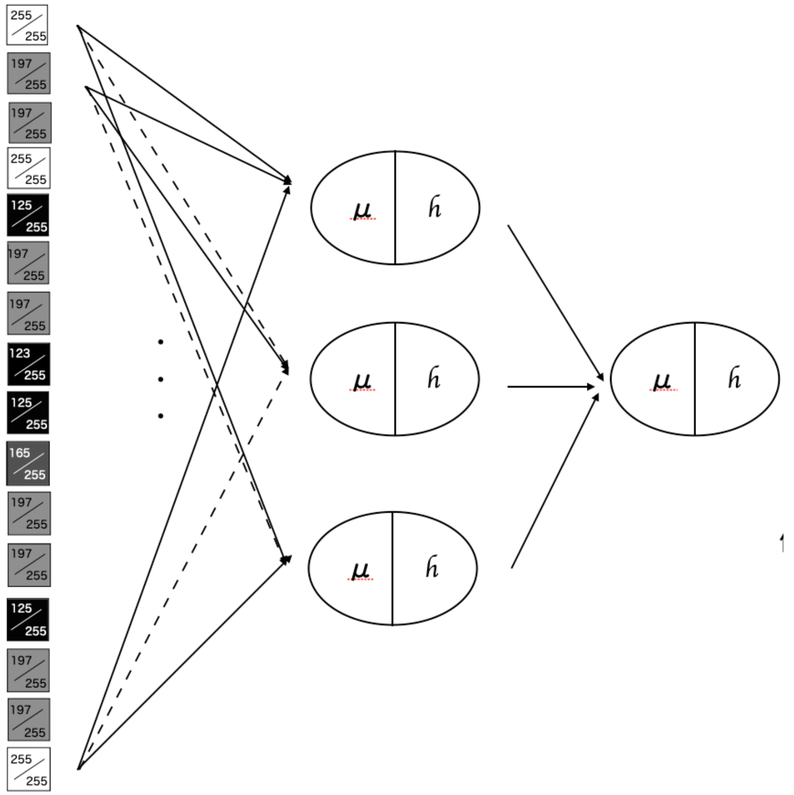
これらの画像の正解は猫(=1とする)

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枚ずつ取り出す)



8枚の予測結果を得る

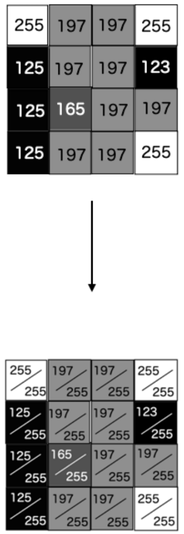


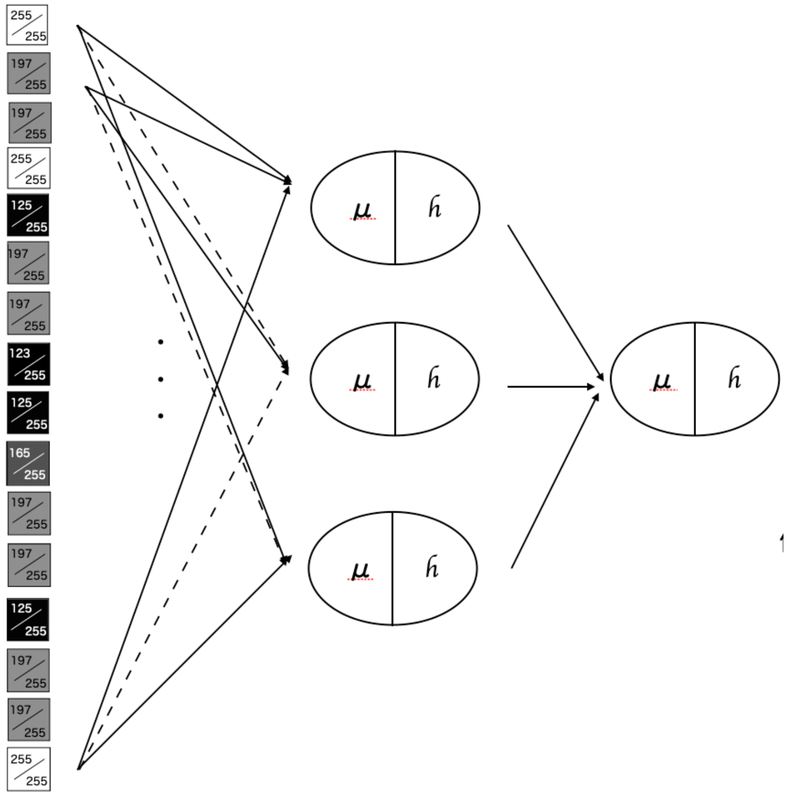
猫が１，犬が０

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枚ずつ取り出す)





