

# Deep Learning勉強会 @ PAAK

narrative nights株式会社  
三好康祐

# TensorFlowのチュートリアルを説明

- MNIST for begginers
- Word2Vec

<https://github.com/miyosuda/intro-to-dl2>

まずこちらからソース一式をダウンロード

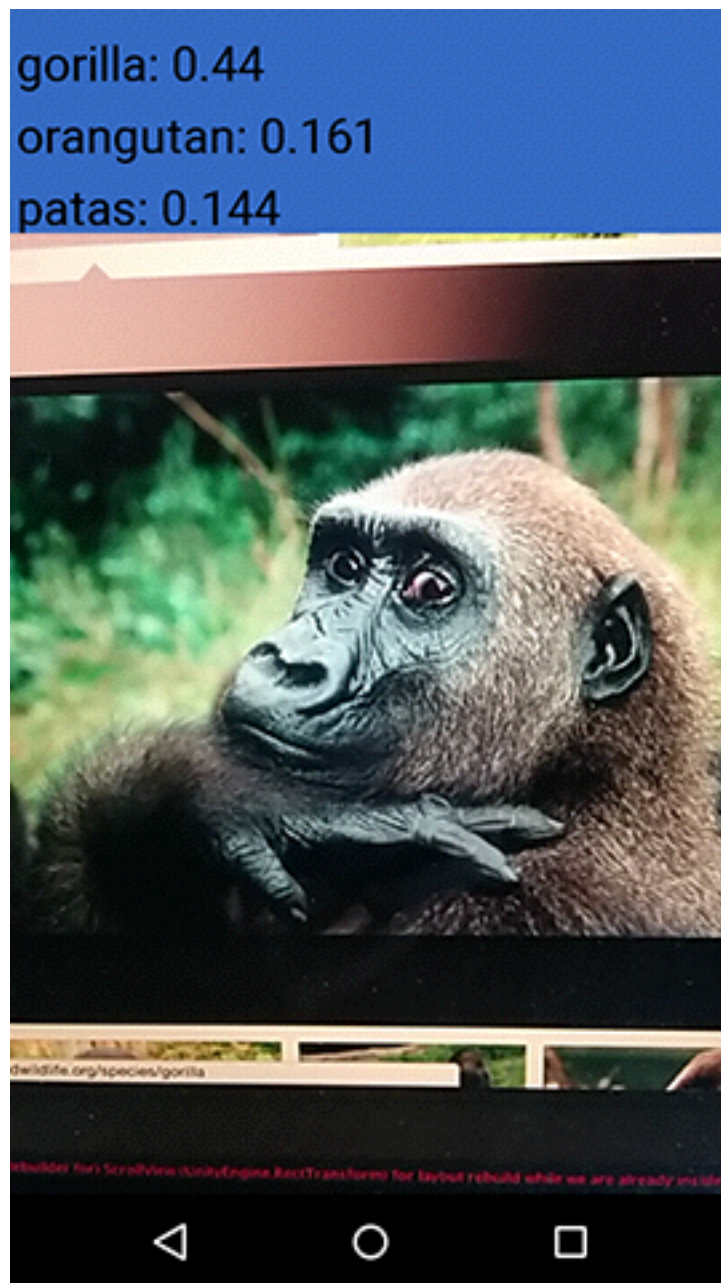
# ニューラルネットとは

**float[]** -> [ニューラルネット] -> **float[]**

みたいなイメージ

floatの配列を入れるとfloat  
の配列が出てくる

- ・ カメラを向けるとリアルタイムに1000種類の中から一番近いと思われるものを提示する

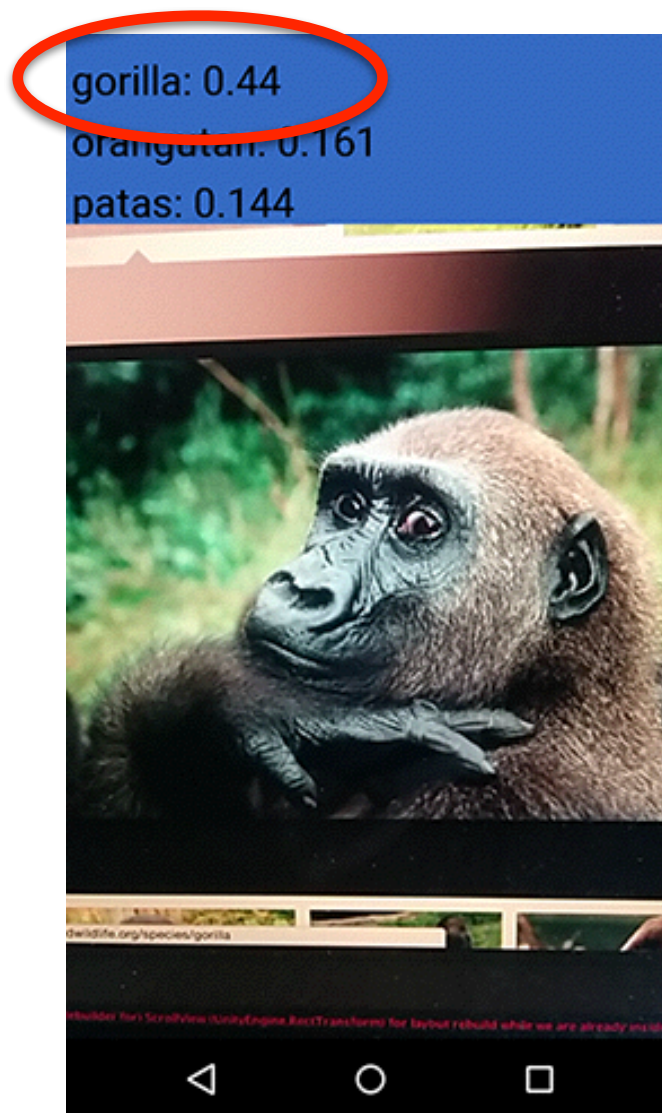


- ・ 手書き数字認識



$\text{float}[224*224*3] \rightarrow \text{ニューラル ネット} \rightarrow \text{float}[1000]$

224ピクセルx  
224ピクセルx  
RGB3色



キツネ = 0.00002  
電車 = 0.00002  
ゴリラ = 0.44  
オラウータン = 0.161  
...  
...  
ノートパソコン = 0.034  
カルボナーラ = 0.0001

1000個の候補の中から  
一番値が大きかったのが  
ゴリラ

$\text{float}[28*28]$   $\rightarrow$  ニューラル  
ネット  $\rightarrow$   $\text{float}[10]$

28ピクセルx  
28ピクセル  
グレースケール



0 = 0.00002  
1 = 0.00002  
2 = 0.03  
3 = 0.95  
4 = 0.02  
...  
...  
9 = 0.012

10個の値の中から  
一番大きいのを提示



・ “0.72”      ->      **ただの数字**

・ {0.72, 0.45, 0.11}      ->      **配列**

・ { {0.72, 0.45, 0.11},  
    {0.23, 1.24, 5.23},  
    {2.23, 0.12, 1.12} }      ->      **二次元配列**

- “0.72” → 0階テンソル
- {0.72, 0.45, 0.11} → 1階テンソル
- { {0.72, 0.45, 0.11},  
{0.23, 1.24, 5.23},  
{2.23, 0.12, 1.12} } → 2階テンソル
- ... → 3階テンソル
- ...

