医療とAI・ビッグデータ応用

MLP

統合教育機構
須藤毅顕

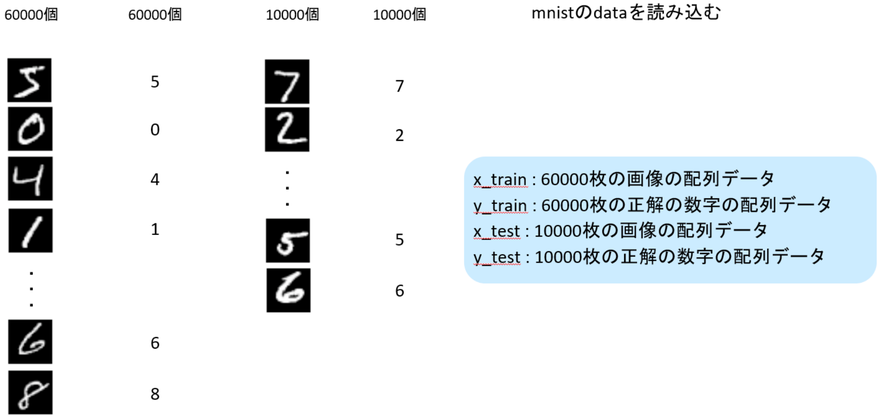
医療とAI・ビッグデータ応用

①7/14　11:10-12:00　MNISTの読み込み
②7/21　11:10-12:00　MNISTの可視化と加工
③10/20　10:05-10:55　深層学習のMLPの理解(学習まで)
④10/20　11:10-12:00　深層学習のMLPの理解(可視化と評価まで)
⑤(オンデマンド)　深層学習のCNNの理解(学習まで)
⑥11/17　11:10-12:00　深層学習のCNNの理解(可視化と評価まで)
⑦12/15　10:05-11:55　グループ演習(CIFAR10でのMLPとCNN)
⑧1/19　11:10-12:00　教師なし学習(次元削減)
⑨1/26　10:05-10:55　教師なし学習(クラスタリング)
⑩1/26　11:10-12:00　総括(講義予定)