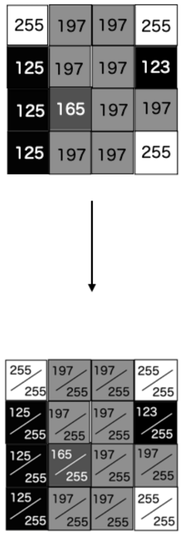
猫が１，犬が０

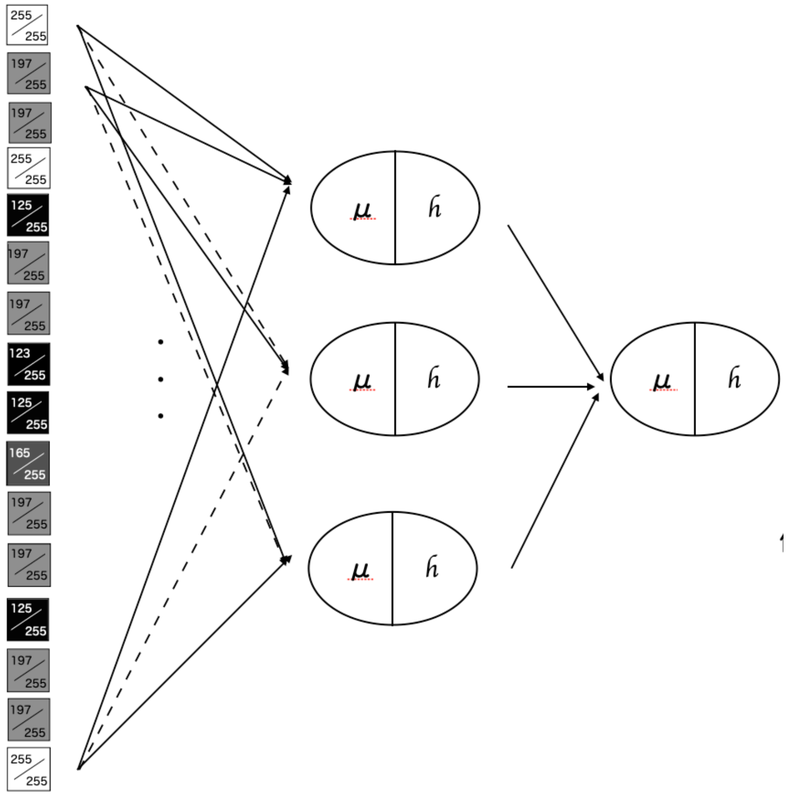
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

この出力と正解の
ズレ(誤差L)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

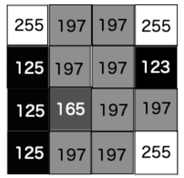
誤差L = 0.8
(だったとします)

