

データサイエンス概論I & II データサイエンス総論I & II

パターン認識とニューラルネットワーク

九州大学 数理・データサイエンス教育研究センター

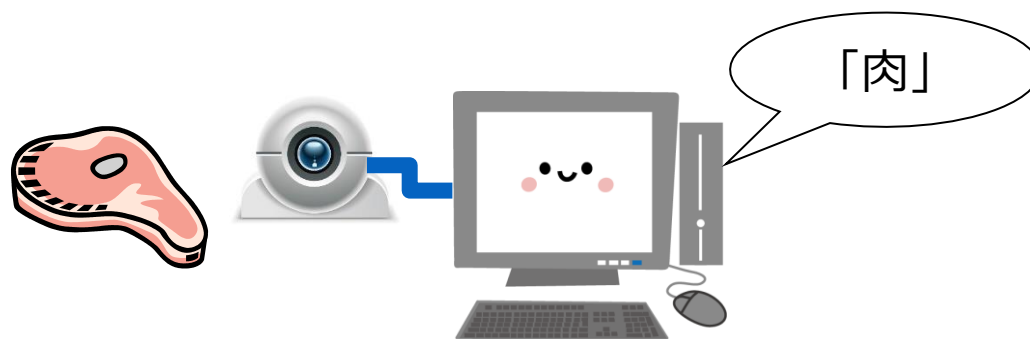
パターン認識とは？

皆さん，無意識にやってます

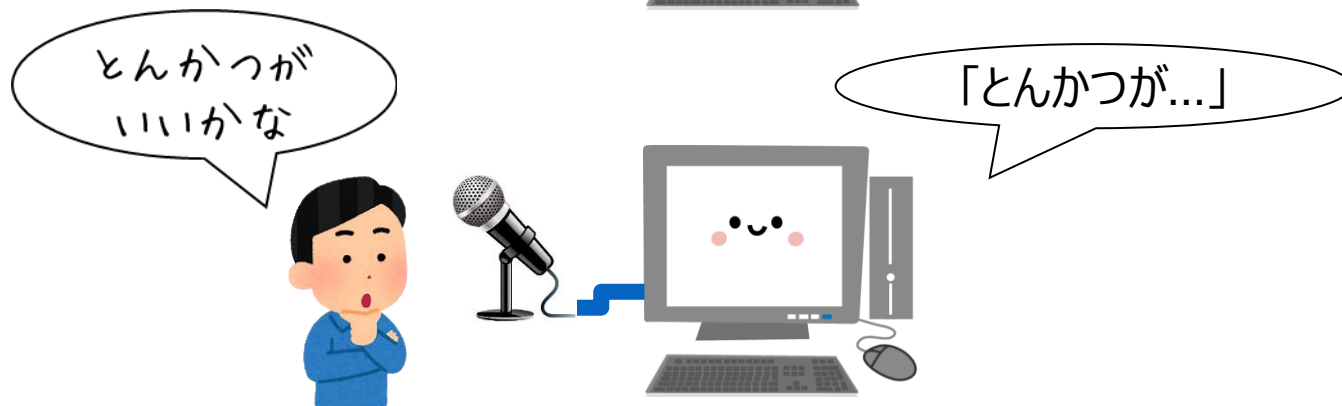
パターン認識とは何か？ ～ その目的

- 画像や音声，テキストなど，様々なデータを対象として，**それが「何」であるかを当てる方法**

- 画像認識



- 音声認識



色々なパターン認識



余談：人間なら，無意識・高精度・高速に認識可能

- データサイエンスなど全く知らない乳幼児ですら，パターン認識



お肉だ!

余裕で画像認識



僕も食べたい!