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)
```

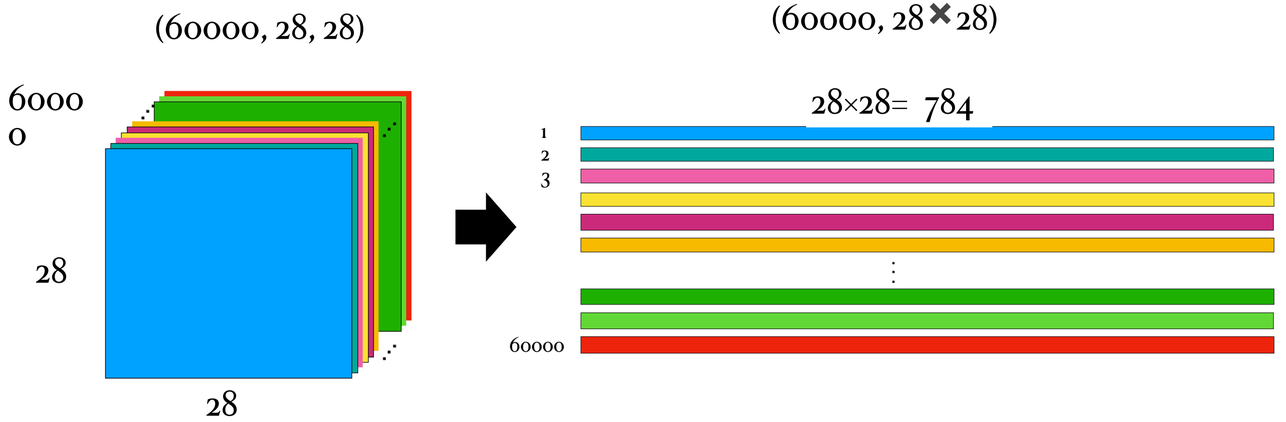
```
(60000, 784)
(10000, 784)
```

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

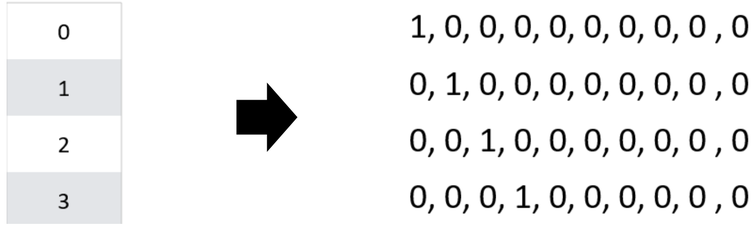
```
(60000, 10)
(10000, 10)
```

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)



医療とAI・ビッグデータ応用

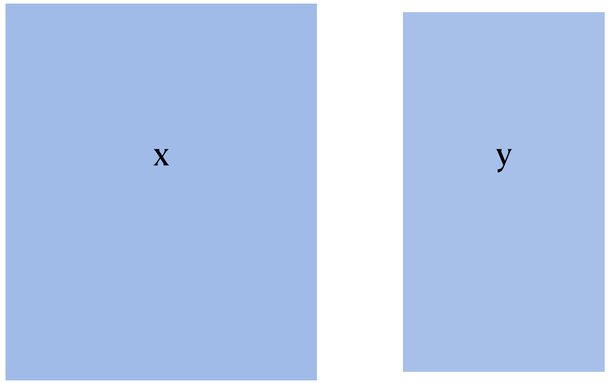
①7/14　11:10-12:00　MNISTの読み込み
②7/21　11:10-12:00　MNISTの可視化と加工
③10/20　10:05-10:55　深層学習のMLPの理解(学習まで)
④10/20　11:10-12:00　深層学習のMLPの理解(可視化と評価まで)
⑤(オンデマンド)　深層学習のCNNの理解(学習まで)
⑥11/17　11:10-12:00　深層学習のCNNの理解(可視化と評価まで)
⑦12/15　10:05-11:55　グループ演習(CIFAR10でのMLPとCNN)
⑧1/19　11:10-12:00　教師なし学習(次元削減)
⑨1/26　10:05-10:55　教師なし学習(クラスタリング)
⑩1/26　11:10-12:00　総括(講義予定)

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)

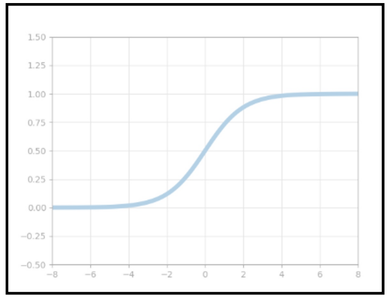
y(正解データ)



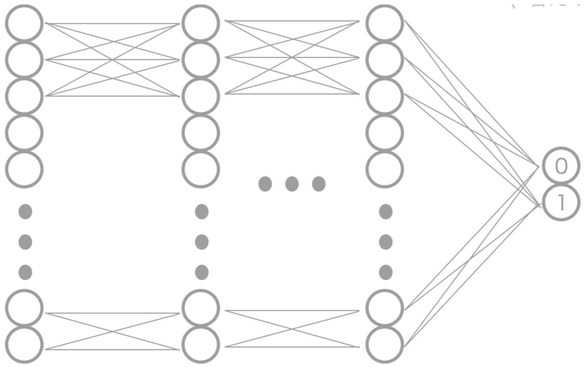
データを配列に整える

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

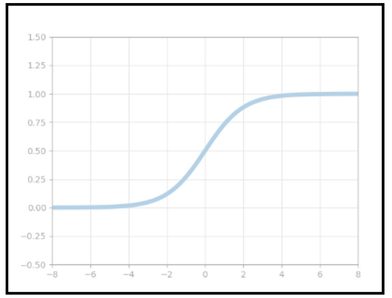
y(正解データ)

y

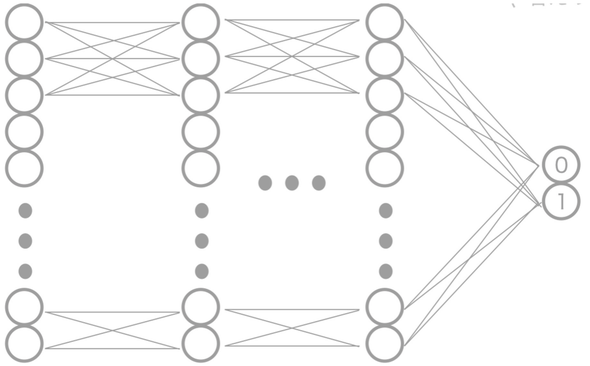
データを配列に整える

学習させる

ロジスティック回帰分析



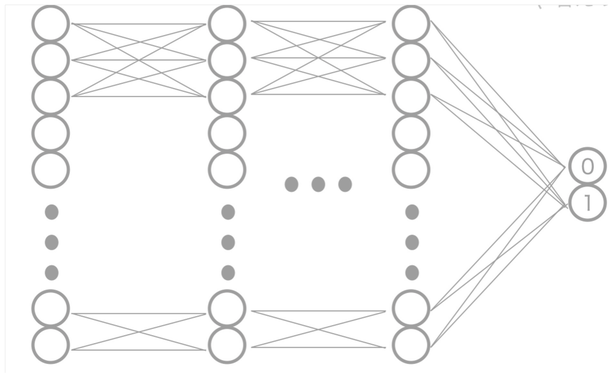
ニューラルネットワーク



達成したいこと

MLP

ニューラルネットワーク

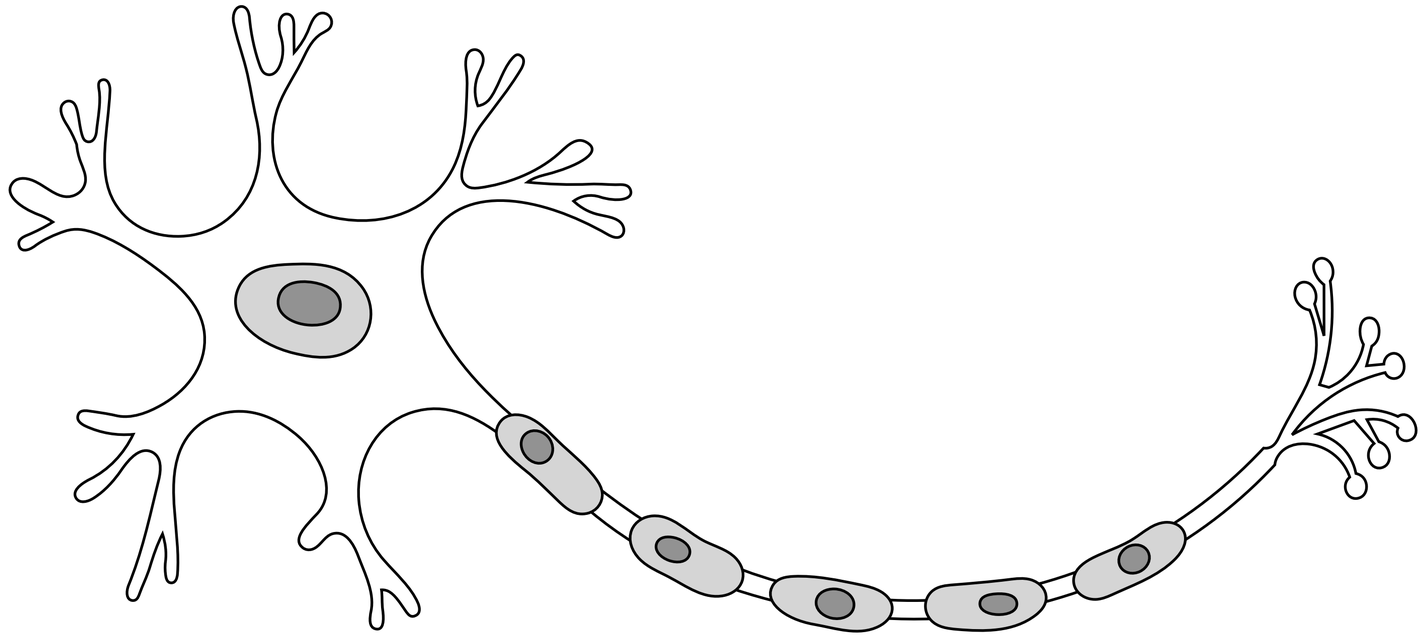


画像を分類
（１０クラス）

0 : T-shirt/top
、
1 : Trouser、
2 : Pullover、
3 : Dress、