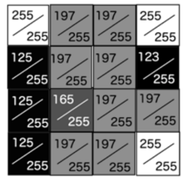
2値分類の時はbinary crossentropy

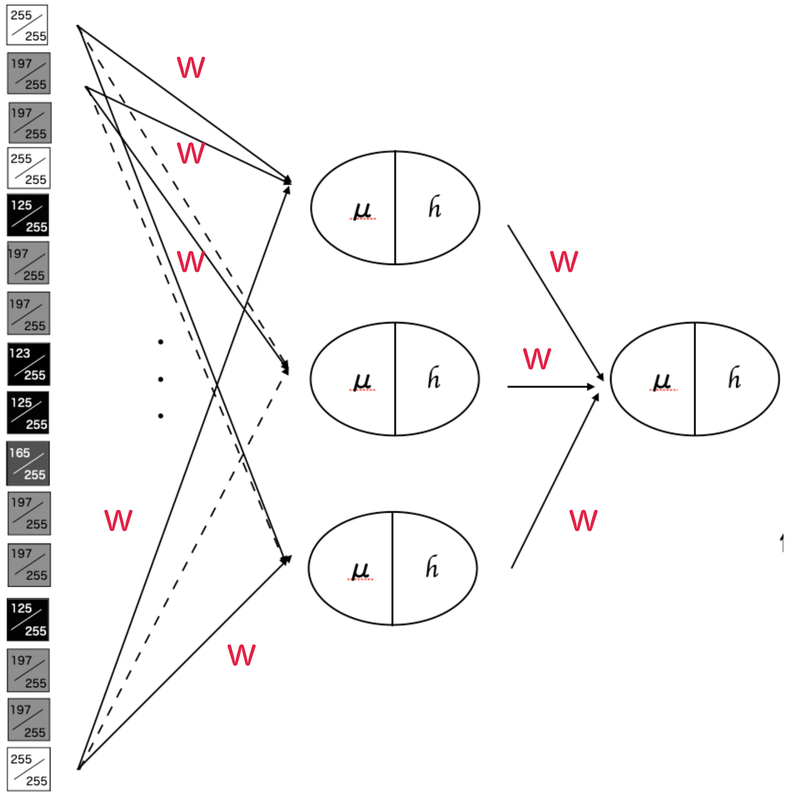
多値分類の時はcategorical crossentropy　　がよく使われる

モデルと入力データ(画像)は変えない
→重みwとバイアスbが変わると誤差Lは変化する

(wとbがいい感じに変化すれば誤差Lは小さくなるはず!)





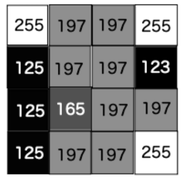


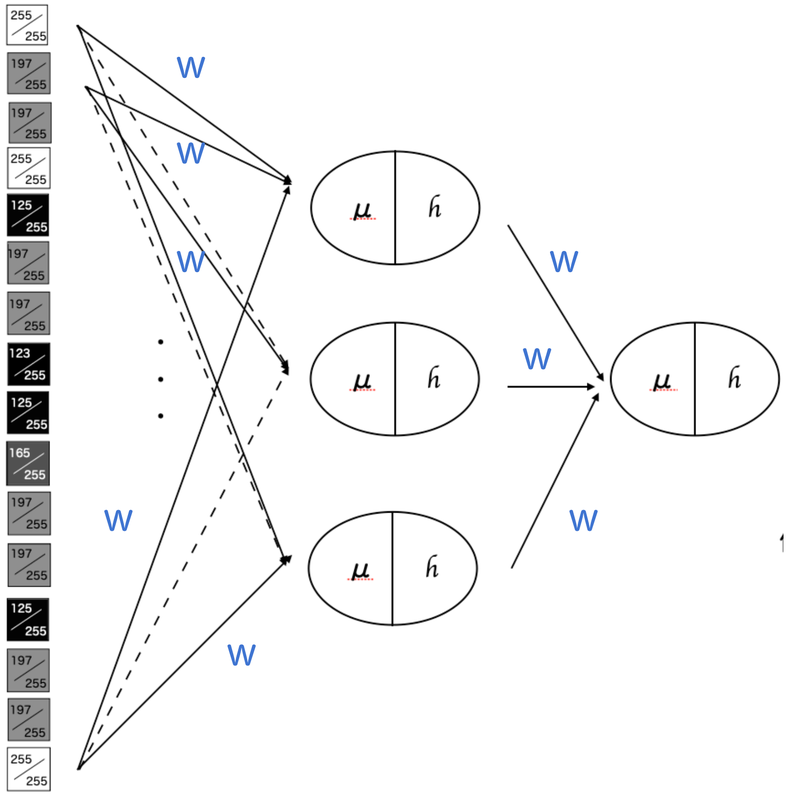
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