余裕で音声認識

- より高度な「雰囲気」や「おもしろさ」の把握なども

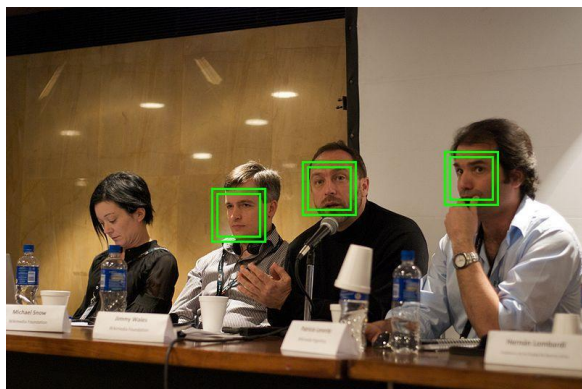
人間であれば、
この画像 1 枚から
雰囲気を含め
様々なことを
読み取れる



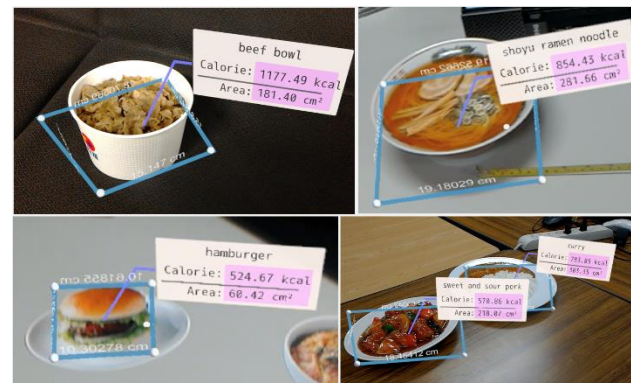
淡いなあ

身近になりつつあるパターン認識：画像認識の例

顔認識



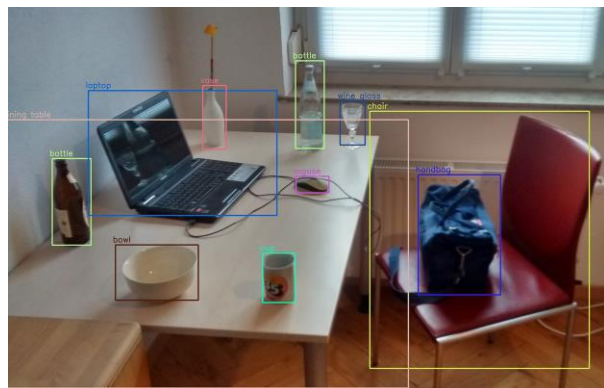
食事認識 (さらにカロリー推定)



電気通信大学 柳井研究室

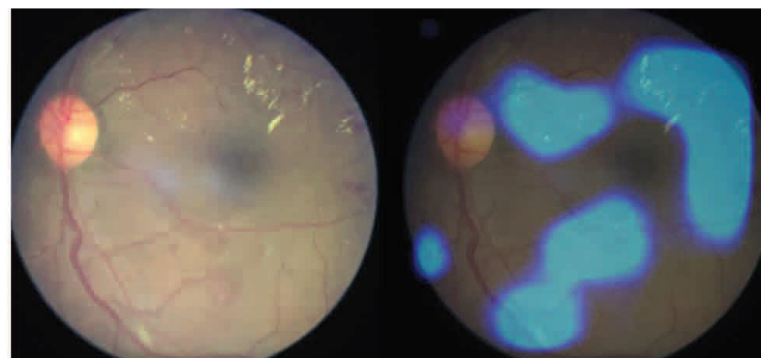
<https://mm.cs.uec.ac.jp/research/?res=naritomi-s&utf8=1>

物体認識



Mtheiler@Wikipedia 物体検出

医用画像認識



[Natarajan+, JAMA Ophthalmol., 2019]

身近になりつつあるパターン認識：様々な認識

● 音声認識 (スマートスピーカー)

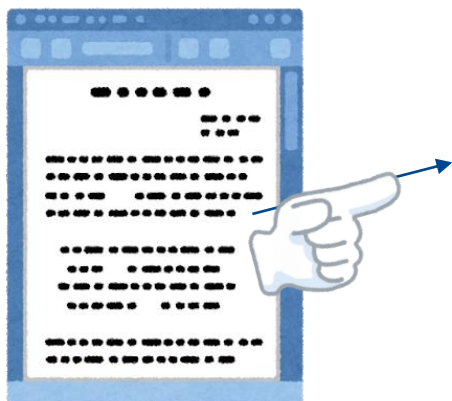


● 行動認識



富士通「AIによる体操の技の認識」 <https://blog.global.fujitsu.com>

● テキストピックアップ認識



- 政治
- 経済
- **スポーツ**
- 国際
- 科学技術
- エッセイ
- ...

● 文字認識(郵便番号認識)



パターン認識の応用～自動化技術

- 自動運転

- 白線認識, 先行車認識, 歩行者認識, 運転者の眠気認識
- ロボットへのパターン認識搭載



- 自動診断

- 病気の自動診断, 機械の故障診断, 適性診断
- 食料品（果物・魚）の等級診断



- 無人店舗・無人工場・植物工場



- 自動採点…?!