4 : Coat、
5 : Sandal
6 : Shirt、
7 : Sneaker、
8 : Bag、
9 : Ankle boot

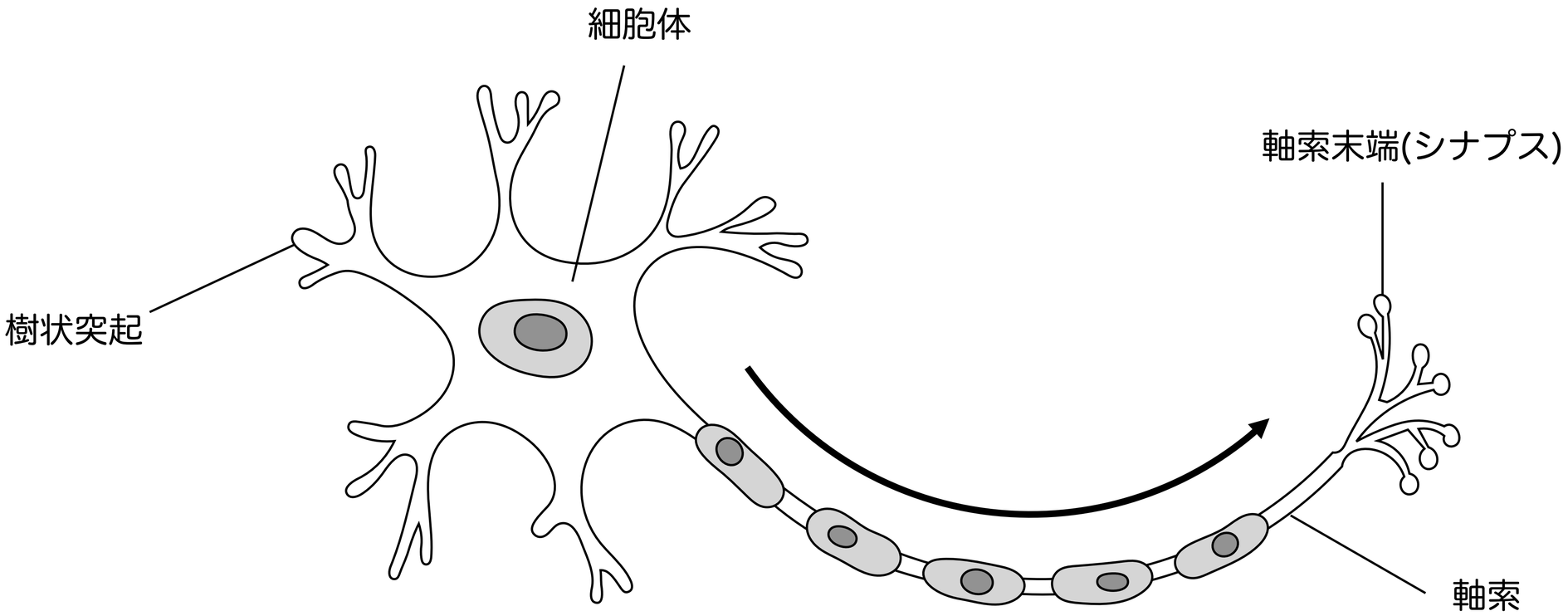
ニューロンとパーセプトロン

脳は膨大な数の神経細胞(ニューロン)から構成されていて、このニューロンが脳の基本単位。ニューロンが互いに結合することで、巨大なネットワークを作り出し、学習機能や情報処理の機能を実現している。
深層学習におけるニューラルネットワークは、本物の脳を模した人工ニューラルネットワークと呼ばれる。



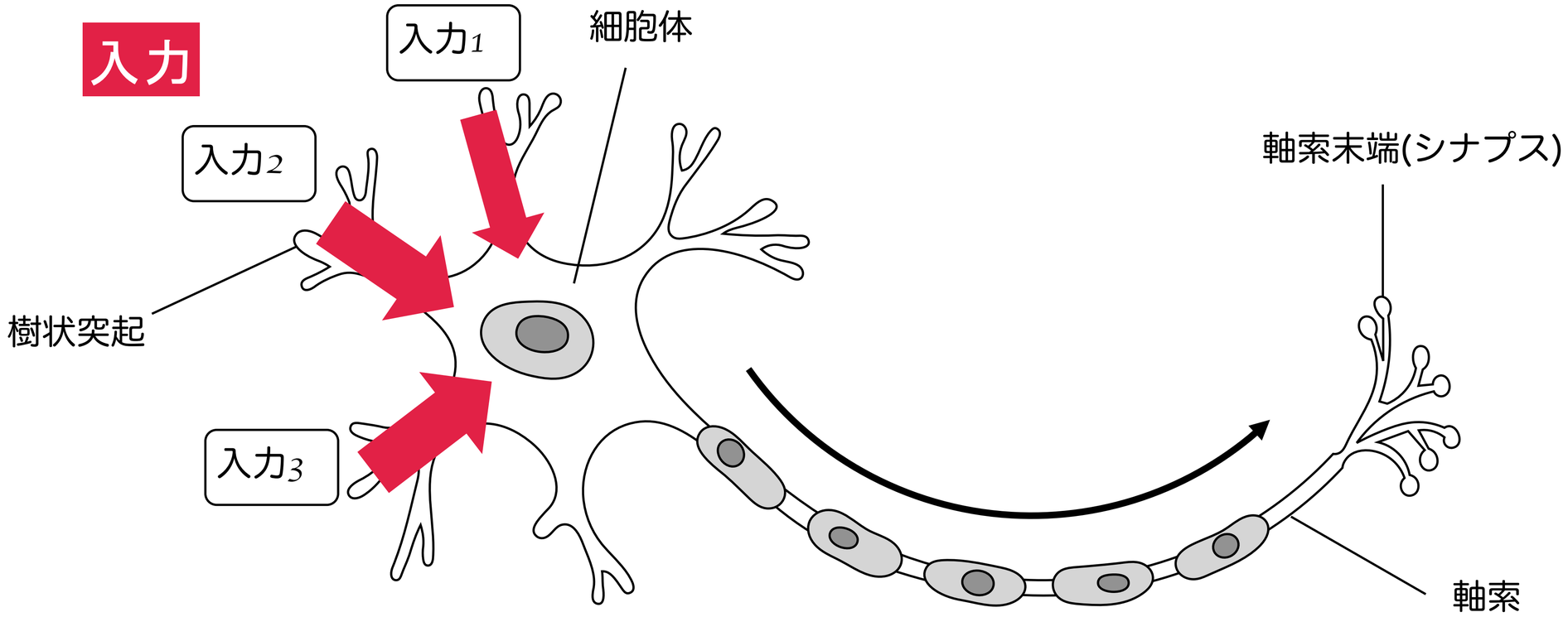
ニューロンとパーセプトロン

・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



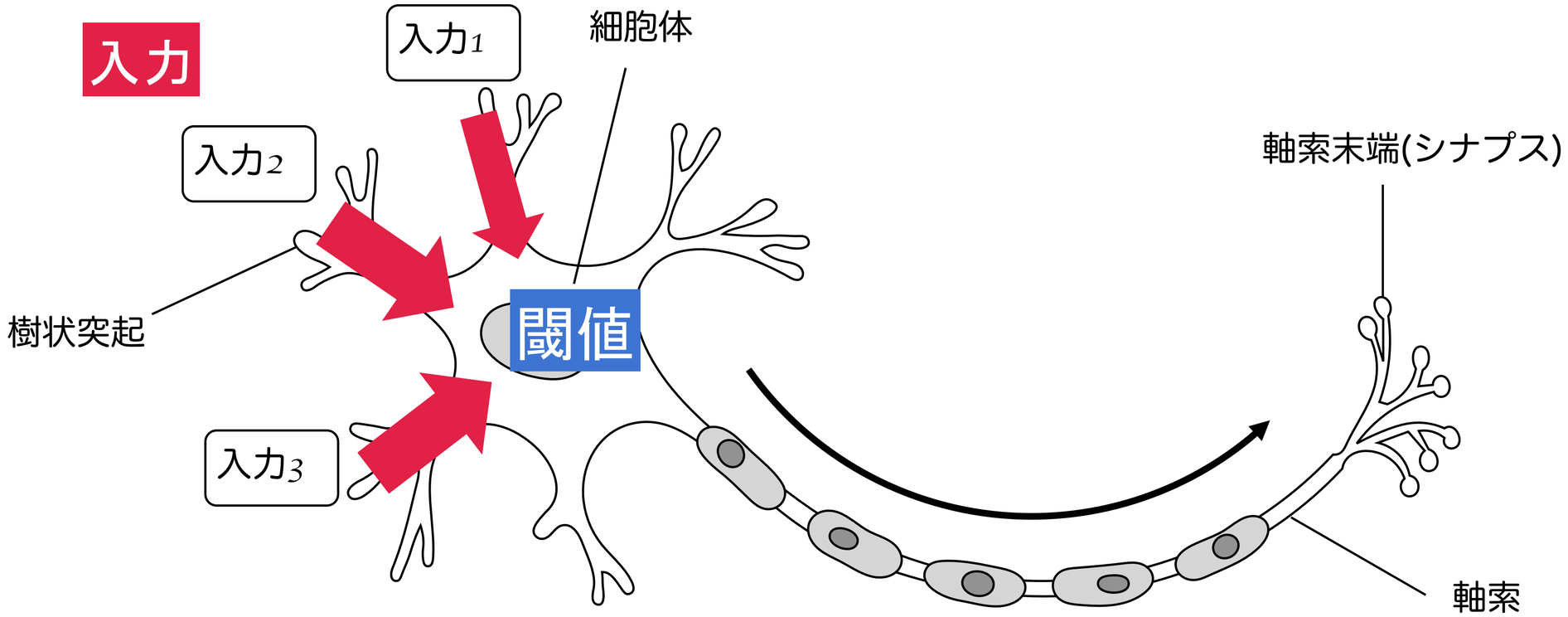
ニューロンとパーセプトロン

・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



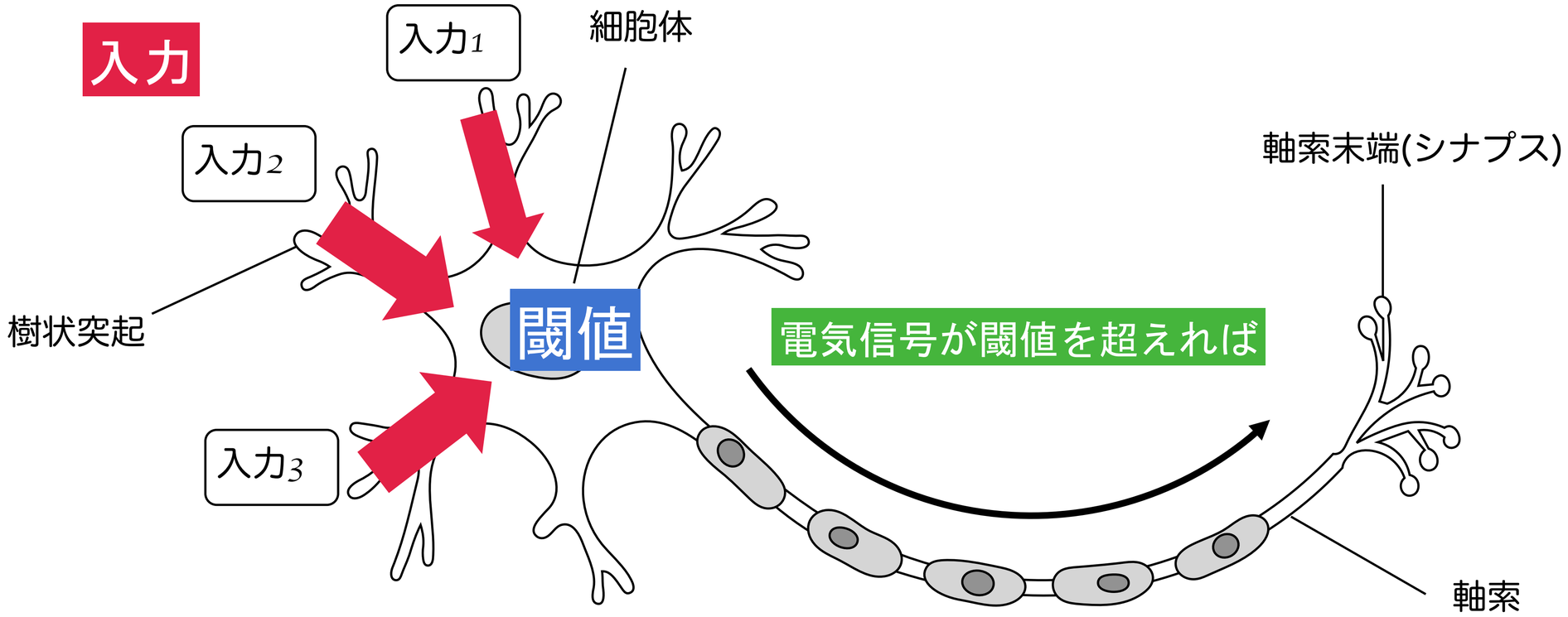
ニューロンとパーセプトロン

・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



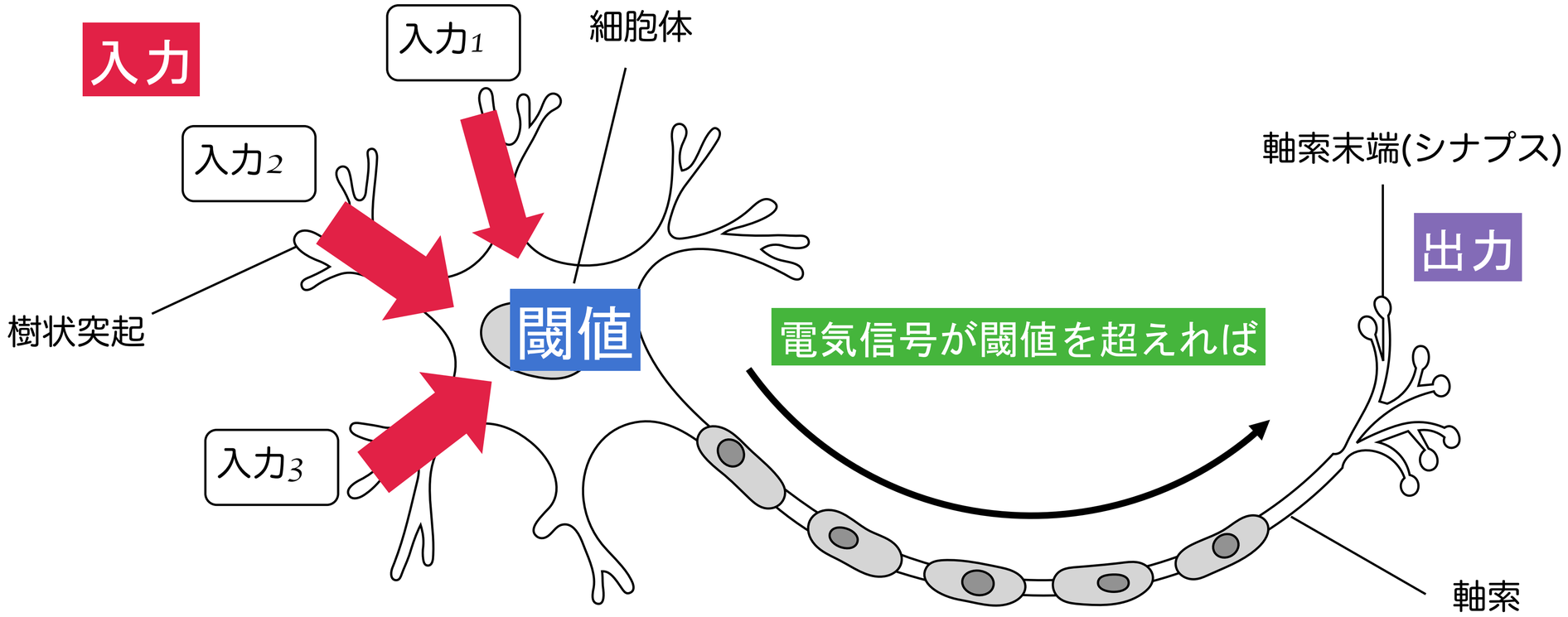
ニューロンとパーセプトロン

・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



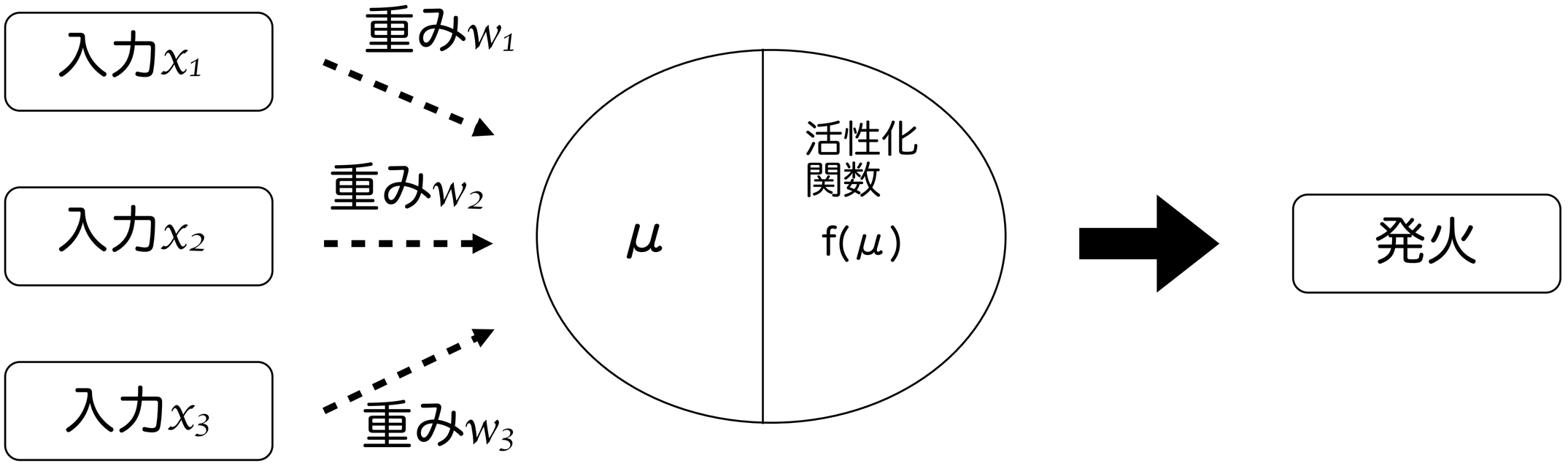
ニューロンとパーセプトロン

・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

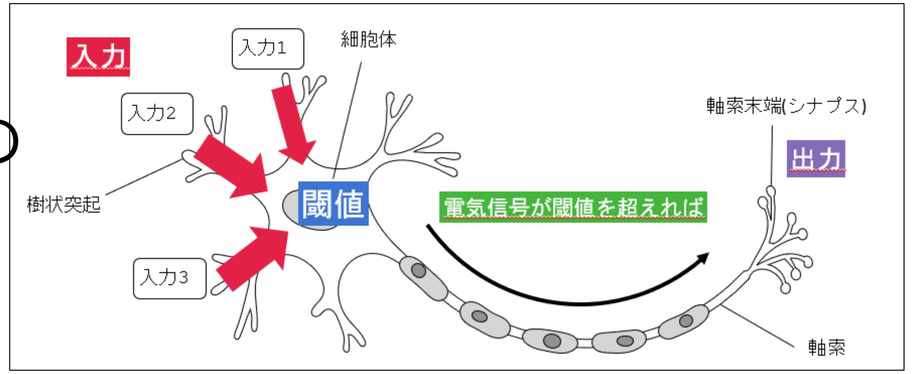
単一の人工ニューロンはこのようなモデルで表すことができる。



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)

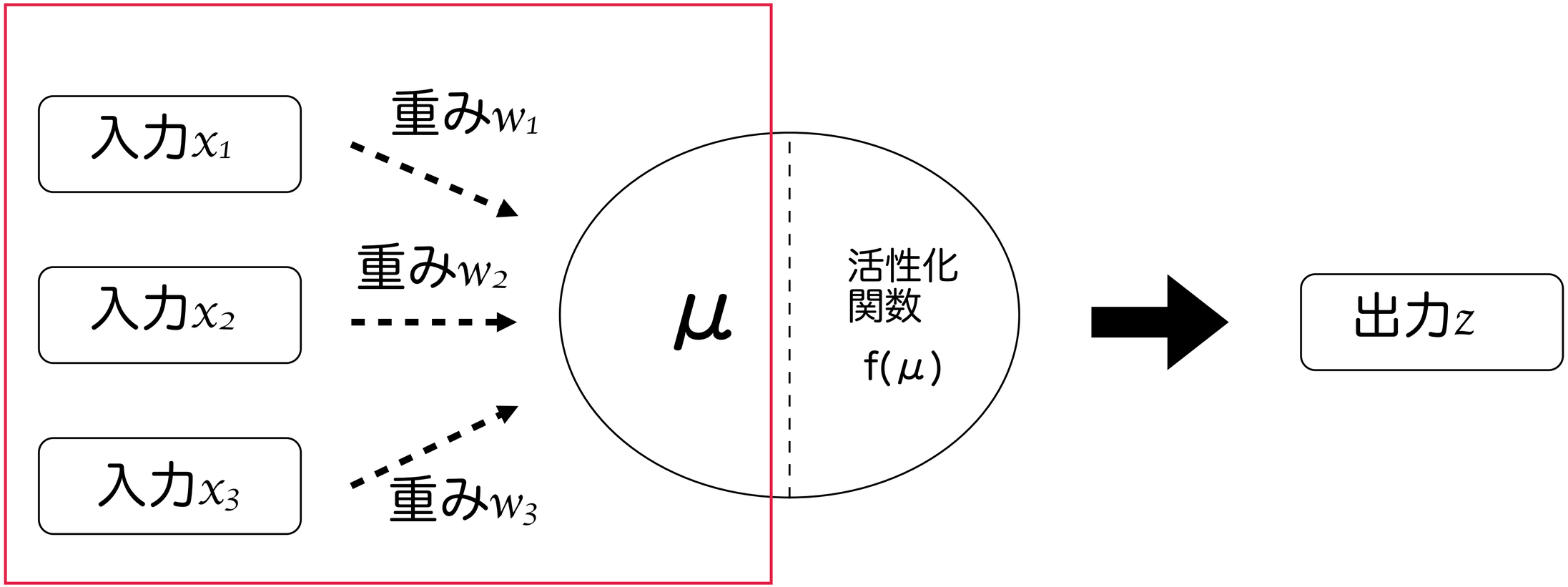
発火

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

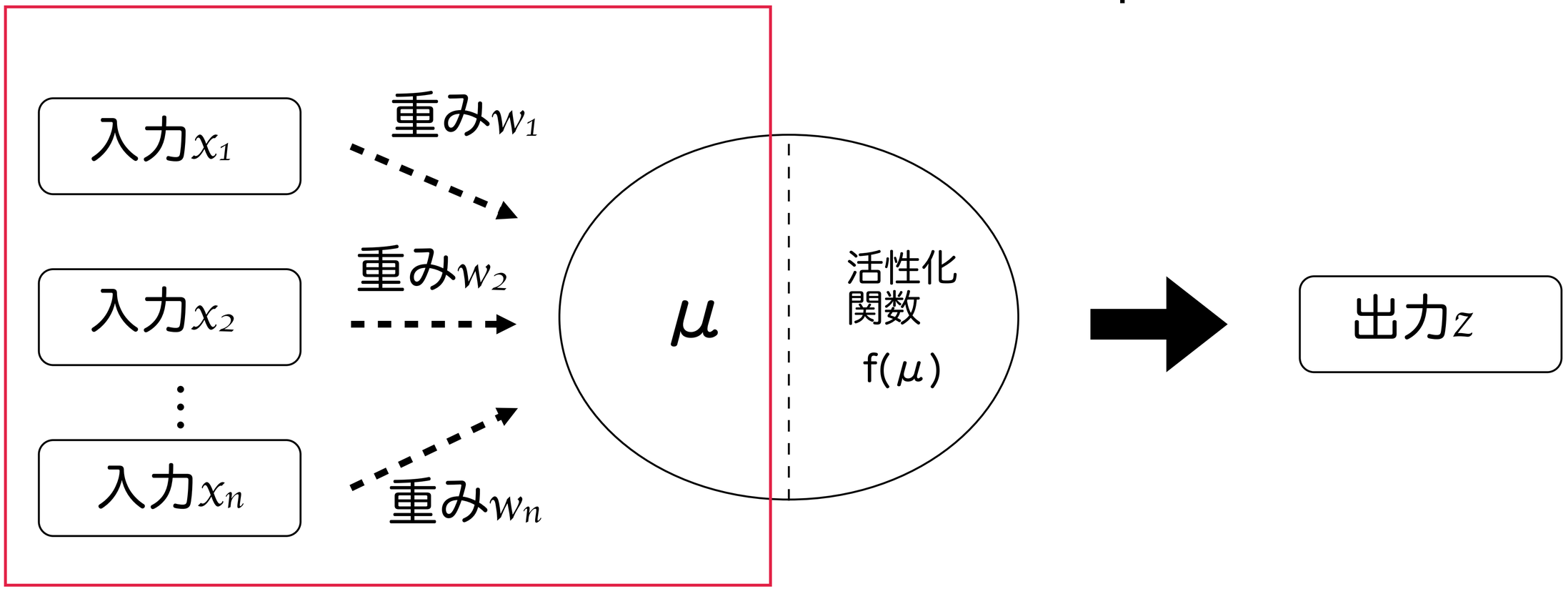
$$\mu = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$$



ニューロンとパーセプトロン

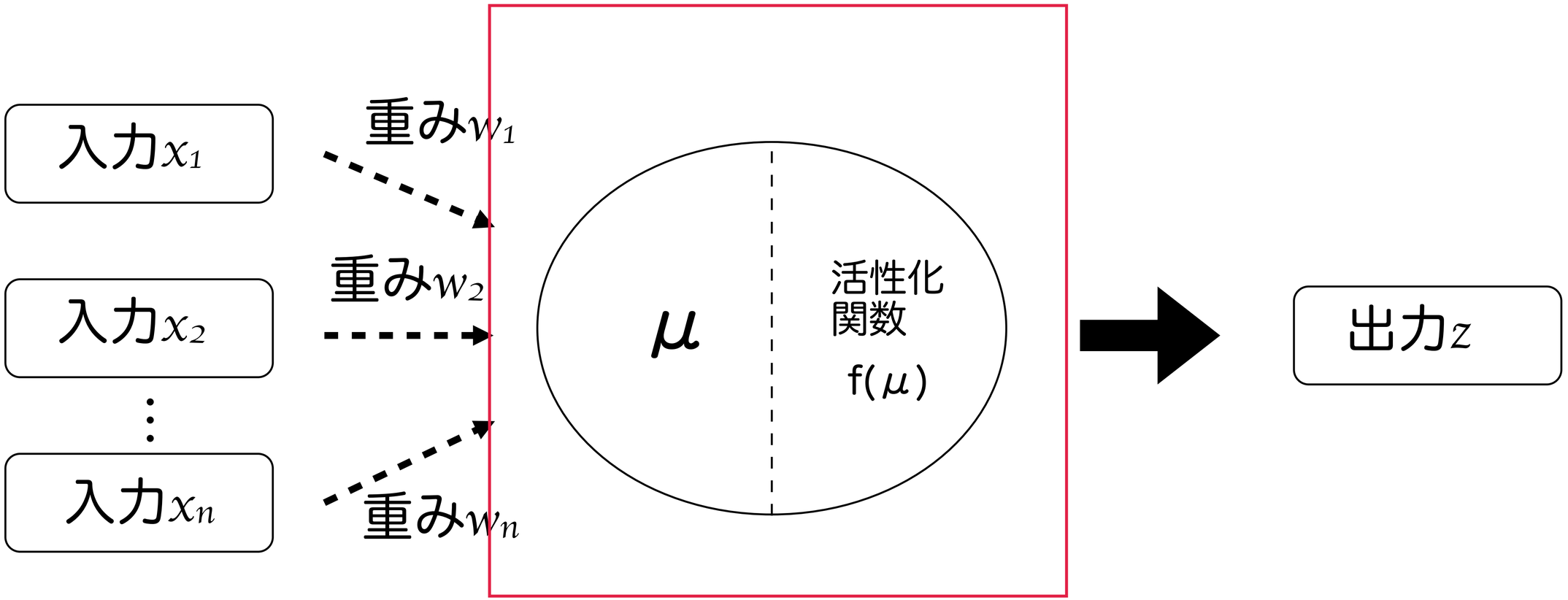
入力がn個あった場合は下のように一般化できる