誤差L = 0.8

(誤差の計算方法である)
損失関数は重みとバイアスの式で表せる　L(w,b) = 0.8

誤差(損失関数)が少しずつ小さくなるように重みとバイアスを更新する最適化アルゴリズムを設定する





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

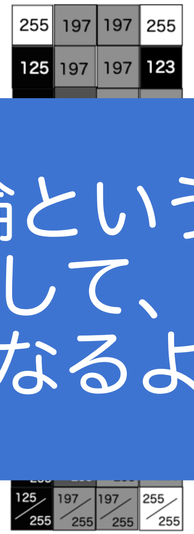
誤差L = 0.8

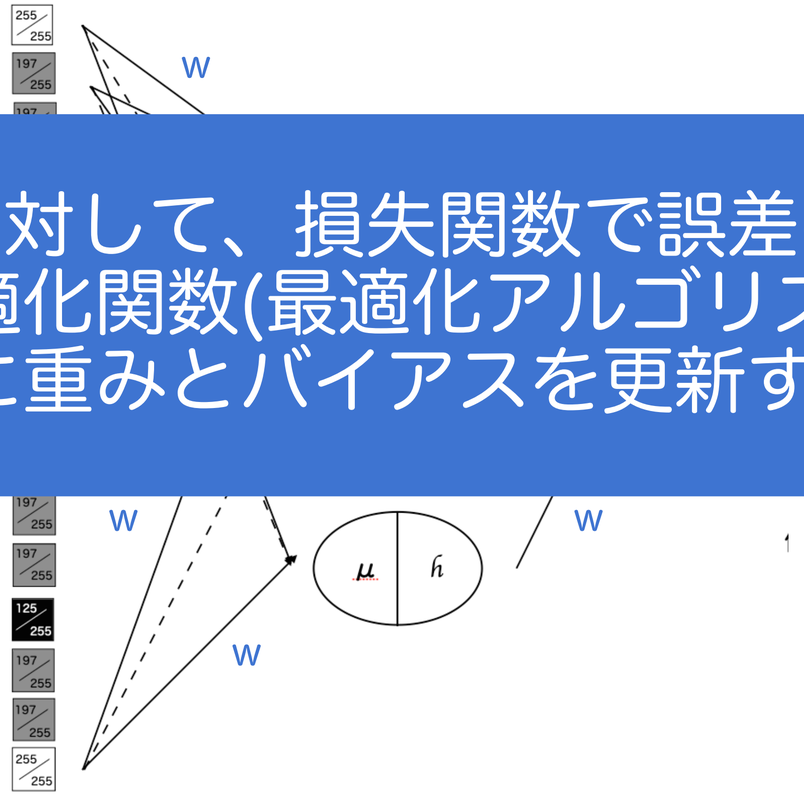
最適化アルゴリズム
Adam

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

誤差(損失関数)が少しずつ小さくなるように重みとバイアスを更新する最適化アルゴリズムを設定する





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

- 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

誤差L = 0.8

最適化アルゴリズム
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

この1行を先ほどのmodel.addの後に追加

```
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()
```

model.summary()

作ったモデルの要約を表示する

```
_________________________________________________________________
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
_________________________________________________________________
```