テンソル → テンソルに対するいろんな計算 → テンソル



# 手書き文字認識 (MNIST)

手書き文字の数字を当てると  
ころを疑似コードにて説明



- `pseudo_code.txt`

`input -> float[784]`

28x28(=784)個のピクセル値を0.0~1.0にして、ベタにつなげて1次元の配列にしたもの

これに演算をかけて、10個のfloat値(“0”~“9”のそれぞれのもらしさ)を出す

- **pseudo\_code.txt**

*// 784x10のweight値*

**private float**[][] **weightW** = **new float**[784][10];

*// 10個のbias値*

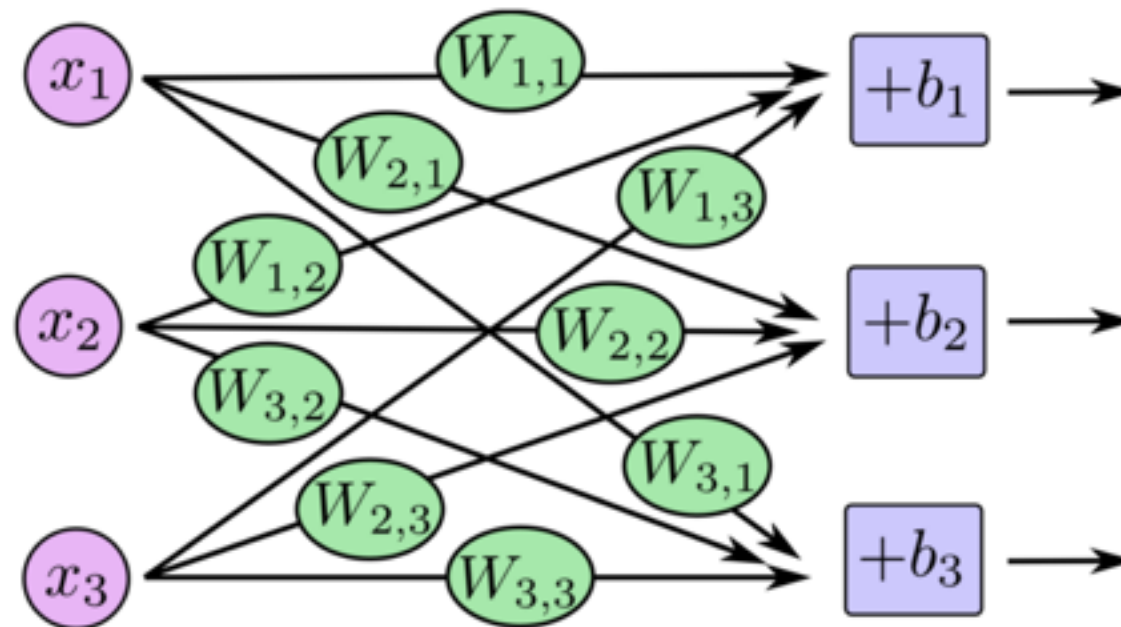
**private float**[] **biasB** = **new float**[10];

- **これらの値が学習によりもう求まった状態だとする**

```
public int detectDigit(float[] input) {  
    // 10個の出力値を準備  
    float[] output = new float[10];  
  
    // input と weightW の掛け算を行う  
    for (int j = 0; j < 784; ++j) {  
        for (int i = 0; i < 10; ++i) {  
            output[i] += input[j] * weightW[j][i];  
        }  
    }  
  
    // それに biasB を足す  
    for (int i = 0; i < 10; ++i) {  
        output[i] += biasB[i];  
    }  
    ...  
}
```



$$\text{output} = \text{input} * W + b$$



784個

10個

- **softmax()**

10個の値を合計して1.0になる様にするexp()を使った計算 (各出力値も0.0~1.0の間になる)

**exp()の結果は必ず0より大きい正の値になる**

**例:**

$\exp(10) \rightarrow 22026.5$

$\exp(5) \rightarrow 148.4$

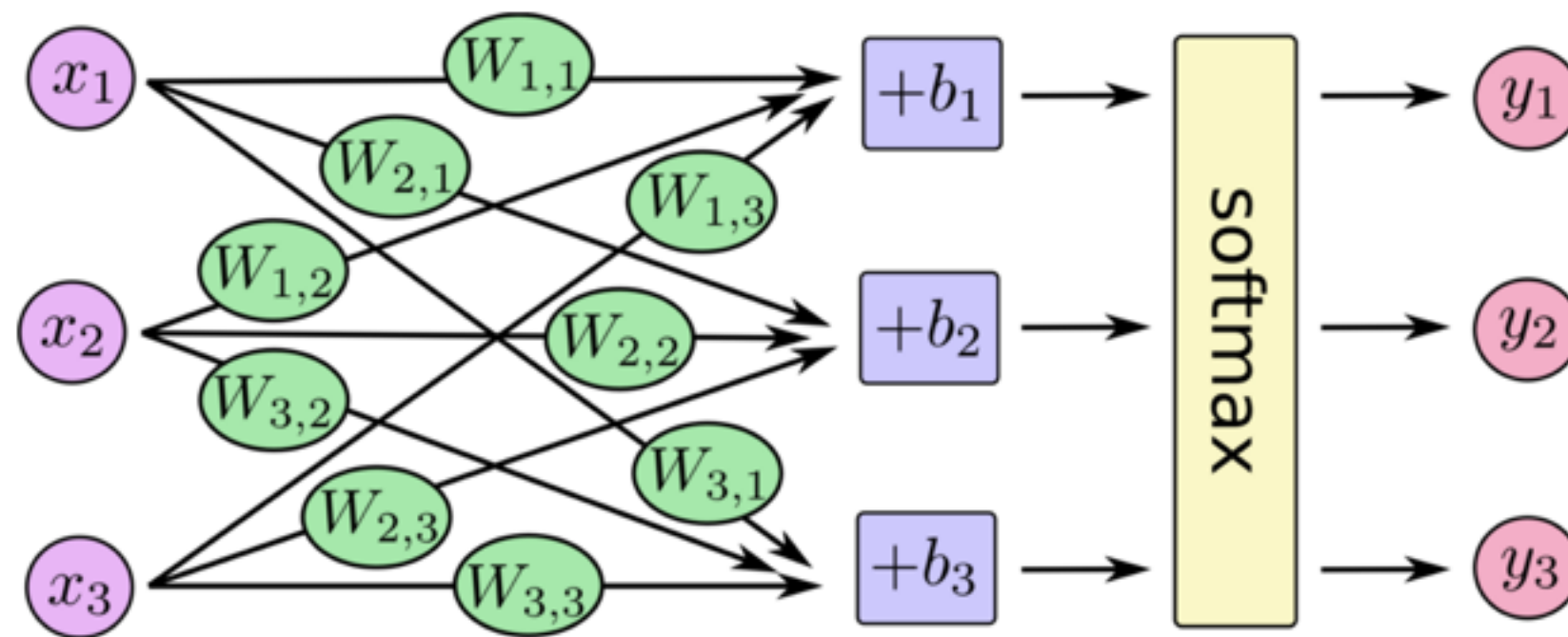
$\exp(0) \rightarrow 1.0$

$\exp(-5) \rightarrow 0.00674$

$\exp(-10) \rightarrow 0.0000454$

```
private float[] softmax(float[] values) {  
    // 各値のexp値  
    float[] expValues = new float[values.length];  
  
    float expSum = 0.0f; // 各値のexp値の合計  
    for (int i = 0; i < values.length; ++i) {  
        // 各値のexp値を出す  
        float exp = (float) Math.exp(values[i]);  
        expValues[i] = exp;  
        expSum += exp; // 合計値を加算  
    }  
  
    // 合計値で割って、全部のexp値の合計が1になる様にする  
    for (int i = 0; i < values.length; ++i) {  
        expValues[i] /= expSum;  
    }  
    // 結果はそれぞれ0.0~1.0の間の値  
    return expValues;  
}
```

$$\text{output} = \text{softmax}(\text{input} * W + b)$$



784個

10個

10個

10個の出力の内最大のものが、output[3]なら、  
文字が”3”だと推測



// 784x10のweight値

**private float[][] weightW = new float[784][10];**

// 10個のbias値

**private float[] biasB = new float[10];**

# MacにTensorFlowをインストール します

[https://github.com/miyosuda/intro-to-dl2/wiki/  
%E7%92%B0%E5%A2%83%E6%A7%8B  
%E7%AF%89](https://github.com/miyosuda/intro-to-dl2/wiki/%E7%92%B0%E5%A2%83%E6%A7%8B%E7%AF%89)