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



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

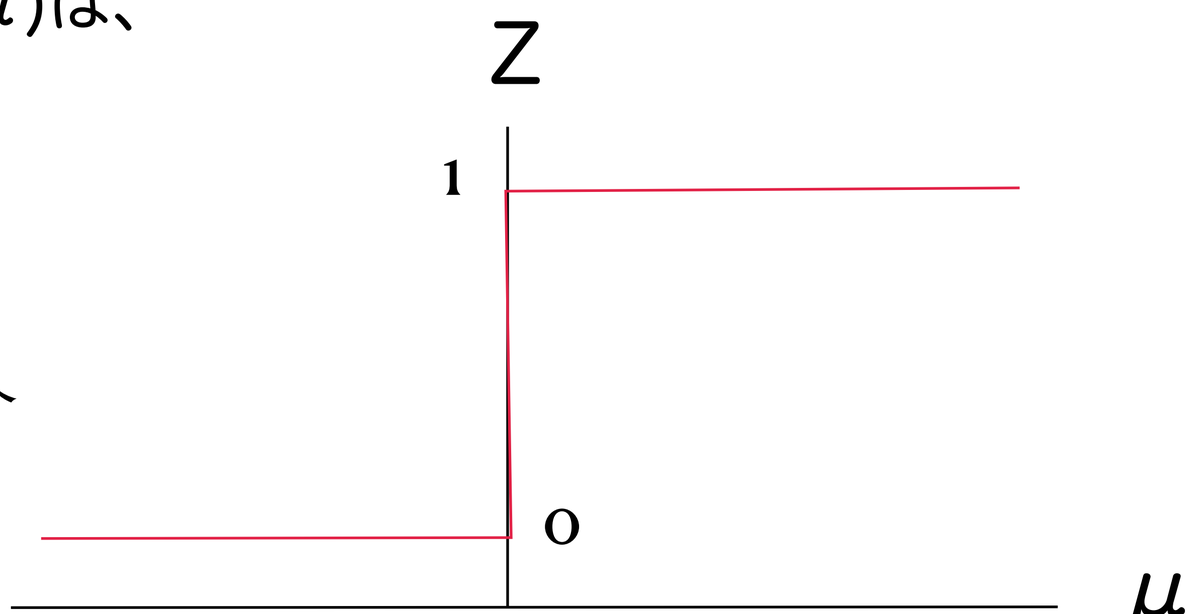
- ステップ関数
- 恒等関数
- シグモイド関数
- tanh関数
- ReLU関数
- ソフトプラス関数
- Leaky ReLU
- ソフトマックス関数
- PReLU / Parametric ReLU
- ELU
- SELU
- Swish関数
- Mish関数

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、

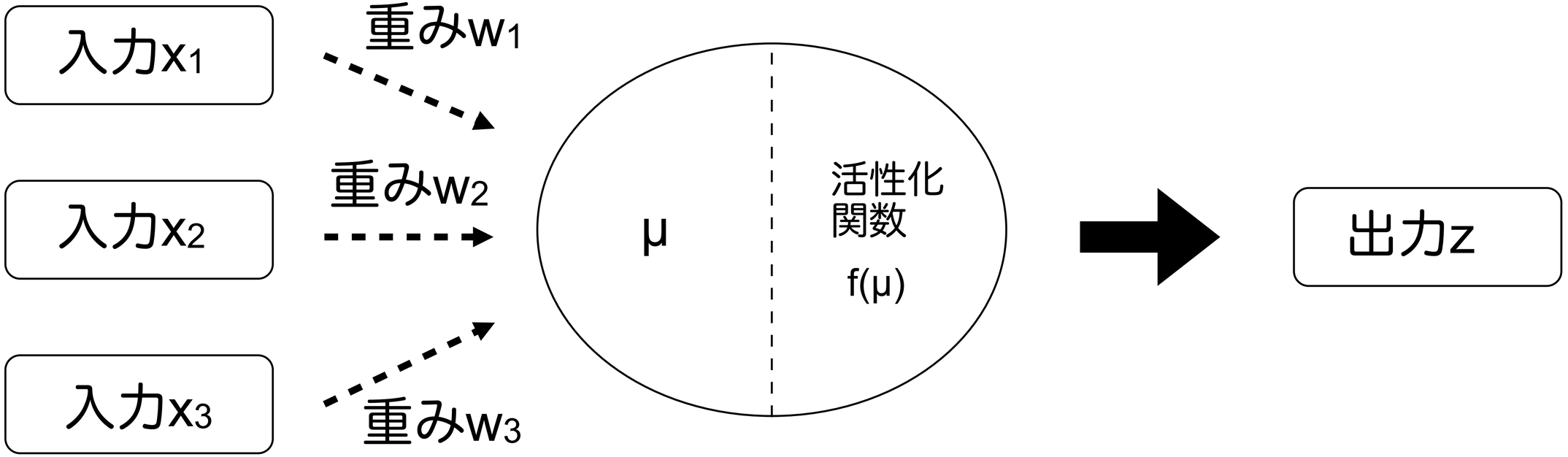
$$Z = f(\mu) = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



μがどんな値でも出力は
0か1のいずれかになる！

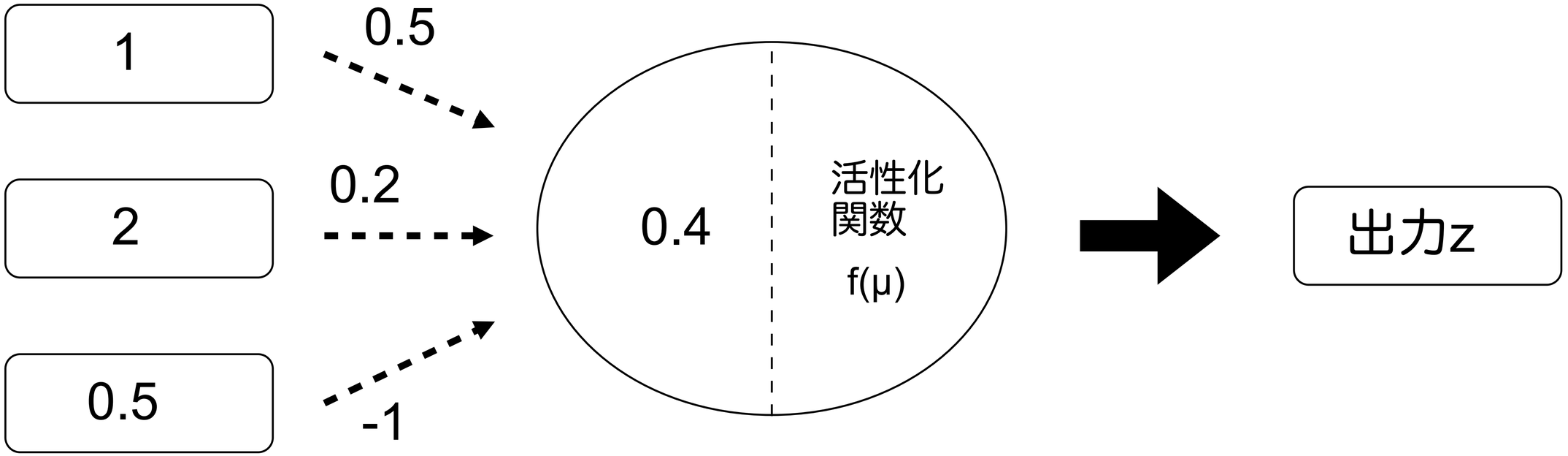
閾値を0とすると、μが0より大きければzは1となり(発火)、
0以下であれば0となる(発火しない)

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？

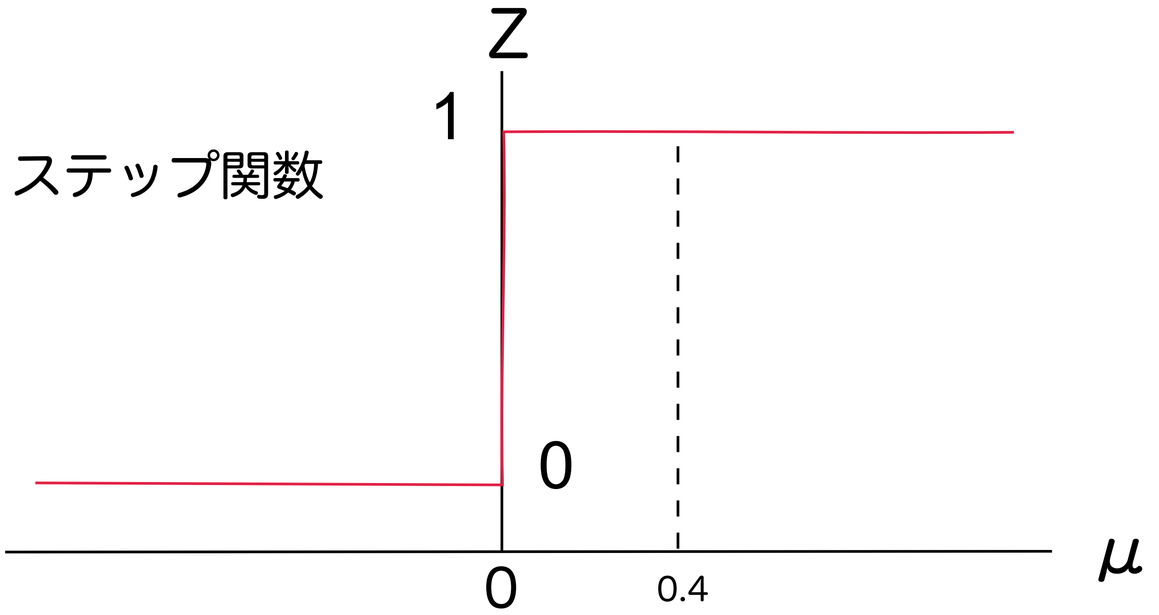


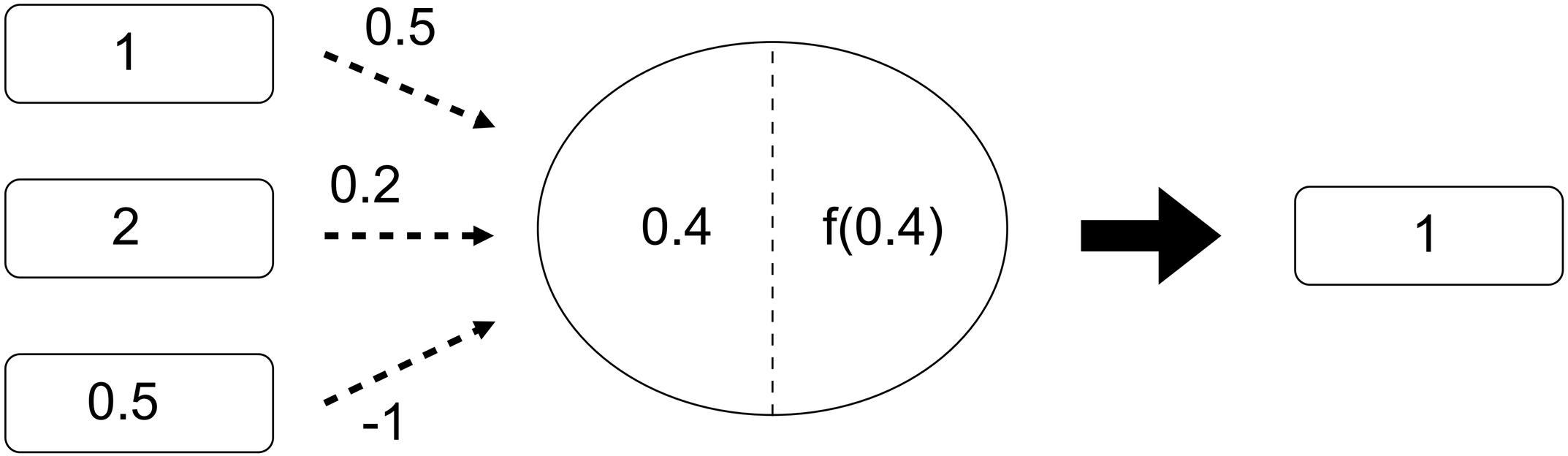
例えばx_1=1、x_2=2、x_3=0.5、w_1=0.5、w_2=0.2、w_3=-1の時は？

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

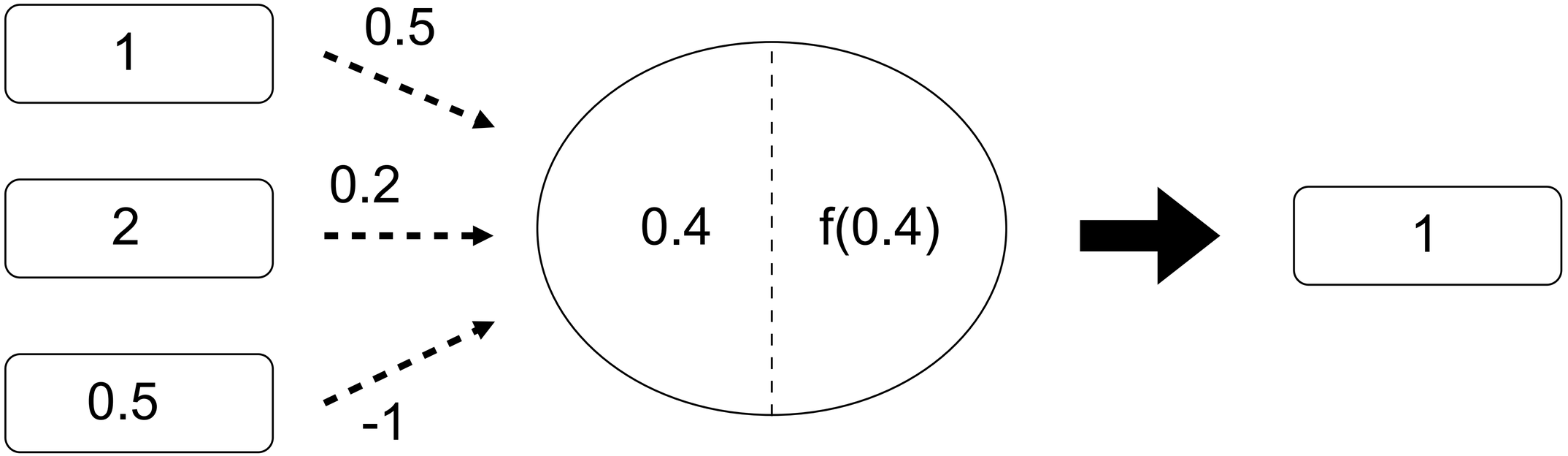


$$z = \begin{cases} 1 & \mu > 0 \\ 0 & \text{それ以外} \end{cases}$$



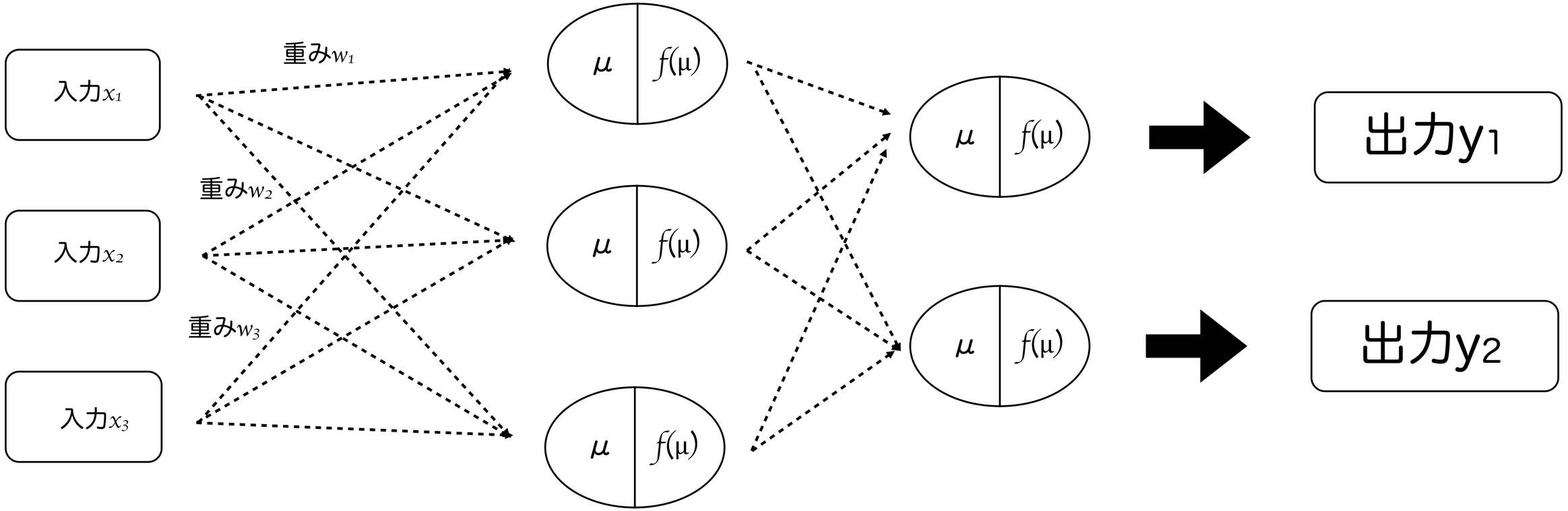


このような重みw_iや閾値を調整することできる
人工ニューラルネットワークで学習する仕組みをパーセプトロンと呼ぶ。

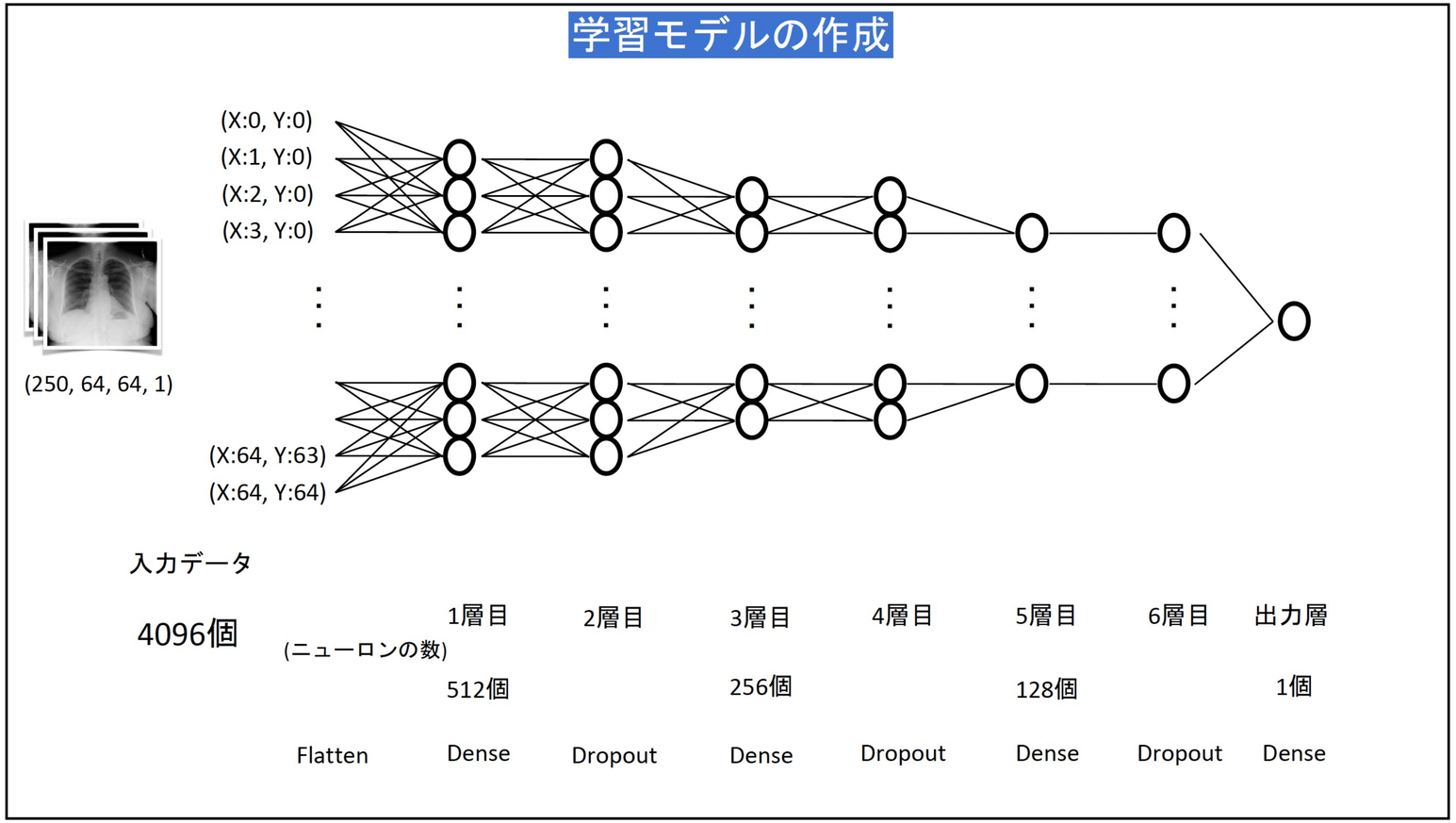


多層パーセプトロン（MLP：Multi Layer Perceptron）

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)

(FASHION-MNISTを使って)

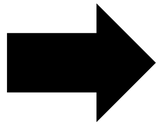
まずはシンプルなモデルで試してみましょう