- 手書き答案の採点はまだまだ難しそう…



遊んでみたい人はどうぞ！ Teachable machine

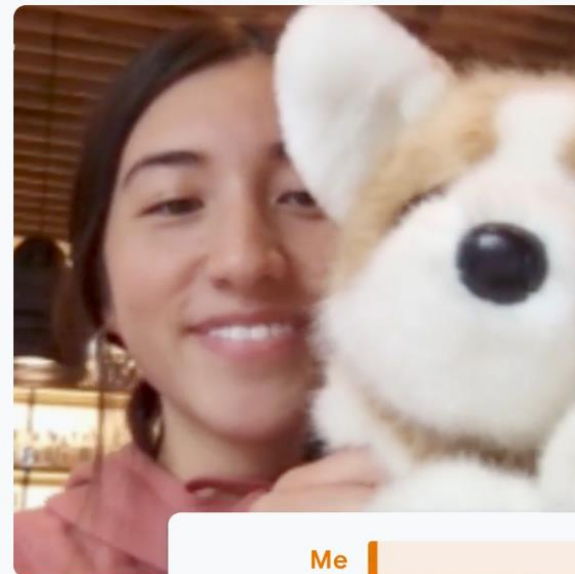
- <https://teachablemachine.withgoogle.com/>

Teachable Machine

独自の画像、音声、ポーズを認識するようコンピュータをトレーニングします。

サイト、アプリなどに使う機械学習モデルをすぐに、簡単に作成できる方法です。専門知識やコーディングは必要ありません。

使ってみる



Me

Me + Dog <3

98%

パターン認識＝「クラスへの分類」

パターン認識 = 事前に決められているクラスへの分類

● 画像認識での例



事前に決めたクラス

- 犬
- **猫**
- 猿
- 鶏
- 牛
- ...

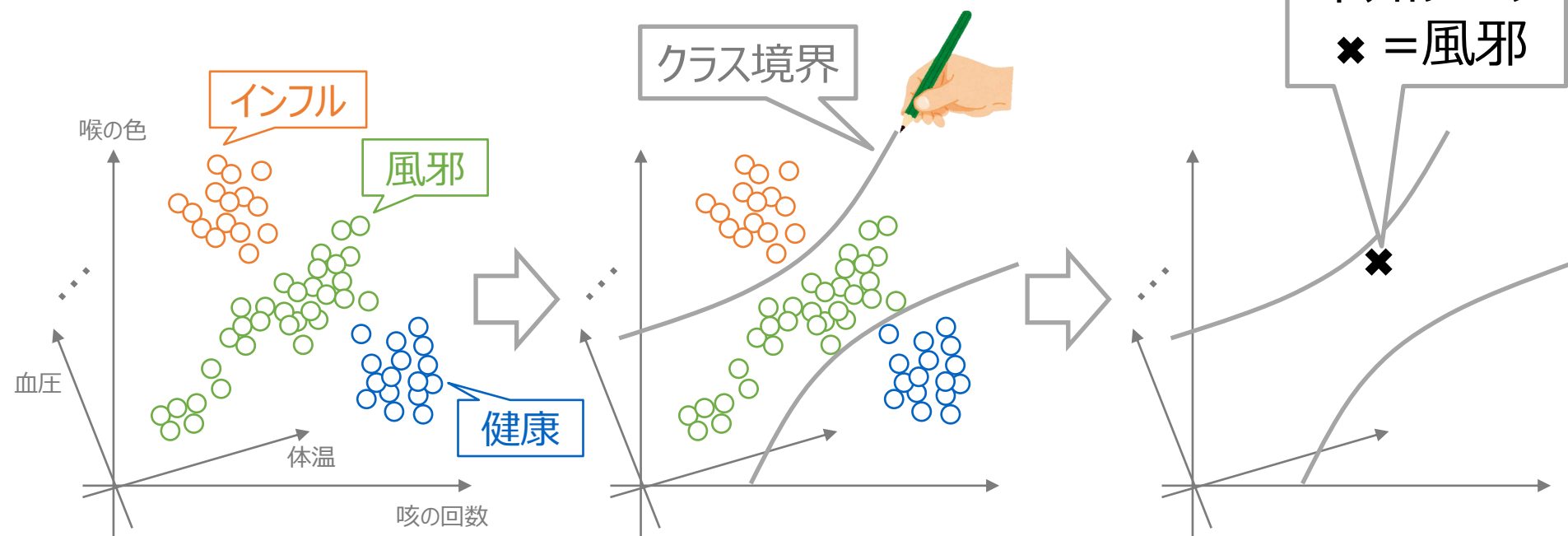
パターン認識 =
入力データの
「**クラスを選ぶ**」処理

● 従って、想定していないクラスには認識できない

- {犬, 猫, 猿, ...}などの動物クラスを対象としたパターン認識器では, 自動車は絶対に認識できない(自動車クラスがないので)
- 同様に{軽自動車, ワゴン車, トラック, ...}などの自動車クラスを対象としたものでは, 動物は絶対に認識できない