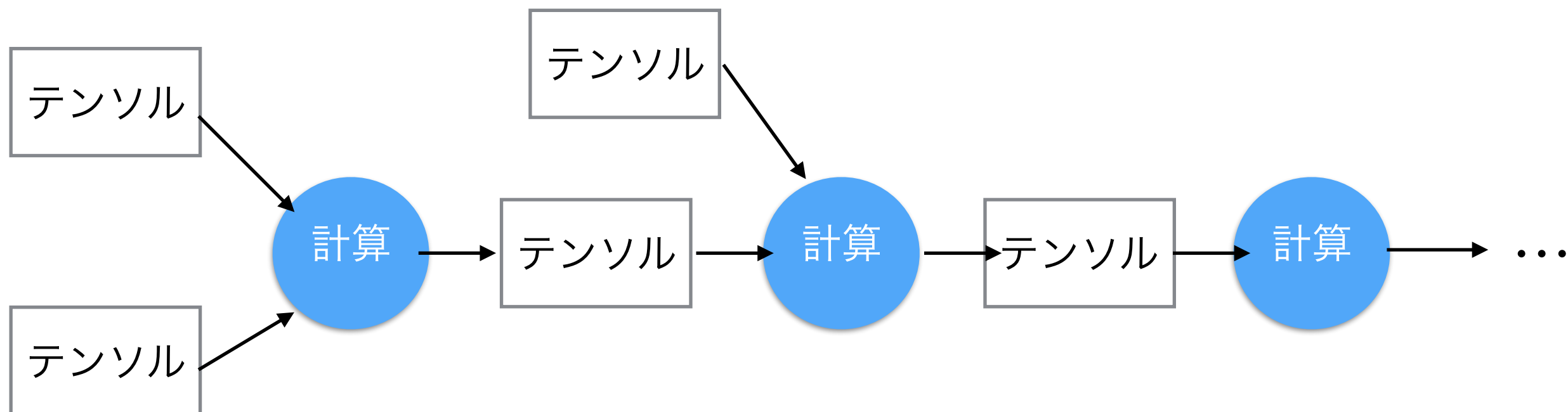
こちらの手順でインストール



# TensorFlowの流れ

1. グラフの定義
2. グラフの実行

# グラフ



# MNIST

- ・ 訓練用の55000組の手書き画像(28x28ピクセル)と正解データ(0~9)
- ・ 10000組の確認用データ(画像+正解データ)
- ・ 違いは訓練用と確認用に分けただけ

[https://github.com/miyosuda/intro-to-dl2/mnist/  
mnist.py](https://github.com/miyosuda/intro-to-dl2/mnist/mnist.py)

# [グラフ定義]

# 入力用の値のいれ物(=Placeholder)を作成する. ここには画像データをテンソルにしたものが  
# 入ってくる.

# Noneは"数が未定"を表す. 学習時はここが100になり、確認時は10000になる.

# なので、学習時は(100x784)のテンソル、確認時は(10000x784)のテンソルになる.

```
x = tf.placeholder(tf.float32, [None, 784])
```

# 784x10個の重み. 学習により変化していく.

```
W = tf.Variable(tf.zeros([784, 10]))
```

# 10個のBias値. 学習により変化していく.

```
b = tf.Variable(tf.zeros([10]))
```

#  $(x * W + b)$ の結果をsoftmax関数に入れ、その結果をyとする.

# yは学習時は(100x10)のテンソル. 確認時は(10000x10)のテンソルになる.

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

# 損失関数(正解とどれくらいずれているかを表すもの)を定義していく

# y\_ は正解データを入れる入れもの.

# Noneとなっているが、学習時にはここが100になり、

# y\_は(100, 10)のテンソルとなる.

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

# ニューラルネットの出した10個の値と正解の10個の値(正解部分だけが1の配列)を

# もちいて、どれくらいずれていたか、を出す.

# 小さければ小さいほど正解に近かった事を表す値.(合計をひとつのスカラー値として集めたもの)

# 100個分の合計を求める

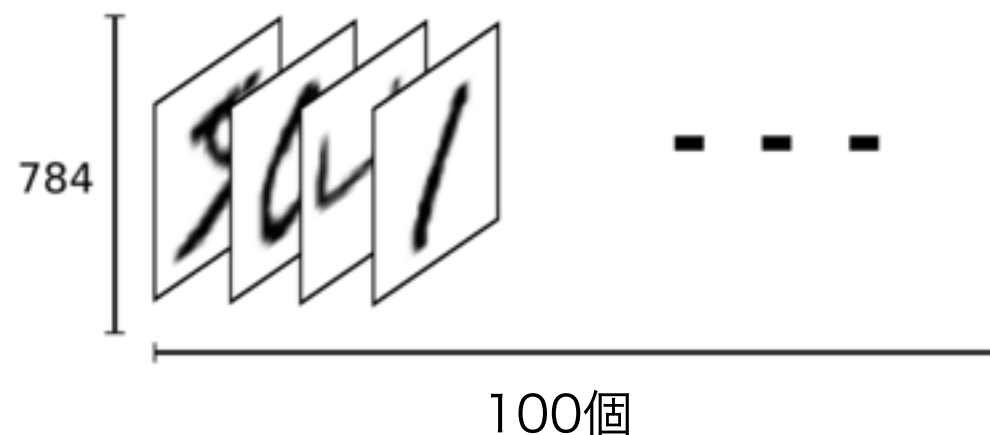
```
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
```