```
from tensorflow.keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_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)
y_test = to_categorical(y_test)
```

x_test	Array of uint8	(10000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]
x_train	Array of uint8	(60000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]
y_test	Array of uint8	(10000,)	[9 2 1 ... 8 1 5]
y_train	Array of uint8	(60000,)	[9 0 0 ... 3 0 5]



x_test	Array of float64	(10000, 784)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]]
x_train	Array of float64	(60000, 784)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]]
y_test	Array of float32	(10000, 10)	[[0. 0. 0. ... 0. 0. 1.] [0. 0. 1. ... 0. 0. 0.]]
y_train	Array of float32	(60000, 10)	[[0. 0. 0. ... 0. 0. 1.] [1. 0. 0. ... 0. 0. 0.]]

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

60000個　60000個　10000個　10000個　fashion_mnistのdataを読み込む


9

0

0

3

0

5


9

2

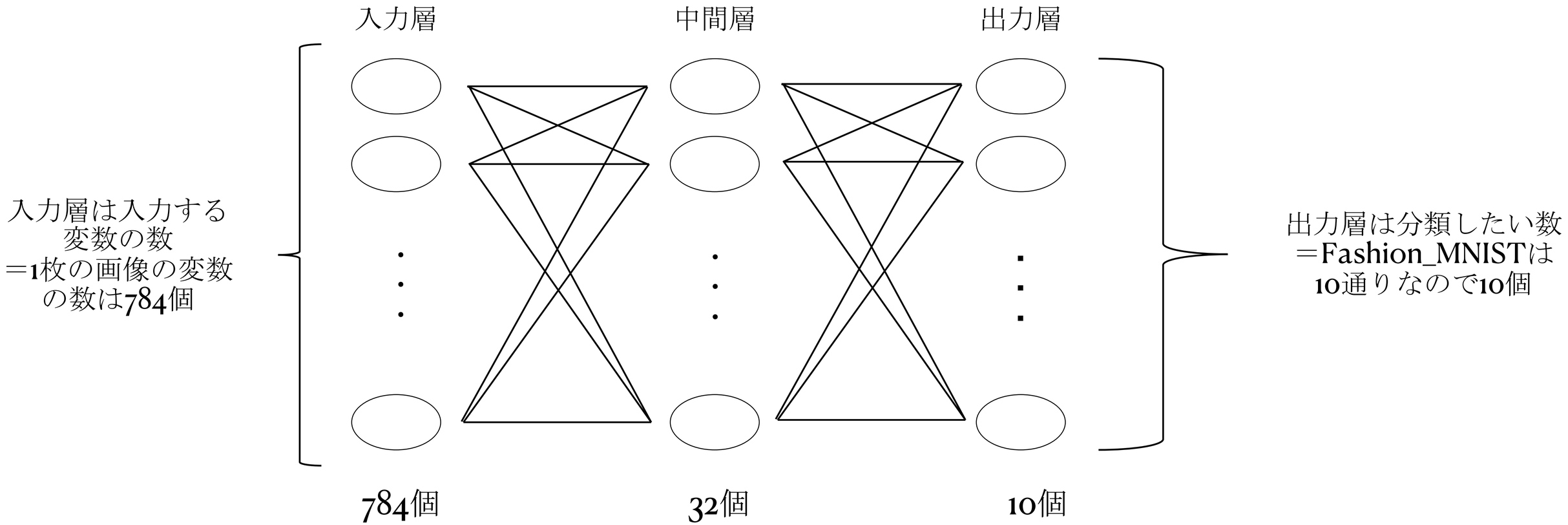
1

5

10種類の画像に
分類したい

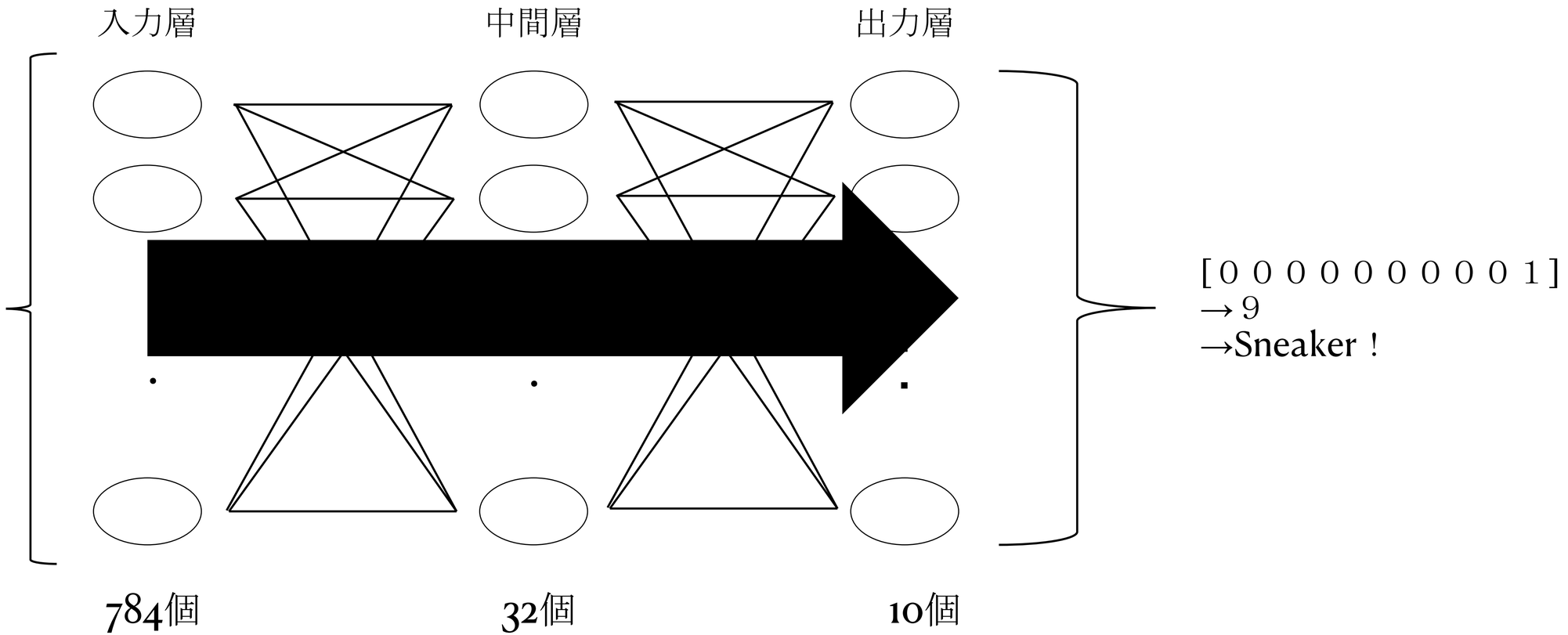
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

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 tensorflow.keras.models import Sequential
from tensorflow.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()
```