モデルの構造は変わらない

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

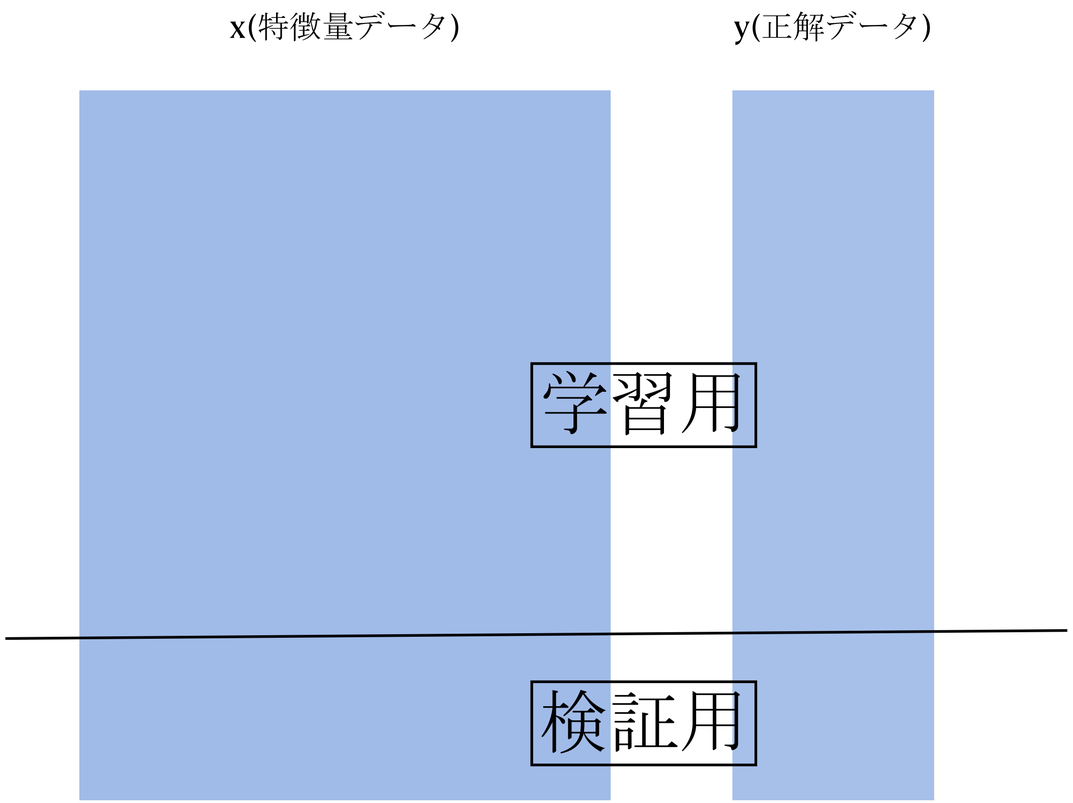
```
Epoch 1/30
750/750 [==============================] - 4s 4ms/step - loss: 0.4633 - accuracy: 0.8709 - val_loss: 0.2517 - val_accuracy: 0.9298
Epoch 2/30
750/750 [==============================] - 3s 4ms/step - loss: 0.2401 - accuracy: 0.9309 - val_loss: 0.2020 - val_accuracy: 0.9448
Epoch 3/30
750/750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

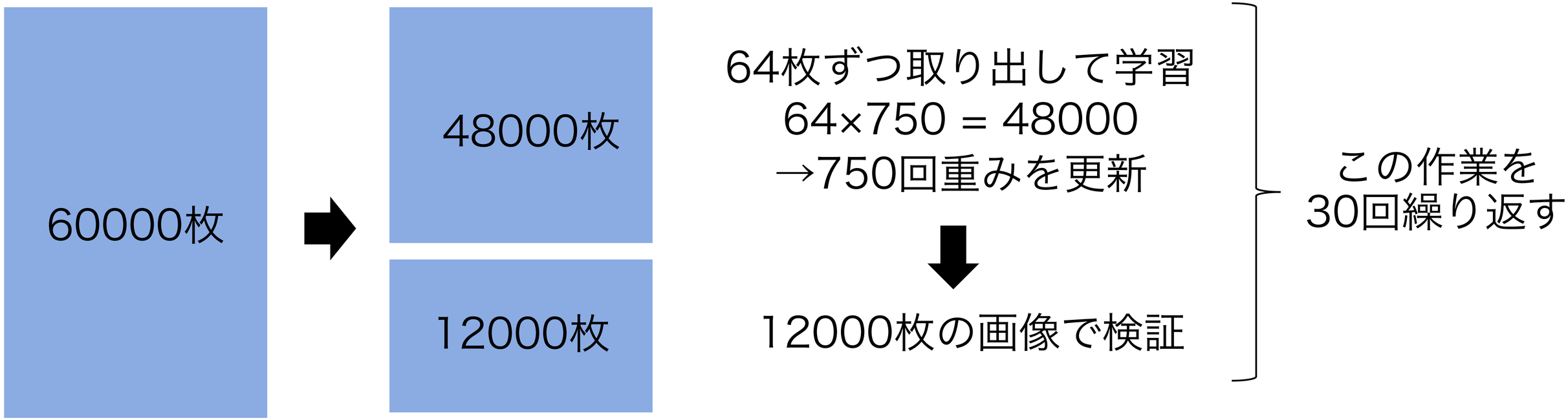
ホールドアウト法



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)