# 上記のずれを小さくする様に学習させるOptimizerを用意する.

```
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

# 学習時の入力

- ・ 学習時は入力画像(28x28ピクセルのfloat値)を100個分まとめて入れている (バッチ処理という)

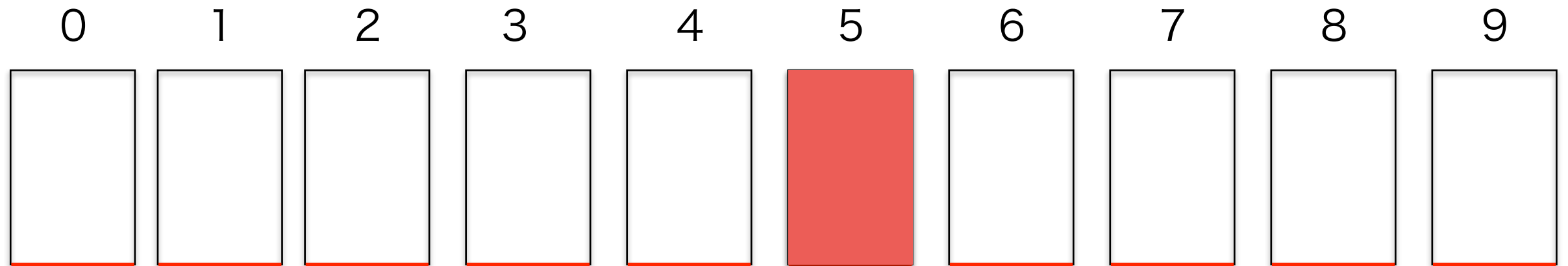


- ・ 出力(float[10])も100個まとめて出てくる

# 正解データ

- 正解データ (0~9のどれになるか) も output と同じ様に 10 個の float 値の配列となっている

例: 正解が5の場合



$$y_ = \{0, 0, 0, 0, 0, 1, 0, 0, 0, 0\}$$

このような形式を”**one hot vector**”と呼ぶ



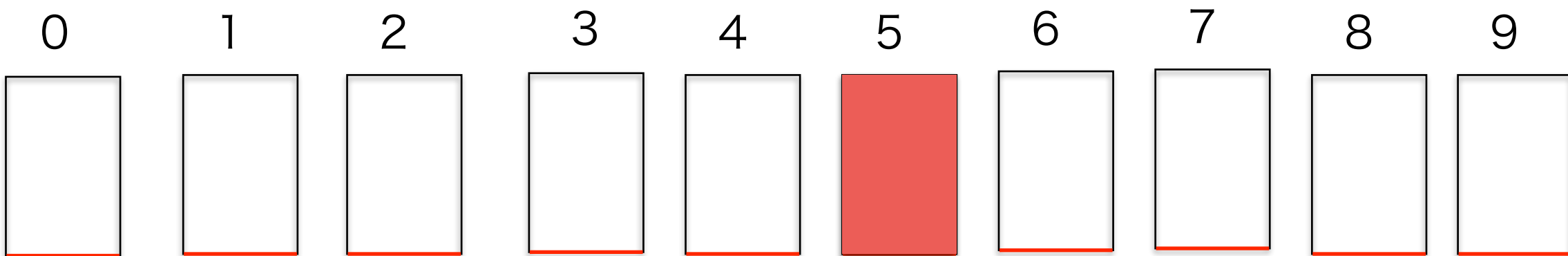
# 正解と出力のずれ

正解

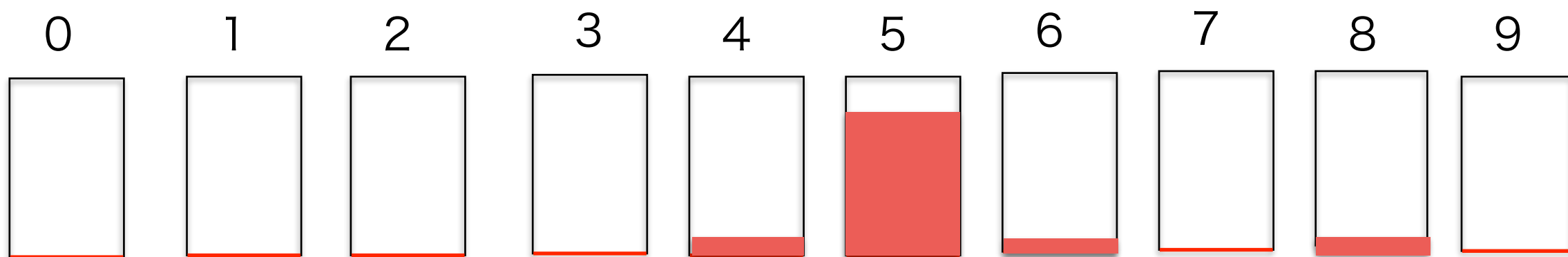
output


$$-y_- * \log(y)$$

## 正解



output



# 0.0～1.0の値のlog()

$$\log(0.0) \rightarrow -\infty$$

$$\log(0.0001) \rightarrow -9.21$$

$$\log(0.1) \rightarrow -2.3$$