[0 0 0 0 0 0 0 0 0 1]
→9
→Sneaker !

```
from tensorflow.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 tensorflow.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'))
```

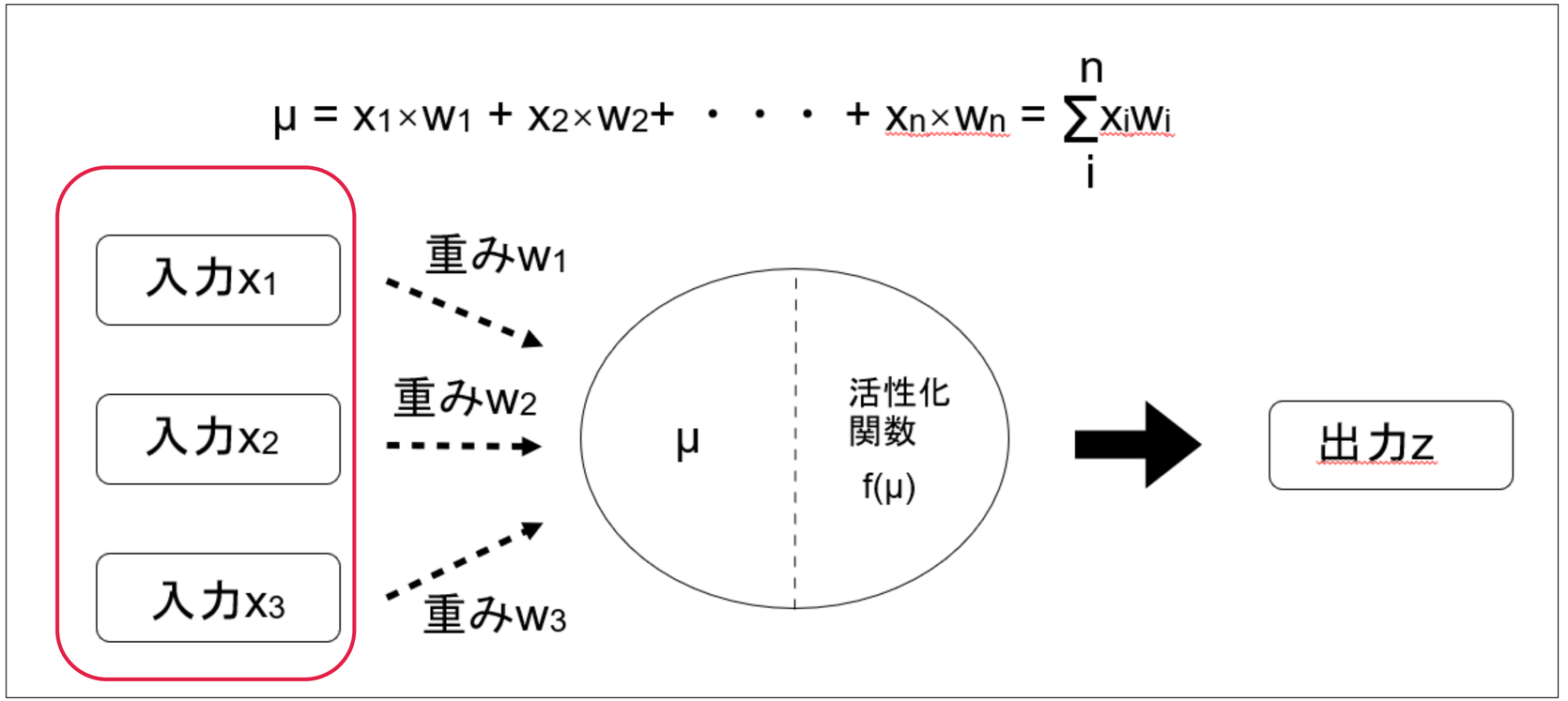
784個

0.53

0.24

0.88

0.34



0.11

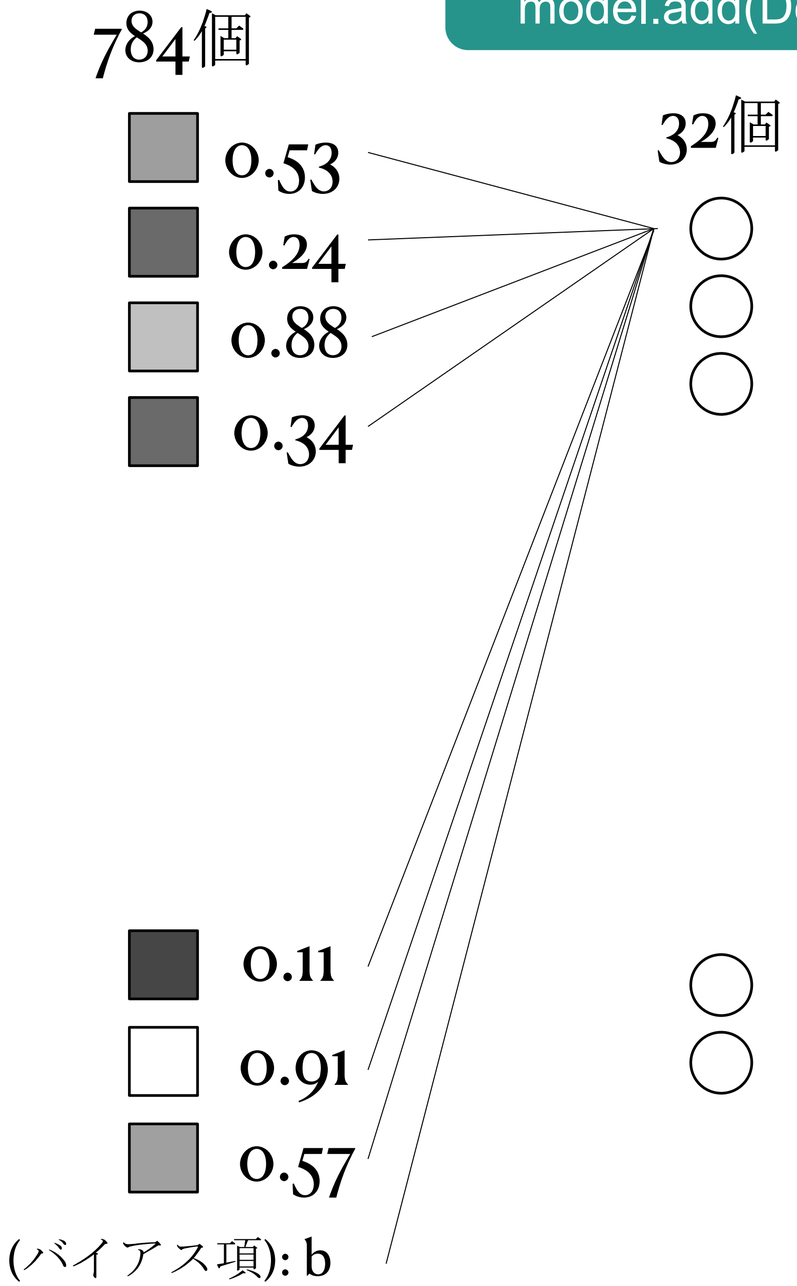
0.91

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'))
```

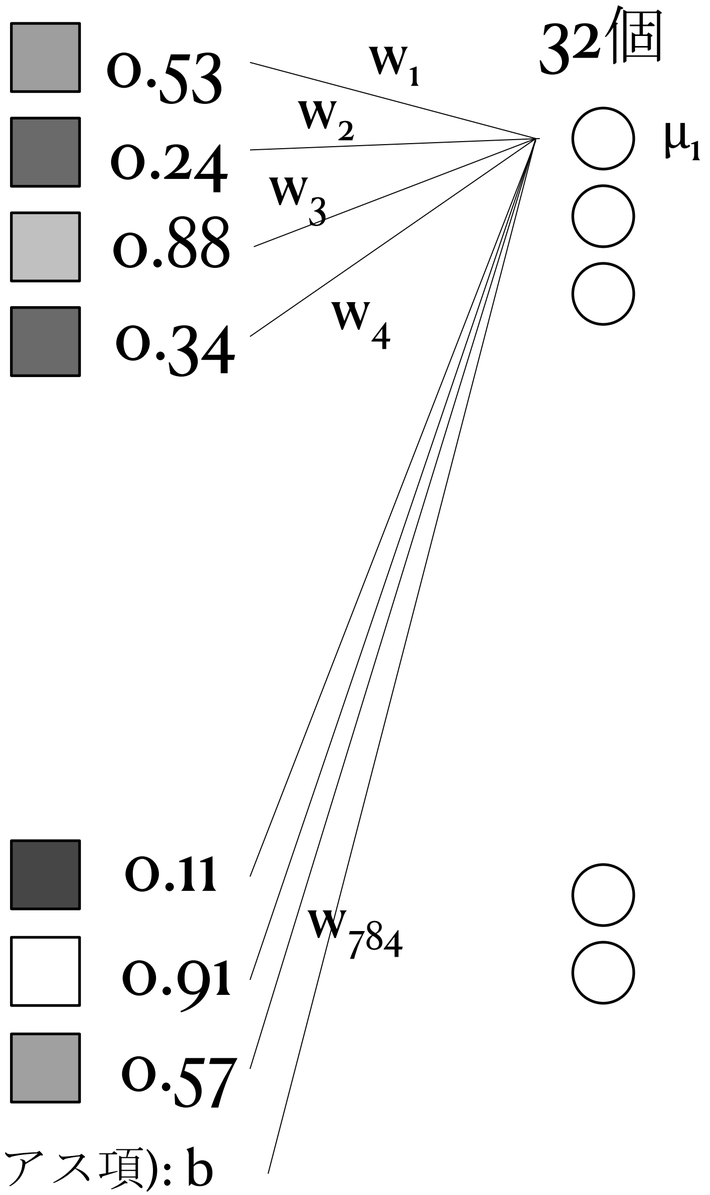


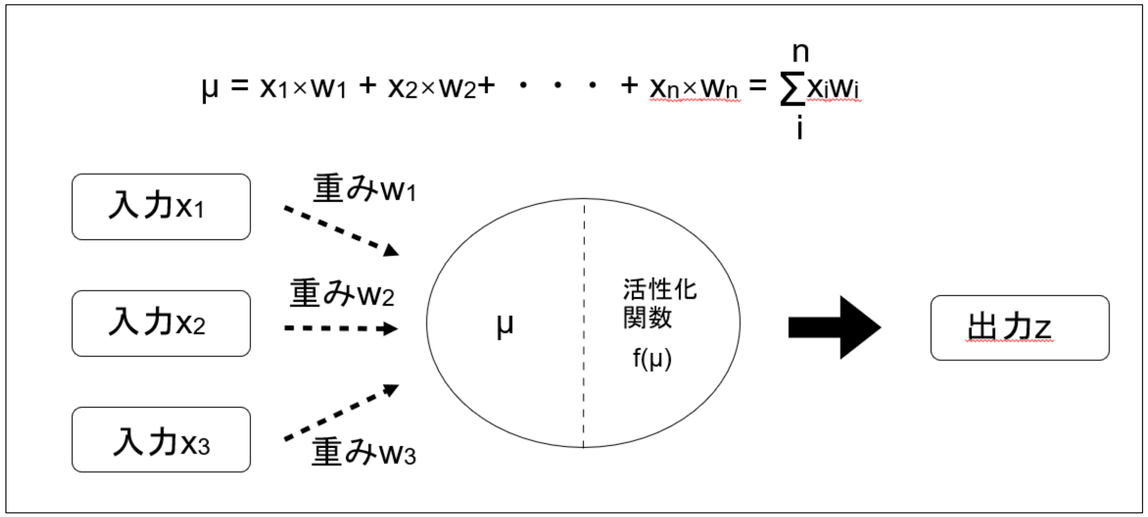
model.add()で層を追加する

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

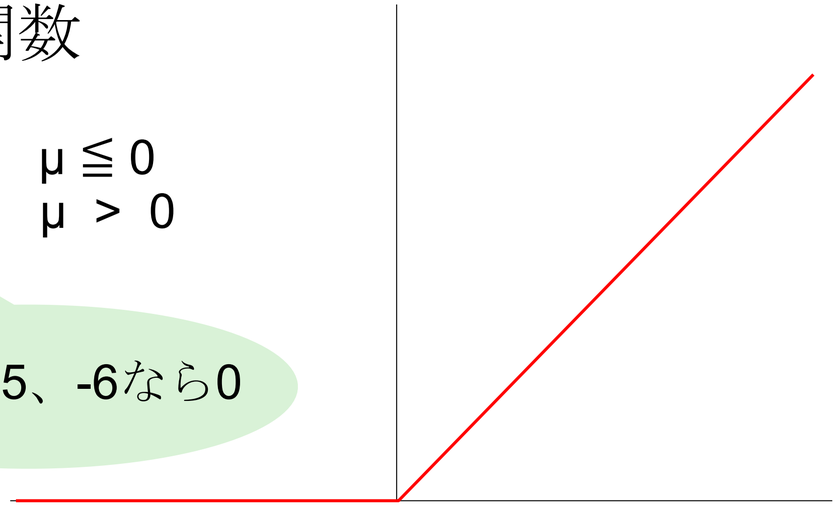
32個

(バイアス項): b

ReLU関数

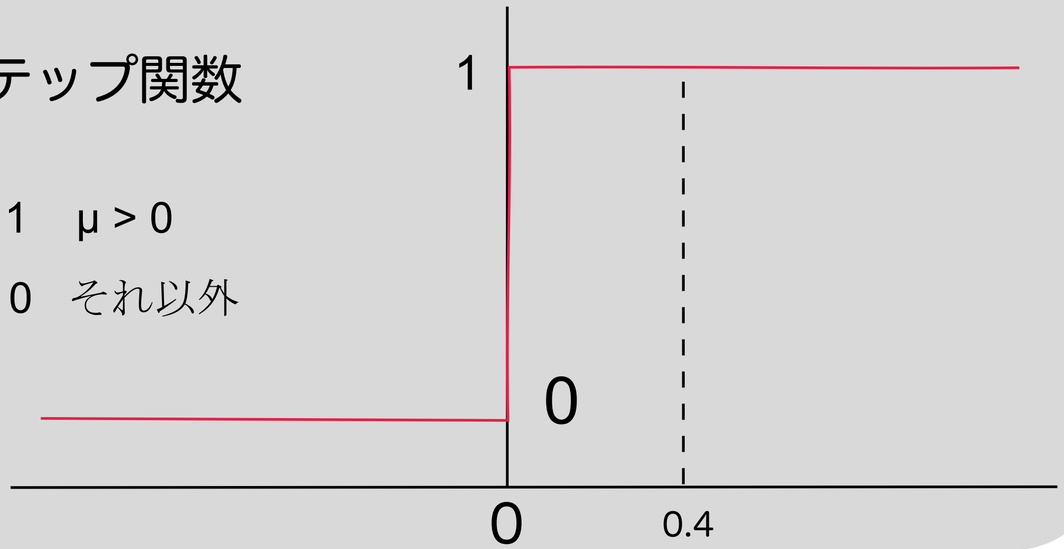
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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

(バイアス項): b

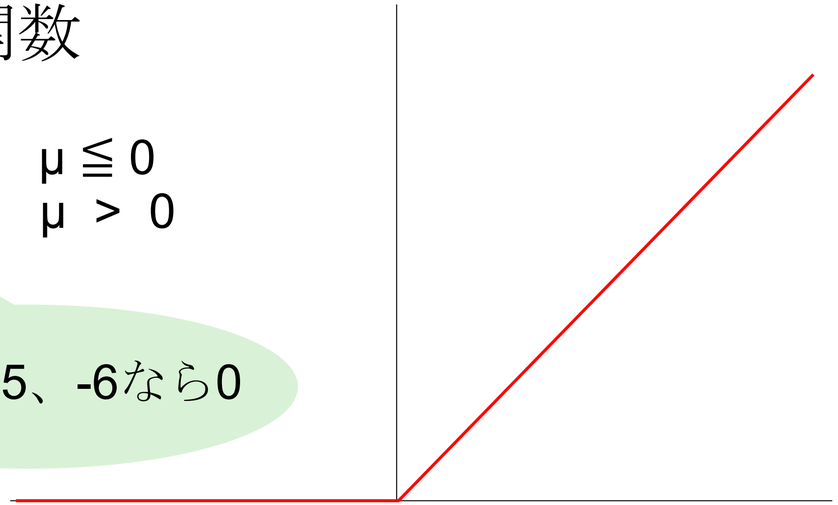
32個

3.23

ReLU関数

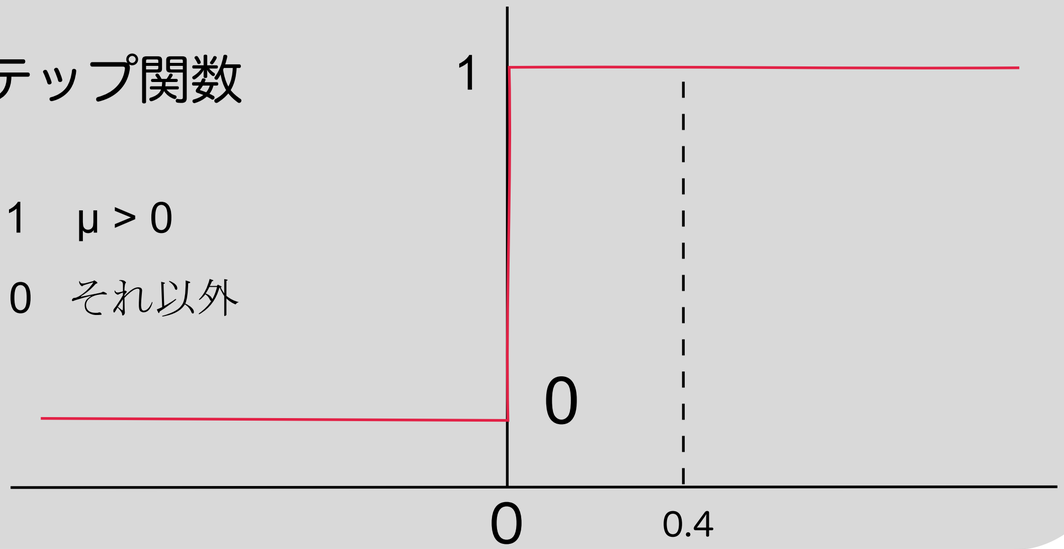
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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

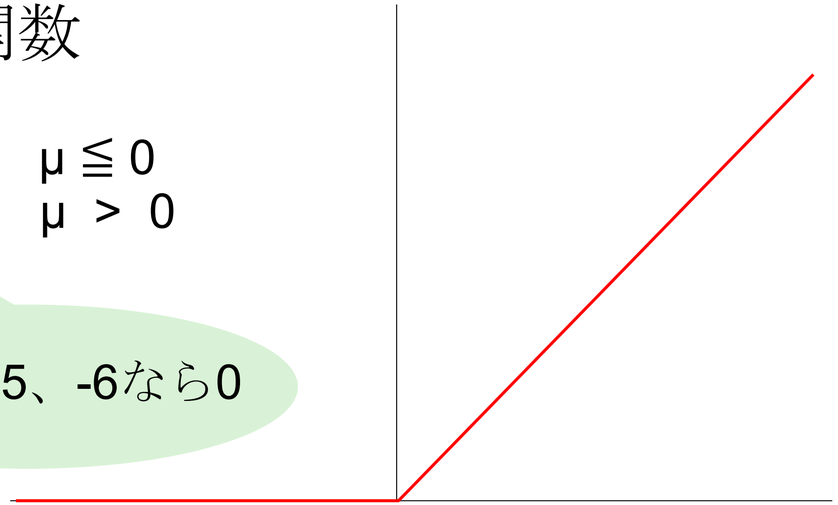
32個

3.23
0
8.45

ReLU関数

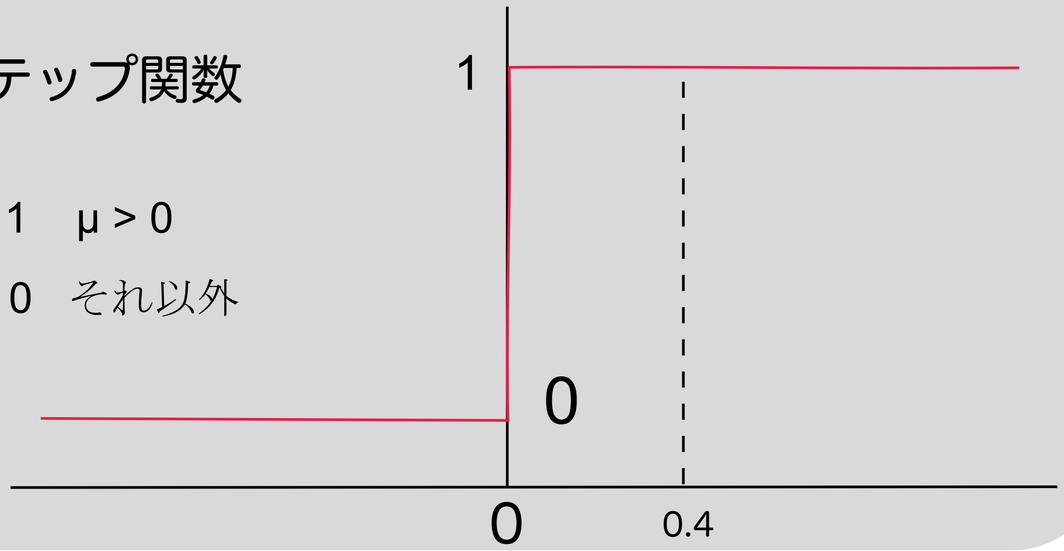
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

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

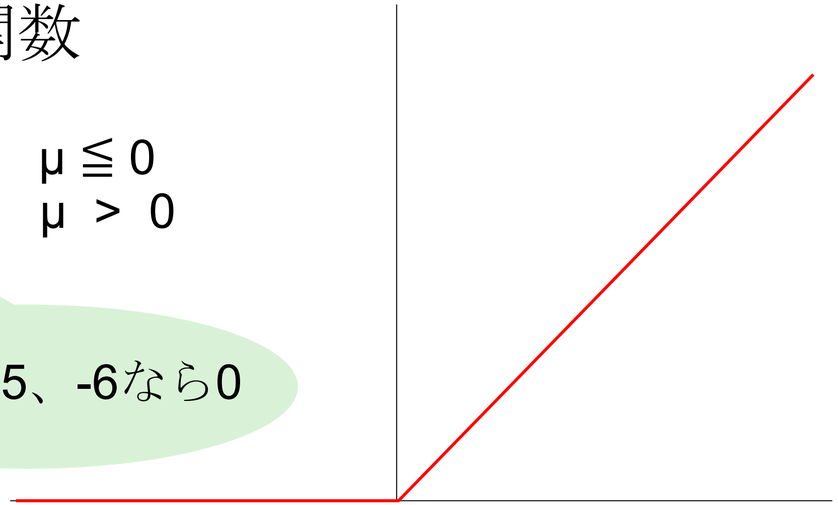
32個

3.23
0
8.45
0
4.33

ReLU関数

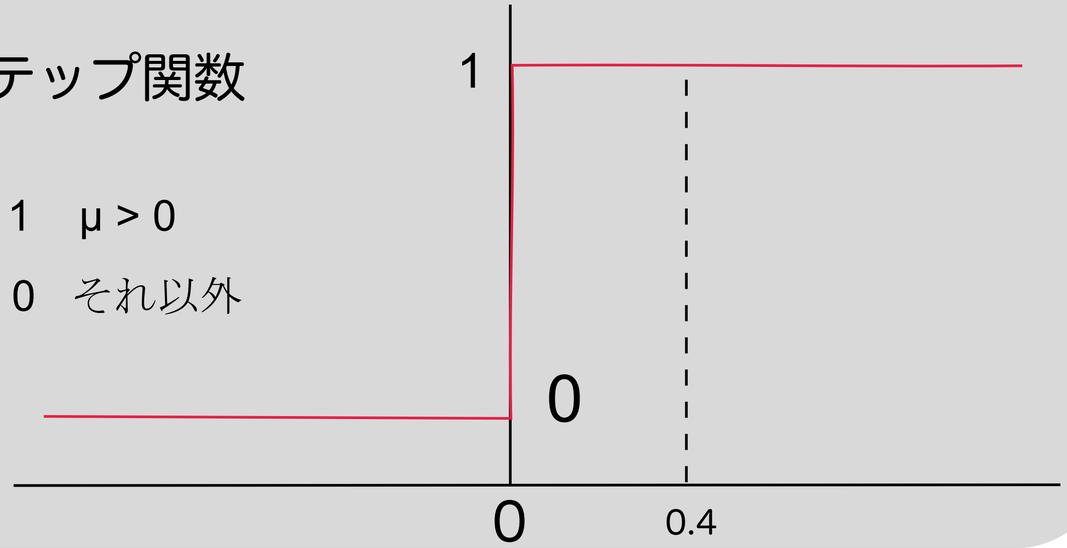
$$f(\mu) = \begin{cases} 0 & \mu \leqq 0 \\ \mu & \mu > 0 \end{cases}$$

μが5なら5、-6なら0

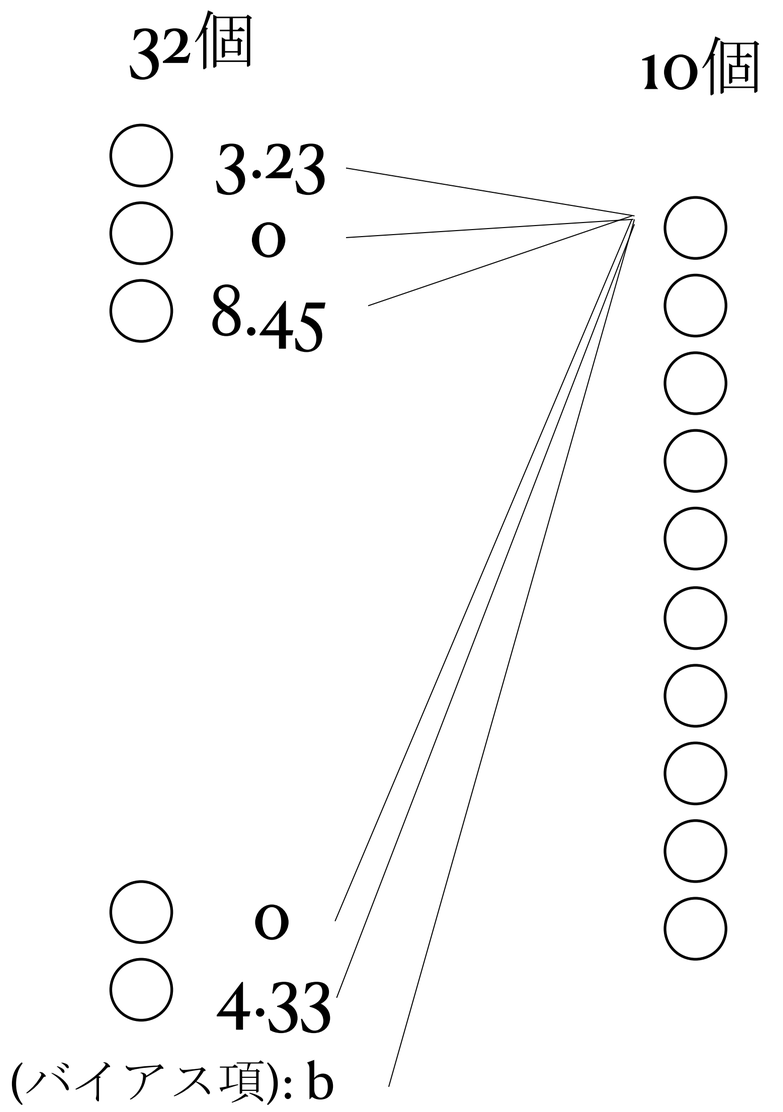


ステップ関数

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



```
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

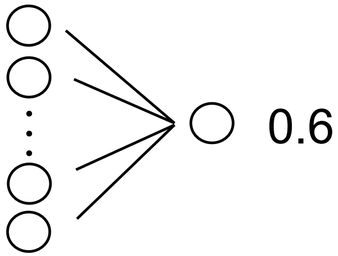
```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```

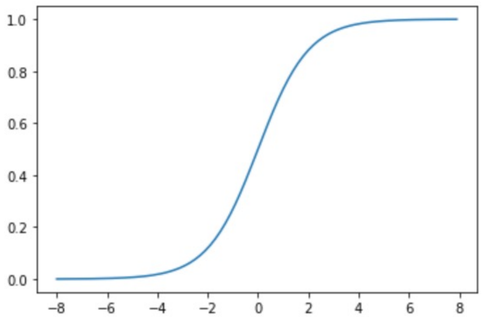
2値分類

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

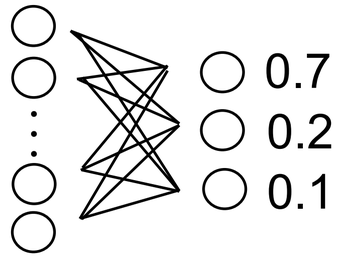
ex) 猫である確率が0.6 (=犬である確率は0.4)

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

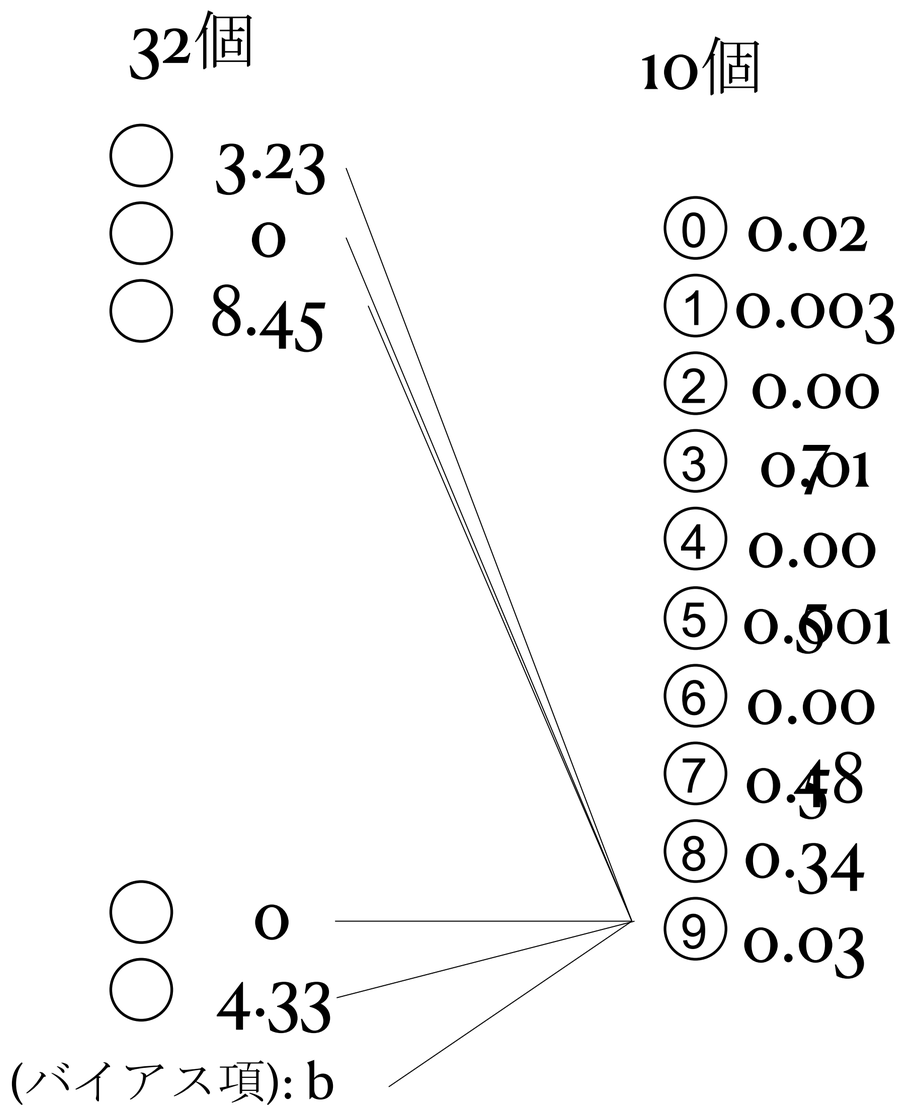
softmax関数
(activation='softmax')



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

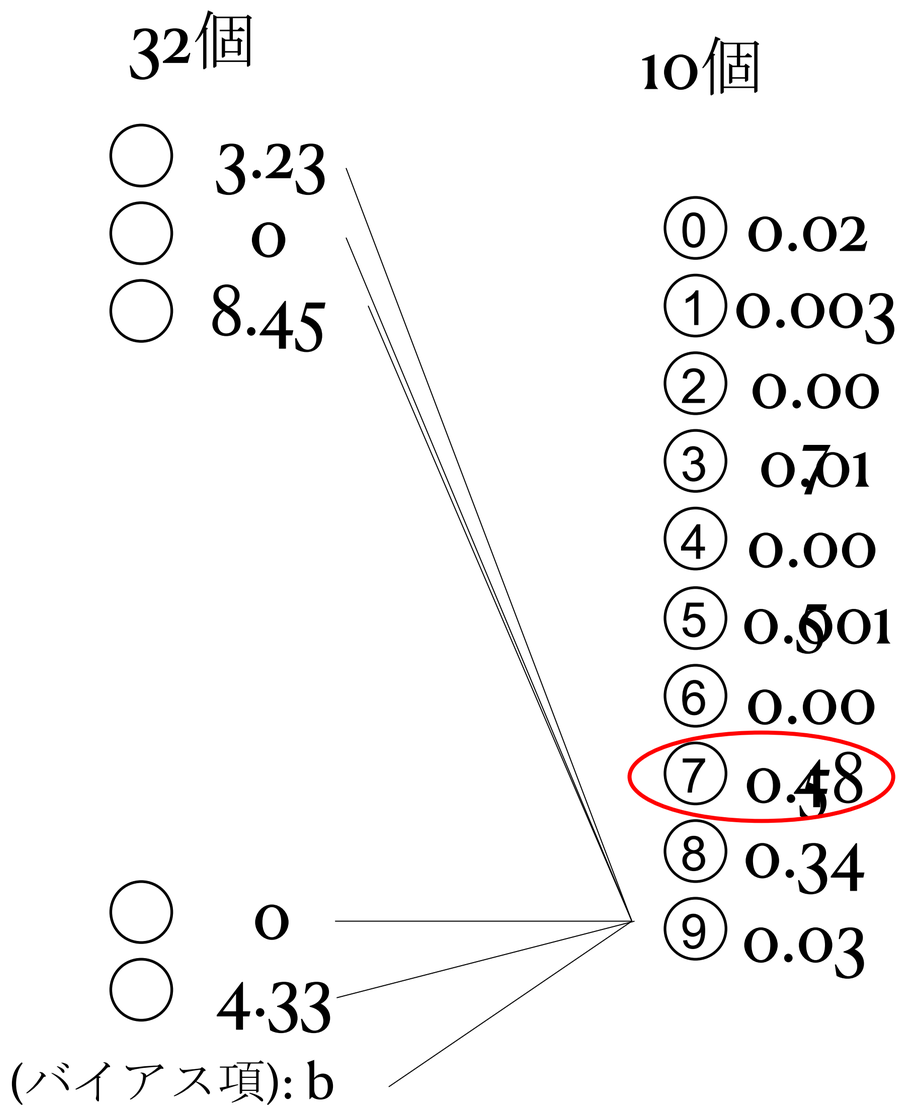
```
model.add(Dense(32, input_shape=(784,), activation='relu'))
model.add(Dense(10, activation='softmax'))
```



fashion_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'))
```