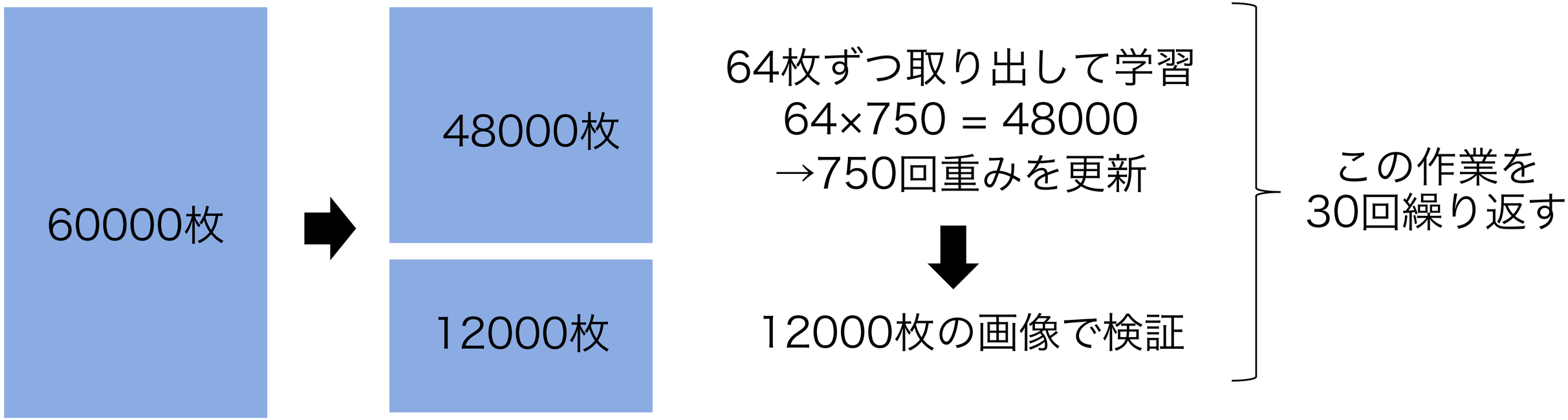
```
result = model.fit(x_train, y_train, epochs=30, batch_size=64, validation_split=0.2)
```

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

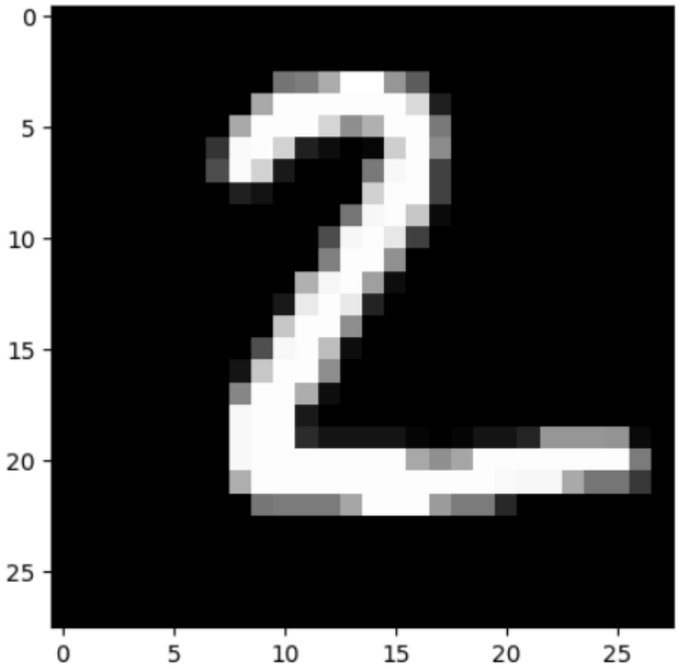
```
import numpy as np
print(np.around(test2[1],3))
print(y_test[1])
```

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

np.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
```

課題

・WebClassにある課題3をやりましょう

締め切りは1週間後の5/16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。
(1週間後に正解をアップします)