$$\log(0.5) \rightarrow -0.69$$

$$\log(0.9) \rightarrow -0.11$$

$$\log(1.0) \rightarrow 0.0$$

0.0以下の負の数になる

# マイナスを掛けると

$$-\log(0.0) \rightarrow \infty$$

$$-\log(0.0001) \rightarrow 9.21$$

$$-\log(0.1) \rightarrow 2.3$$

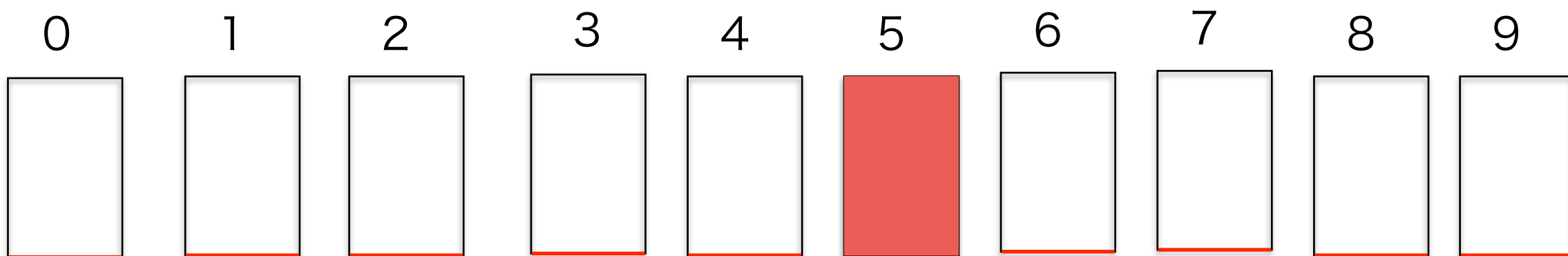
$$-\log(0.5) \rightarrow 0.69$$

$$-\log(0.9) \rightarrow 0.11$$

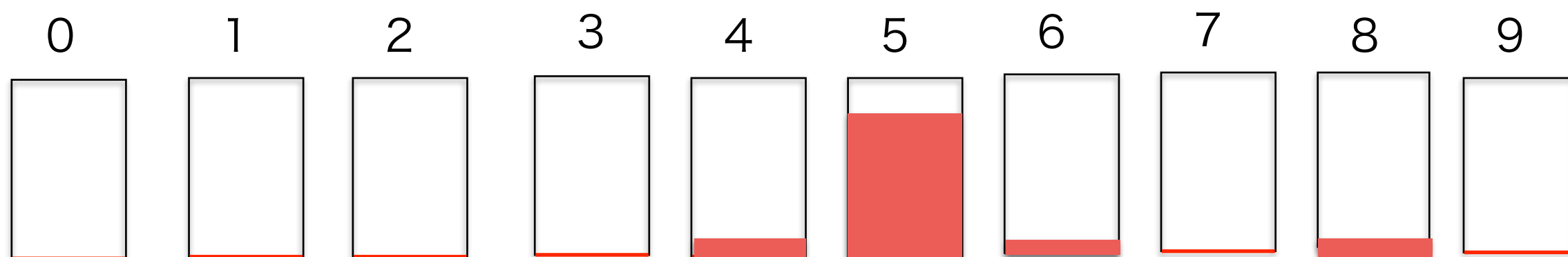
$$-\log(1.0) \rightarrow 0.0$$

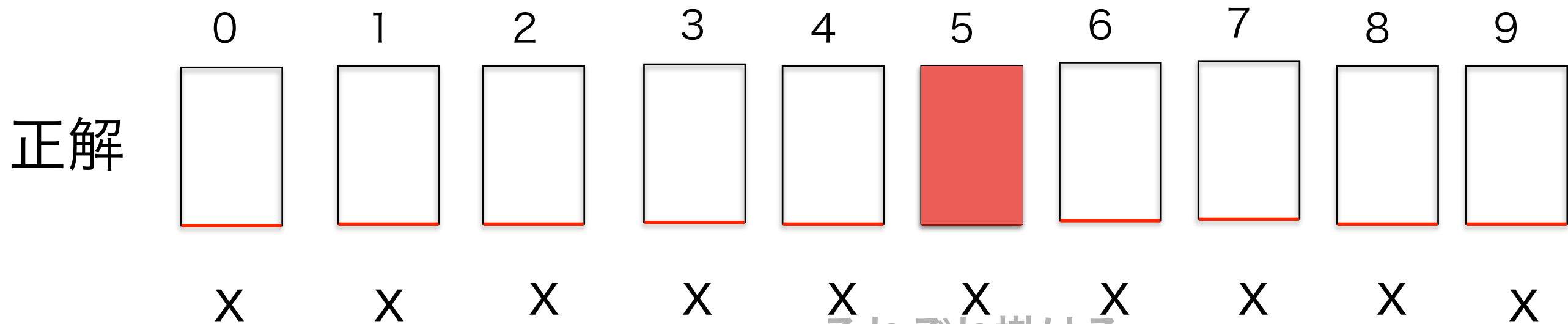
$-\log(1.0)$ が最小で0.0

正解



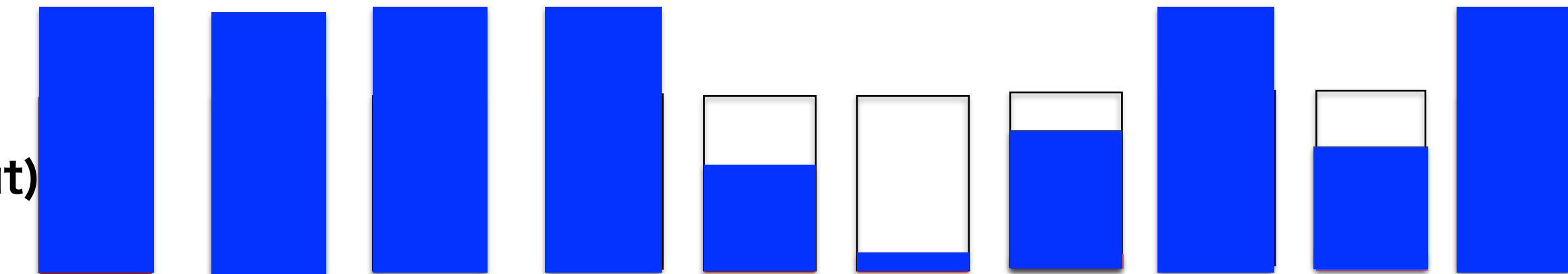
output





それぞれ掛ける

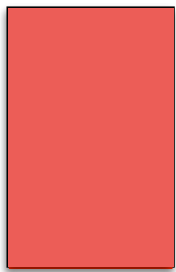
$-\log(\text{output})$



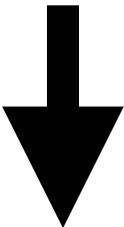
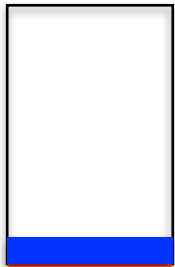