fashion_mnistは10クラスあるので
出力するニューロンの数は10個

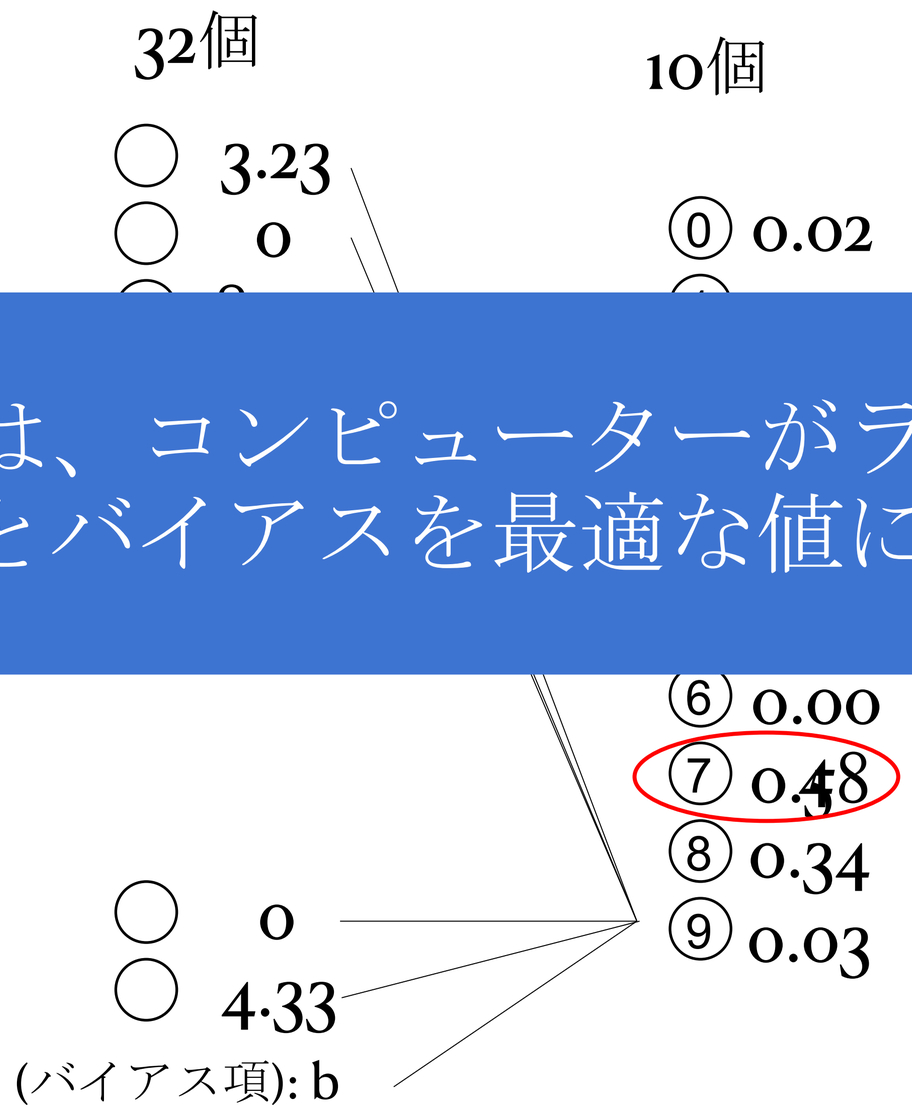
最大の確率が予測される値

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

fashion_mnistは10クラスあるので

る値

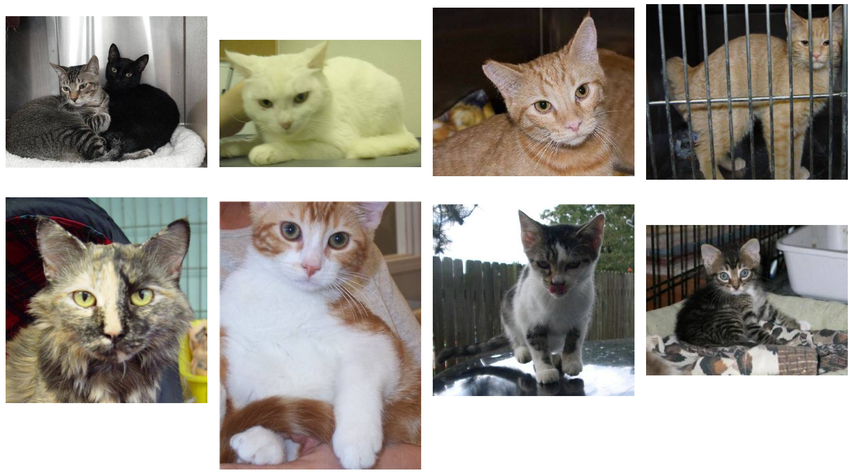
例えばこの場合は7と予測

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

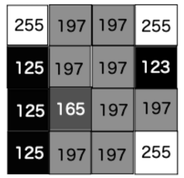
仮に猫と犬の画像の2値分類で考えると、

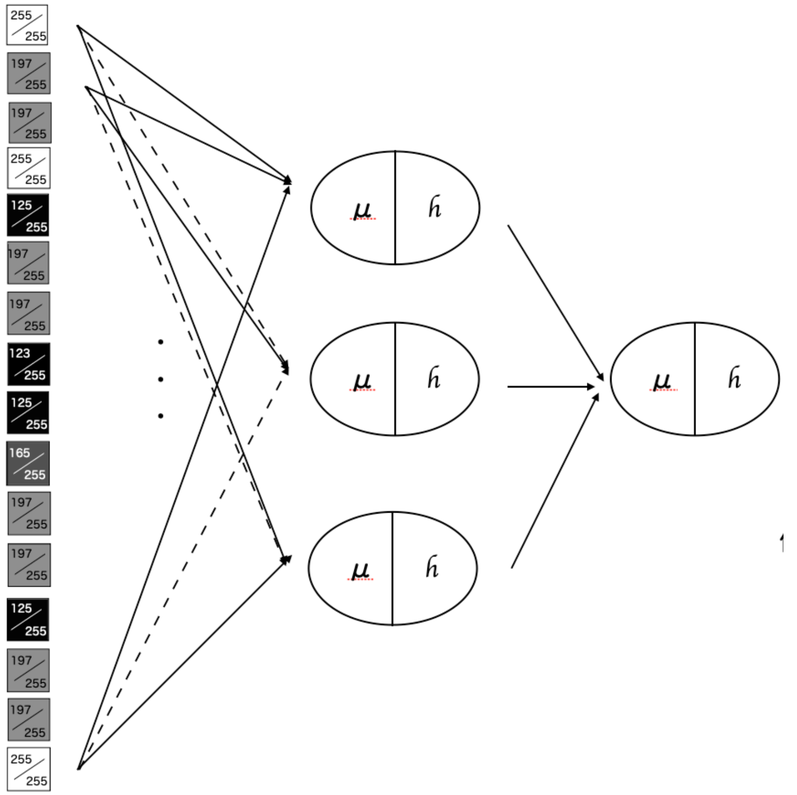


これらの画像の正解は猫(=1とする)

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)



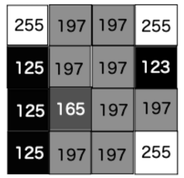


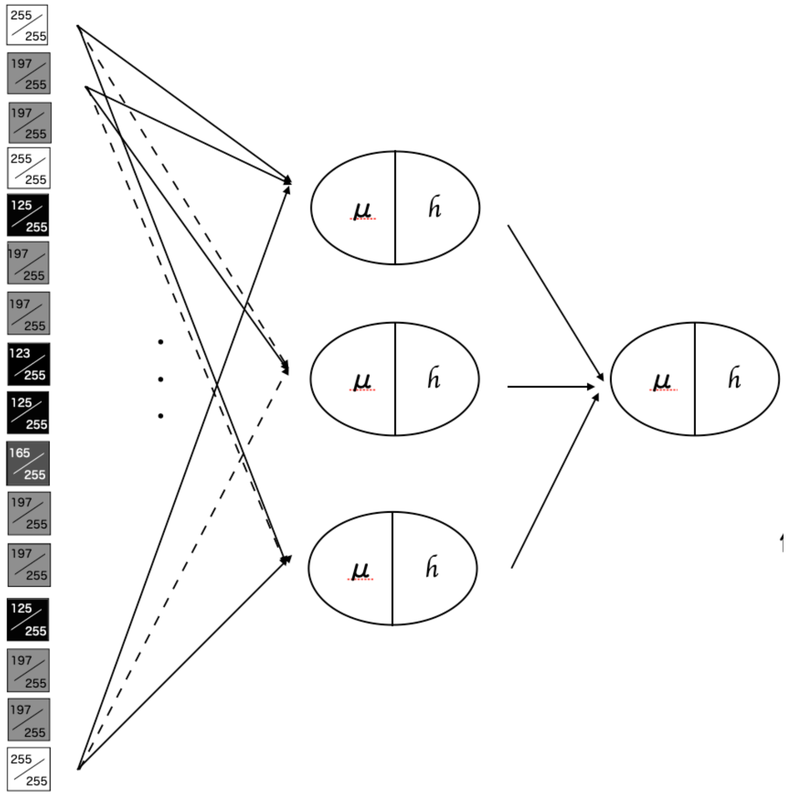
猫が１，犬が０

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)





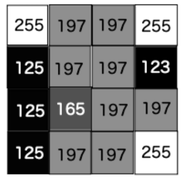
猫が１，犬が０

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

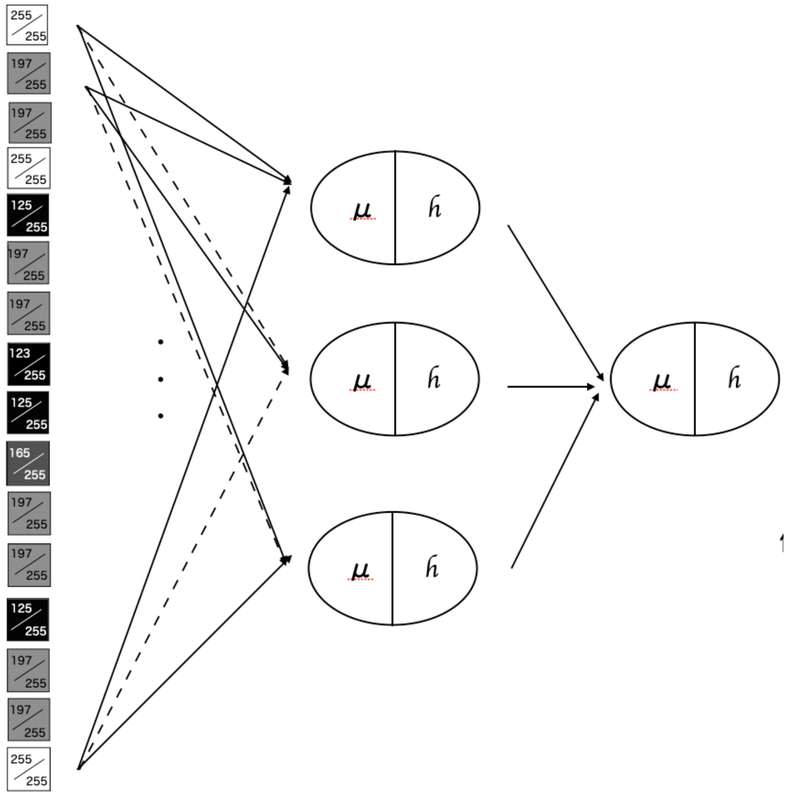
最初はテキトーな重みwとバイアスbなので正解にならない
（＝確率が低い）

損失関数を用いて出力と正解の間の誤差を計算する



2値分類の時はbinary crossentropy

多値分類の時はcategorical crossentropy



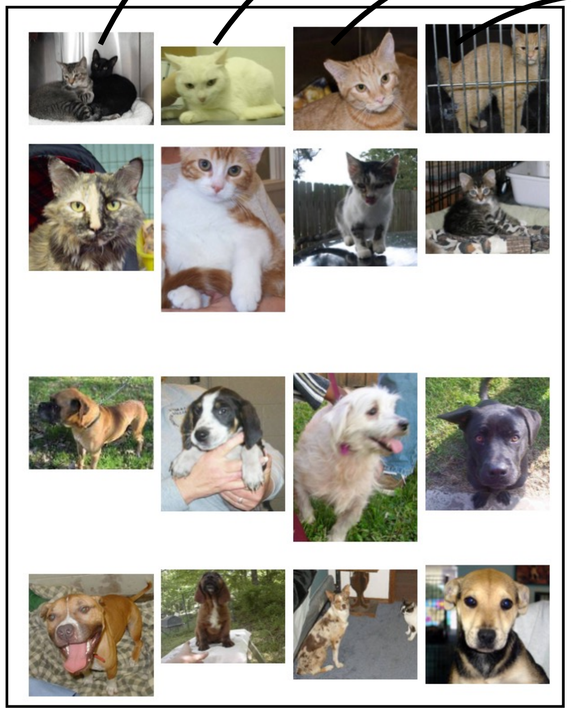
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

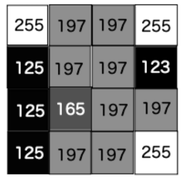
誤差E = 0.8
(だったとします)

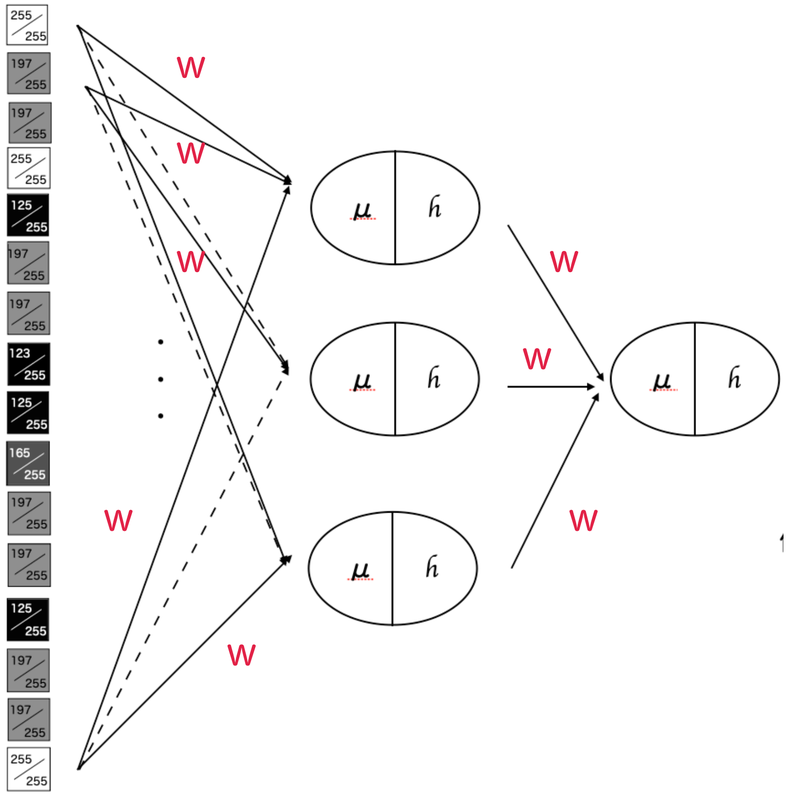
(損失関数はまた次回以降に再度説明します)