合計？

ここしか残らない

5



X



求めようとしているずれの値

正解

掛ける

$-\log(\text{output})$

$$-y_ * \log(y)$$

output[5]が1.0(=正解と完全に一致)ならばこの値は0

output[5]が1.0から小さくなればなるほど、  
この値は大きくなる

=>この値が小さければ小さいほどより正解に  
近い値をoutputできるニューラルネット



この値を小さくする様に **$W$** と **$b$** を調整していく

# Word2Vec

- ・ 自然言語をニューラルネットで扱いたい
- ・ 単語をベクトルとして表現する必要がある

# one hot vector

犬  $[1, 0, 0, 0, 0, 0, \dots]$

猫  $[0, 1, 0, 0, 0, 0, \dots]$

歩く  $[0, 0, 1, 0, 0, 0, \dots]$

人間が扱う語彙数は通常、数万～のオーダーになる  
-> ベクトルの次元数も数万次元とかになってしまう

# 分散表現

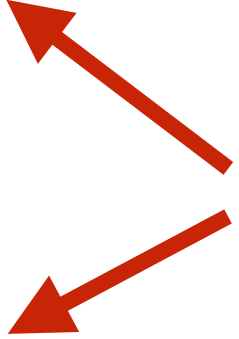
犬 [0.1, 0.3, 0, 0.3, 0, ...]

猫 [0.1, 0.3, 0, 0.2, 0, ...]

歩く [0.5, 0.0, 1.0, 0.0, ...]

複数のニューロンで一つの単語を表す。

数万個の単語を、それぞれ**数百次元**程度の小さなベクトルで十分に表現可能

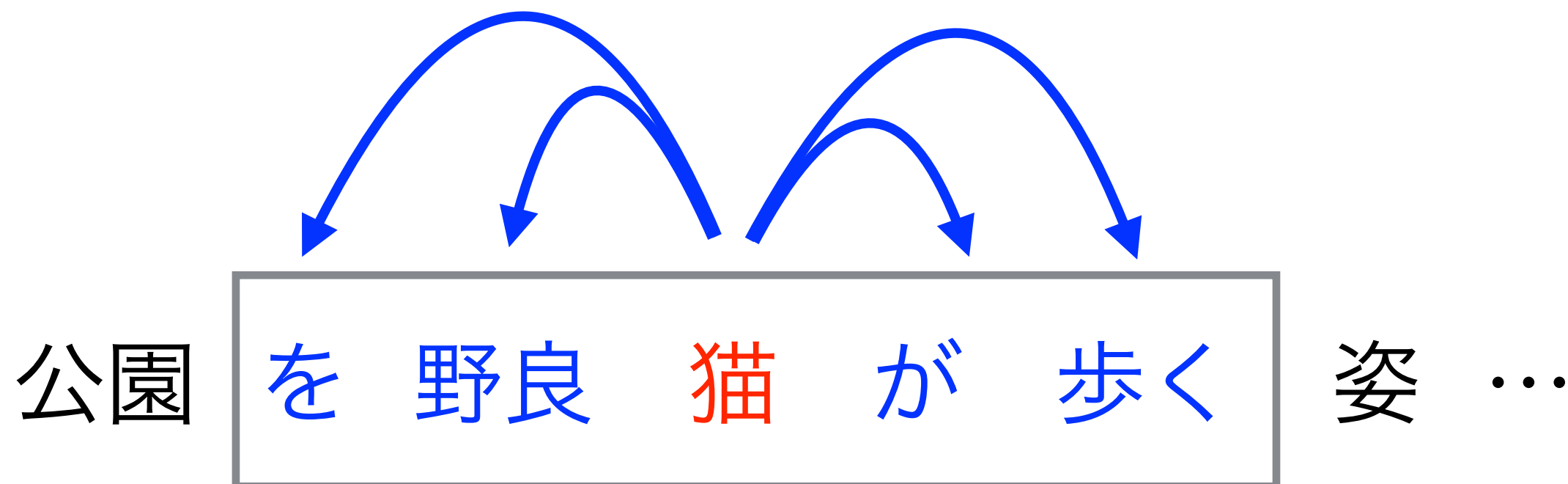
犬	[0.1, 0.3, 0, 0.3, 0, ...]	 似ている
猫	[0.1, 0.3, 0, 0.2, 0, ...]	
歩く	[0.5, 0.0, 1.0, 0.0, ...]	

意味的に近いものを似たような値で表現できる。  
このような分散表現をWord embeddingと呼ぶ

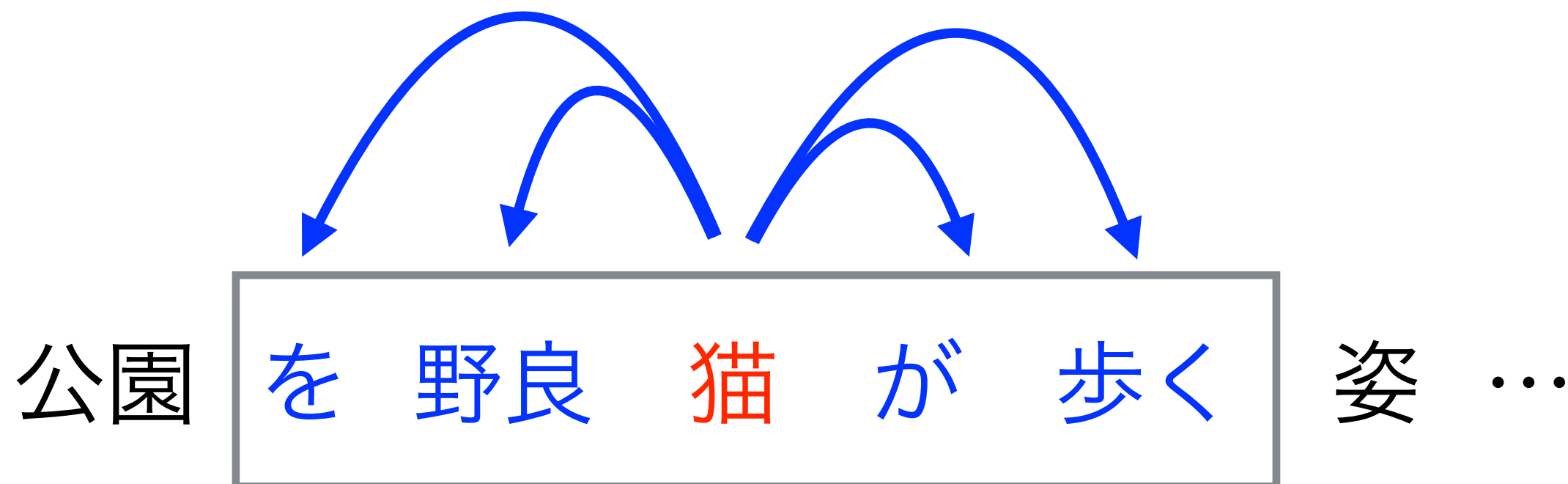
公園 を 野良 犬 が 歩く 姿 …

公園 を 野良 猫 が 歩く 姿 …

同じ文脈で用いられる単語は、同じ様な意味を持つ、  
と想定できる



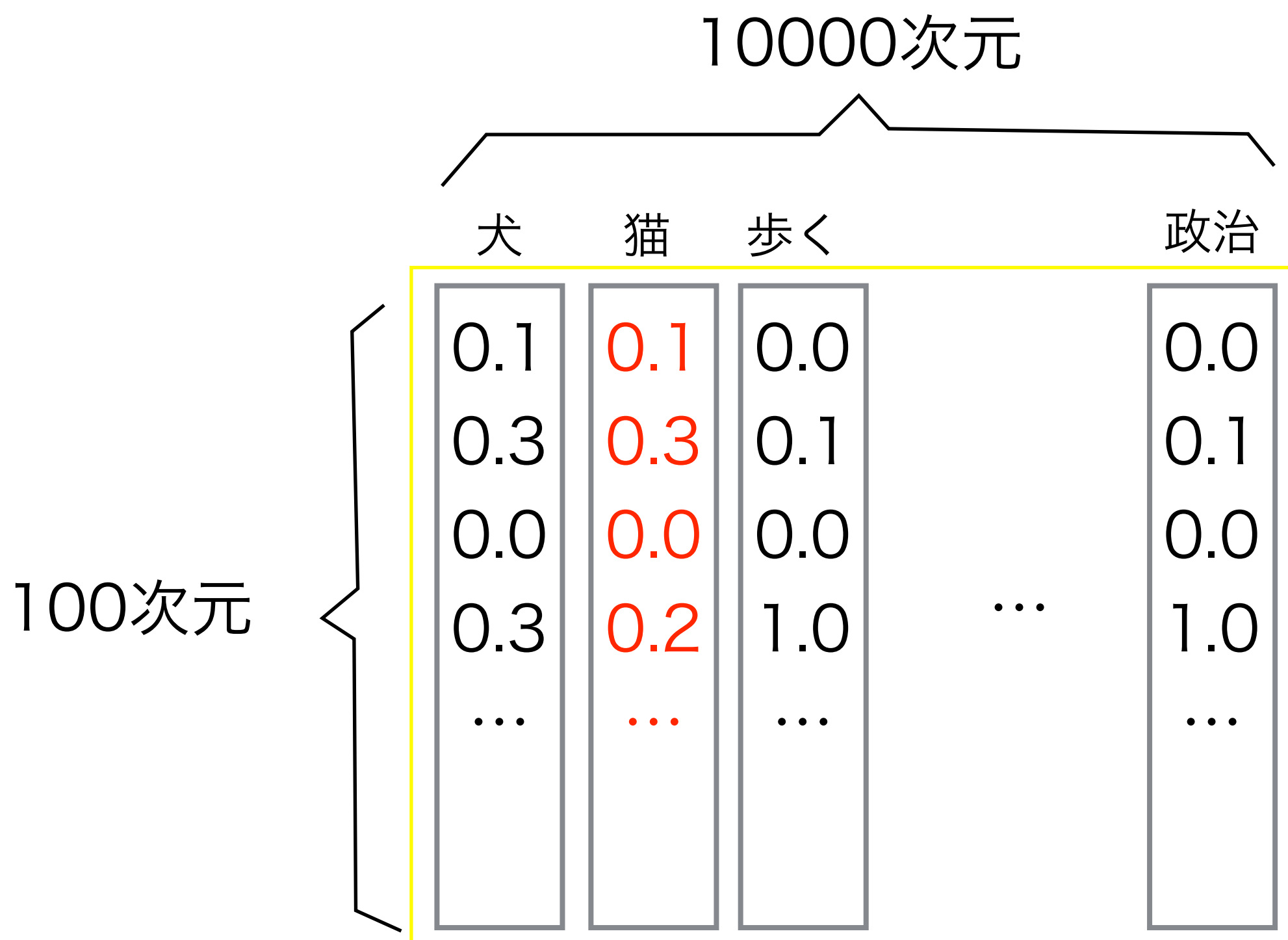
ある単語の周りに出現する単語は関係が深い



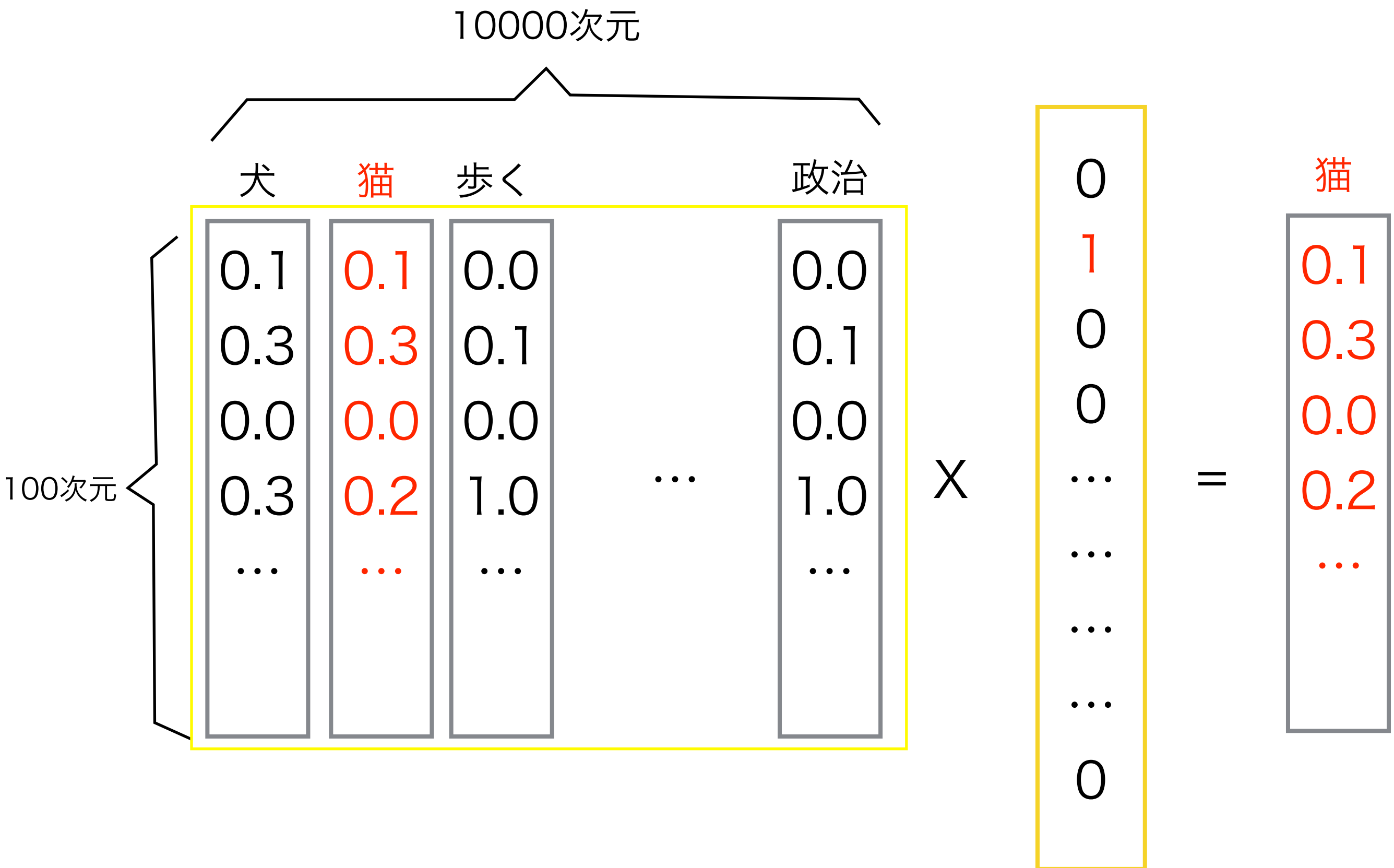
0.1  
0.3  
0.0  
0.2  
...

ある単語の周りにどんな単語が来るかを予測できる様に学習する

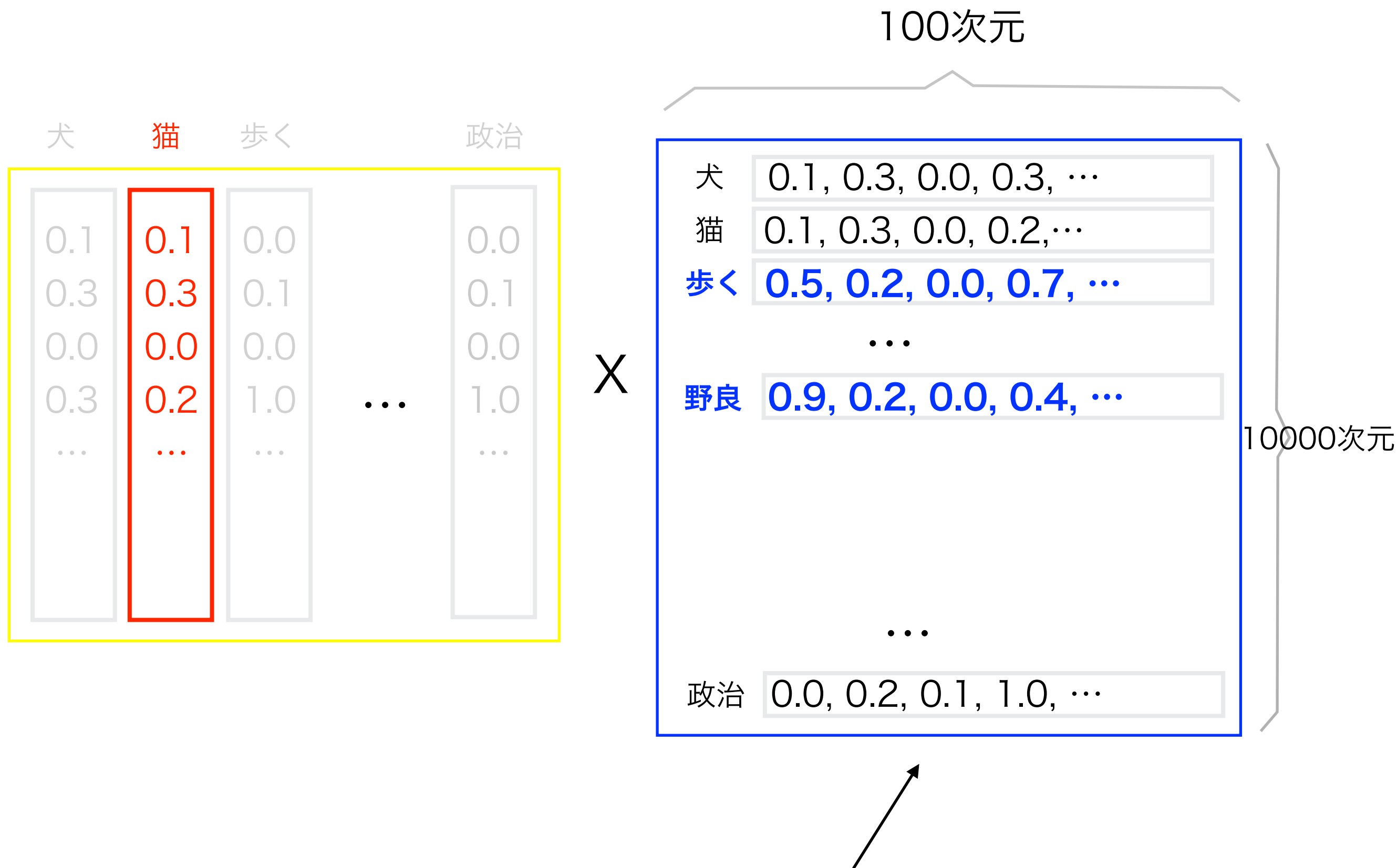




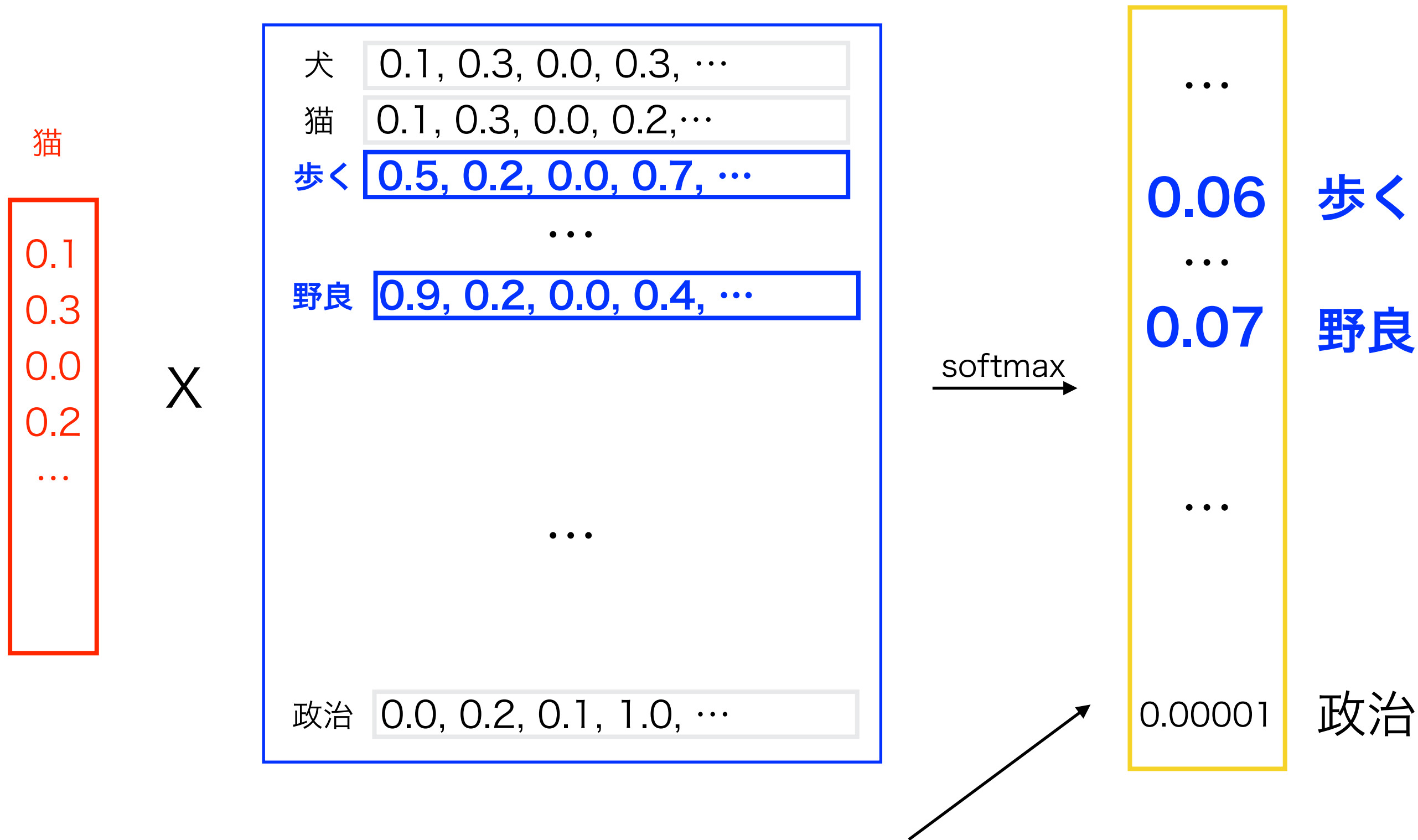
単語ベクトルを横に並べて行列にしたものを考える。  
語彙が1万語、Word embeddingの次元数を100とすると  
(10000 x 100) の行列になる