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





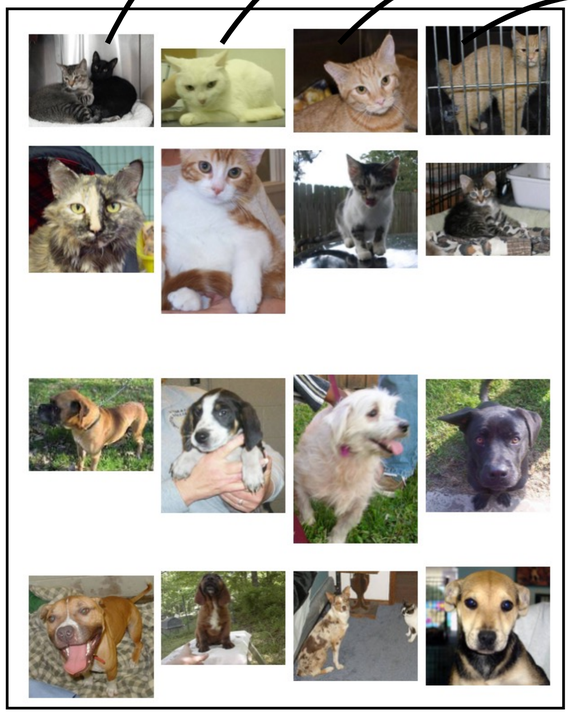


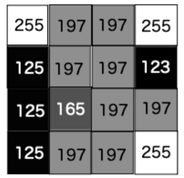
No.	出力	正解
1	0.5	1
2	0.3	1
3	0.6	1
4	0.7	1
5	0.5	0
6	0.3	0
7	0.7	0
8	0.5	0

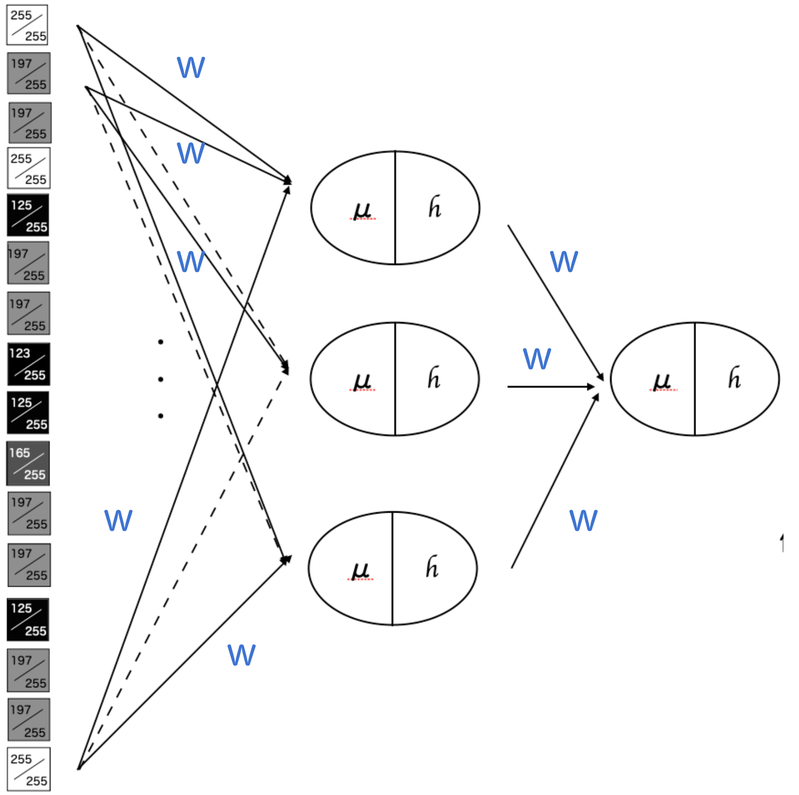
誤差E = 0.8

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

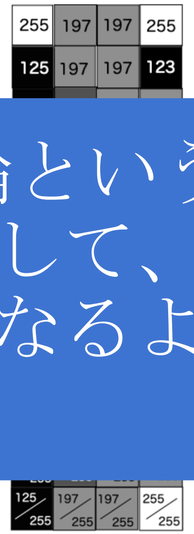
最適化関数
Adam

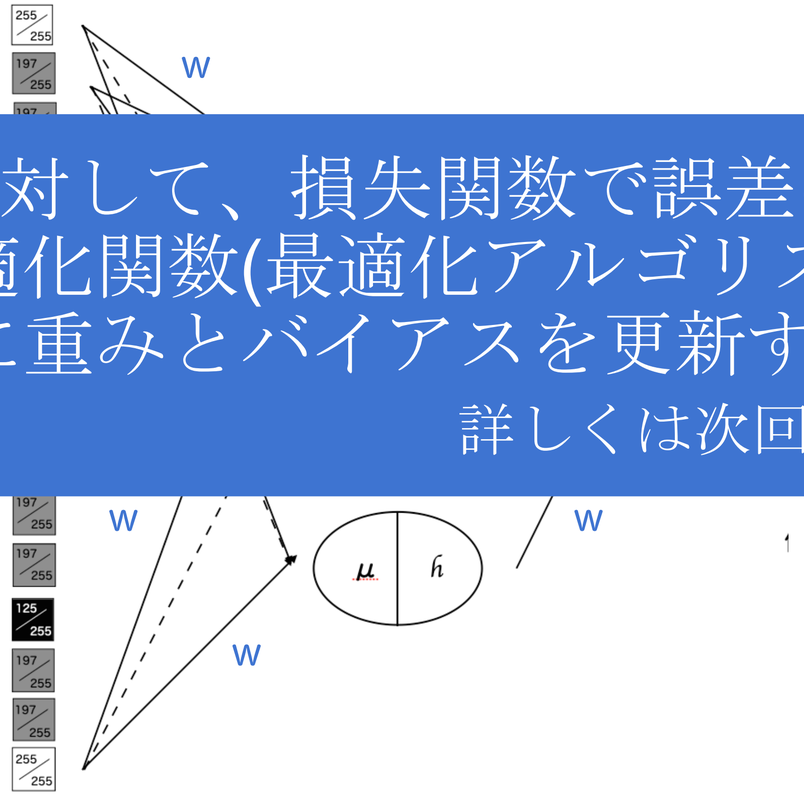
重みとバイアスを更新
各ニューロンのwとbが変わる

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

詳しくは次回以降またやります

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

```
result = model.fit(x_train, y_train, epochs=50, batch_size=64, verbose=1, validation_split=0.2, shuffle=True)
```

```
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
```

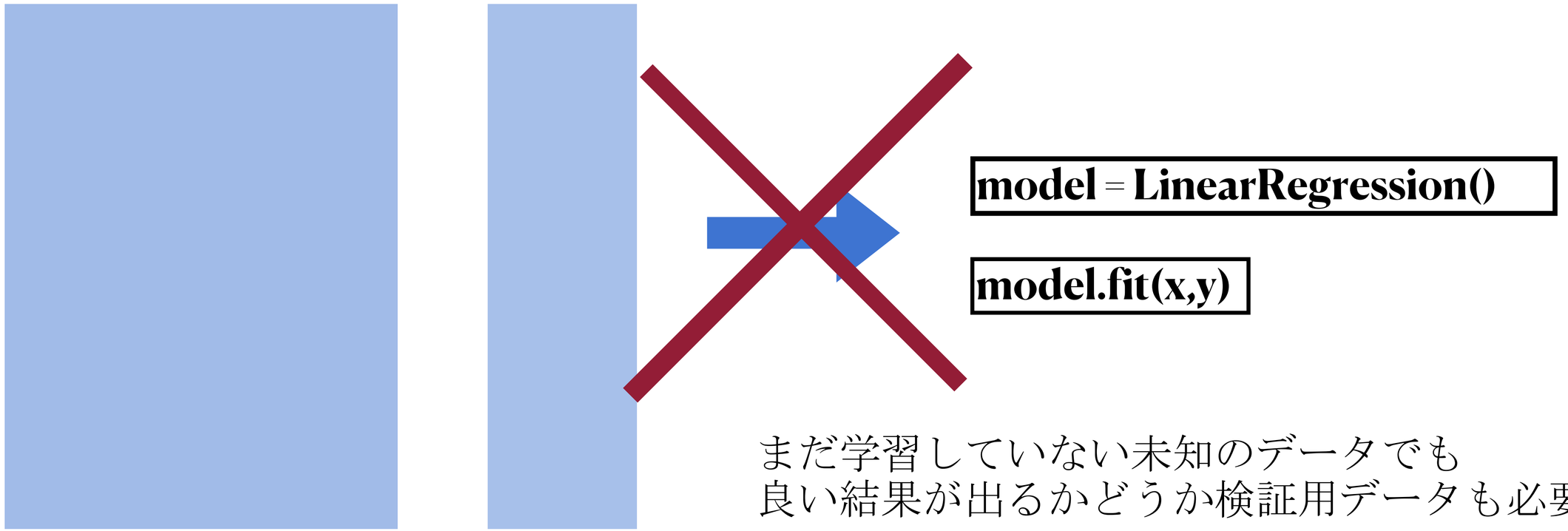
model.fit()で実際に学習が行われる

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！

→そのままだと実力よりも良すぎる正解率が出る可能性(過学習)
(偏ったデータの可能性を否定するため)

x(特徴量データ)　　y(正解データ)



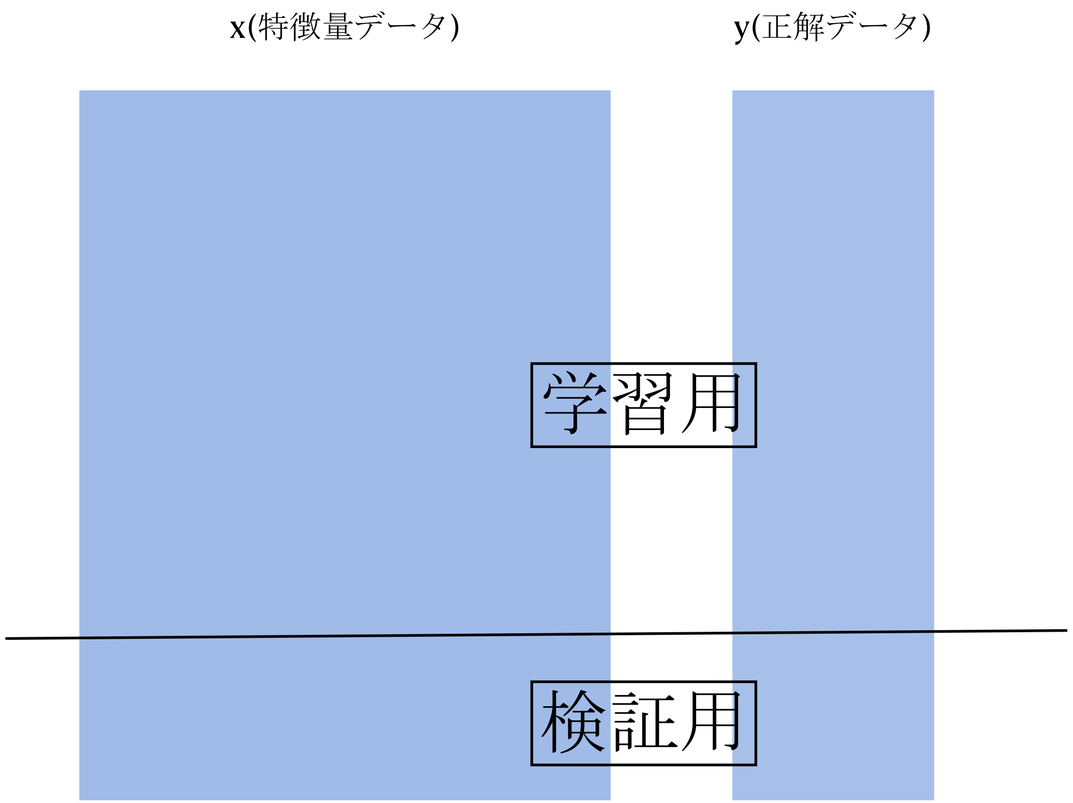
model = LinearRegression()

model.fit(x,y)

まだ学習していない未知のデータでも
良い結果が出るかどうか検証用データも必要

機械学習ではそのままデータを丸ごと学習させない！

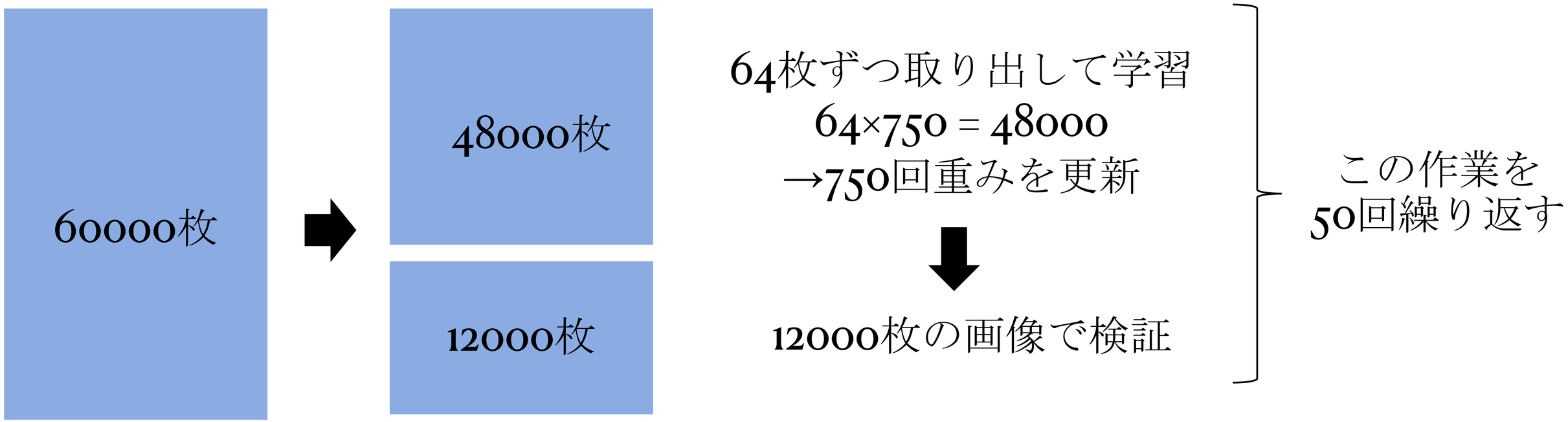
ホールドアウト法



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**)**

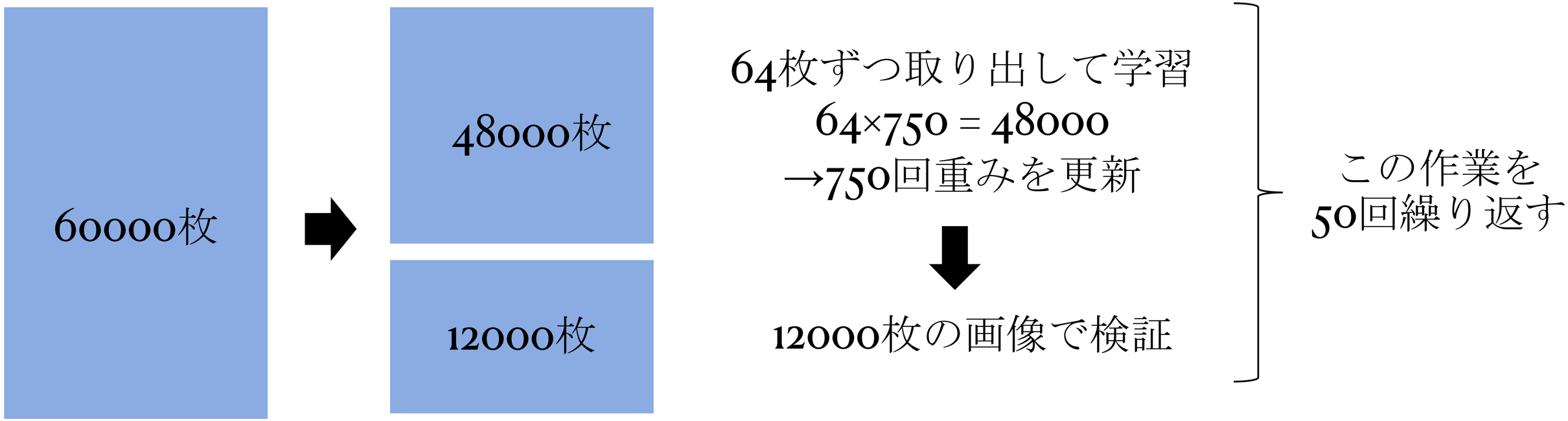
```
result = model.fit(x_train, y_train, epochs=50, batch_size=64, verbose=1,  validation_split=0.2, shuffle=True)
```

x_train : 60000枚の画像
y_train : 60000枚の画像の正解ラベル
epochs : 50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
shuffle : 学習用データを使用する際にデータをシャッフルする



```
result = model.fit(x_train, y_train, epochs=50, batch_size=64, verbose=1,    validation_split=0.2, shuffle=True)
```

```
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
```



課題

①入力層の変数の数が784、中間層のニューロンの数を８つ、出力層のニューロンの数を1とした場合、パラメータ（重みとバイアス）の総数はいくつになるか？

②入力層の変数の数が58、中間層の1つ目のニューロンの数を８つ、中間層の2つ目のニューロンの数を3つ、出力層のニューロンの数を5とした場合、パラメータ（重みとバイアス）の総数はいくつになるか？

③ fashon_mnisitの画像が60000枚の時、学習用に7割、検証用に3割、バッチサイズを128とすると、一回(1エポック)に重みを更新する回数はいくつになるか？

④画像が100000枚の時、学習用に9割、検証用に1割、バッチサイズを256とすると、一回(1エポック)に重みを更新する回数はいくつになるか？