この行列に1万次元のone hot vectorをかけると該当の単語ベクトルが抜き出せる

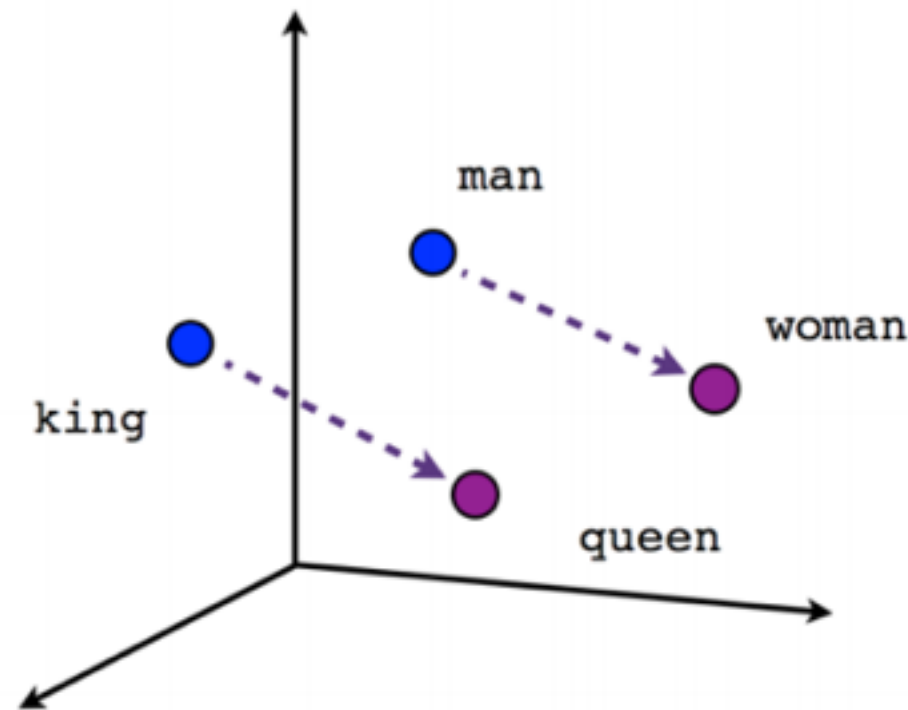


今度は単語ベクトルを縦にならべた行列を考える

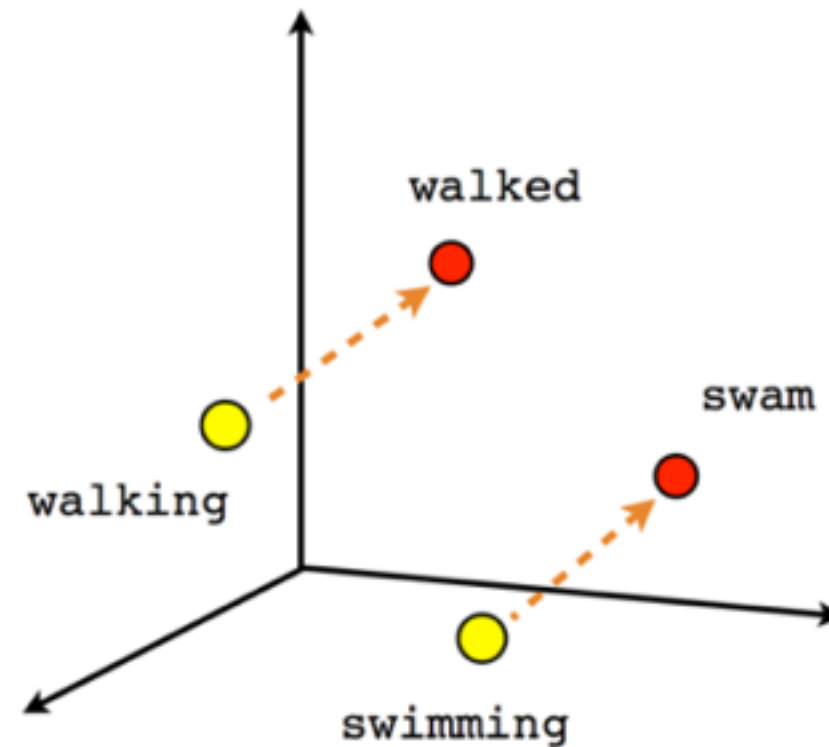


猫の周りに「歩く」や「野良」が出る確率を大きくする様に学習させる

得られた単語ベクトルで足し算、引き算などの演算ができる



Male-Female



Verb tense

man - woman + queen = king

walked - walking + swimming = swam

パリ - フランス + 日本 = 東京

猫

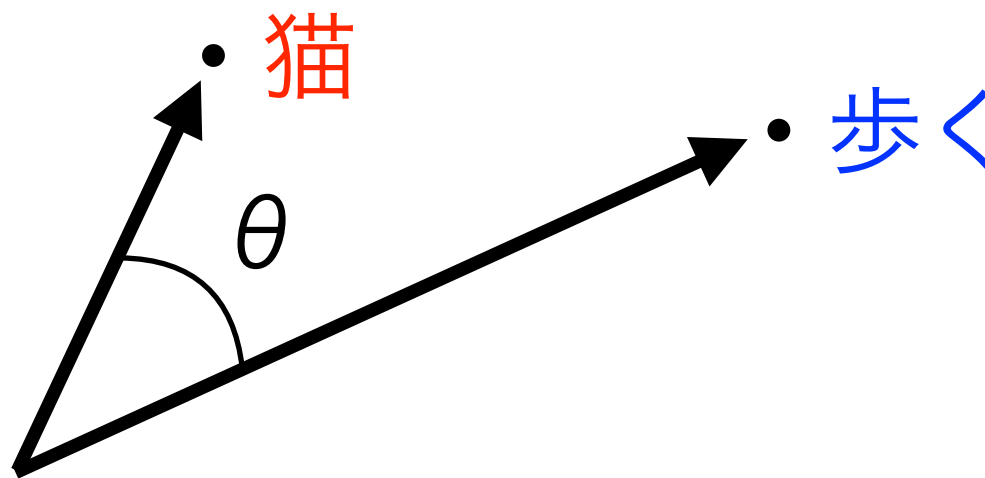
0.1  
0.3  
0.0  
0.2  
...

X 歩く  $[0.5, 0.2, 0.0, 0.7, \dots]$

$$V_{\text{猫}} \cdot V_{\text{歩く}}^T$$

ベクトルの内積

$$\cos \theta = \frac{V_{\text{猫}} \cdot V'^{\top}_{\text{歩く}}}{\|V_{\text{猫}}\| \|V'^{\top}_{\text{歩く}}\|}$$



$\theta$  が小さい ( $\cos \theta$  が大きい) ほど内積が大きい  
= 類似度が高い