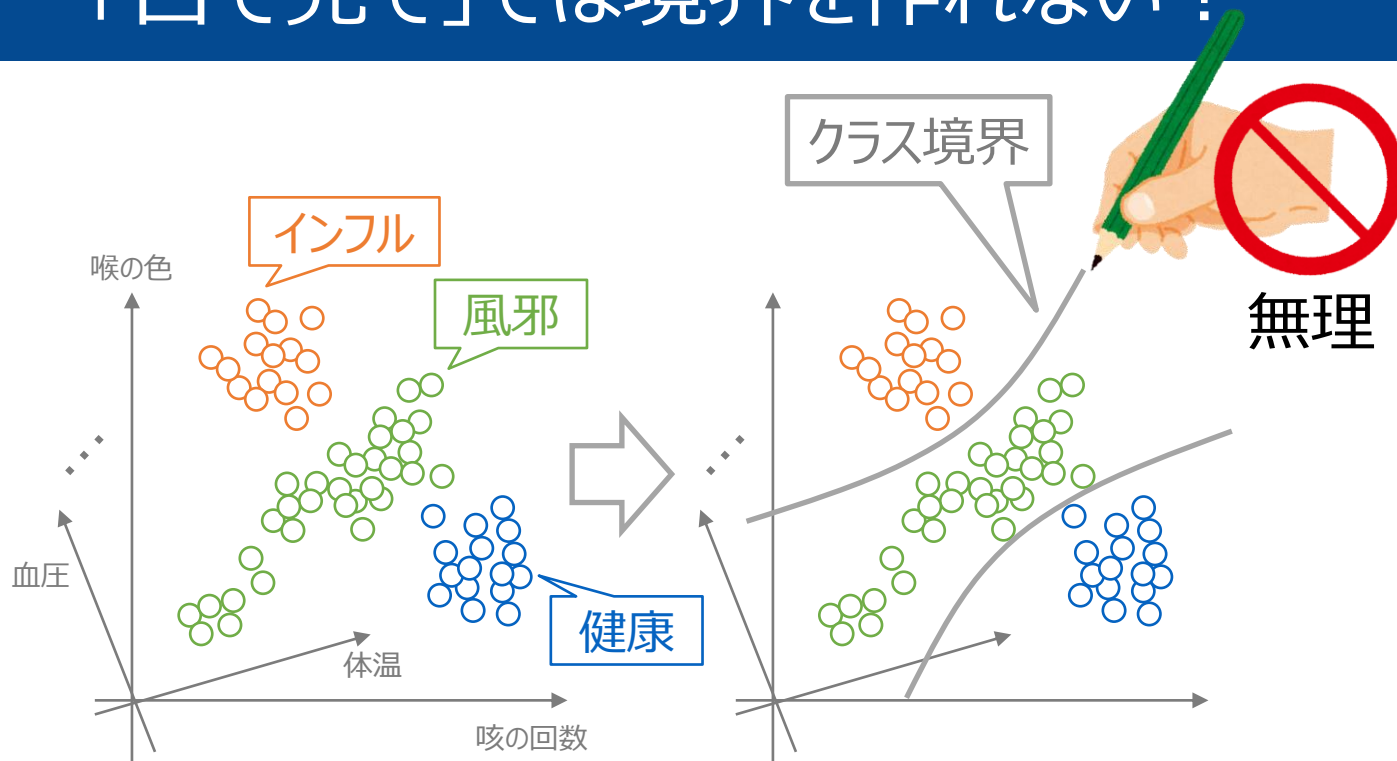
パターン認識を実現するための 基本的な考え方

- 準備①：「クラスがわかっている」データを集める
- 準備②：それらに境界線(クラス境界)を引く
→ この境界線を使えば、未知のデータも「分類」できる



● しかし...

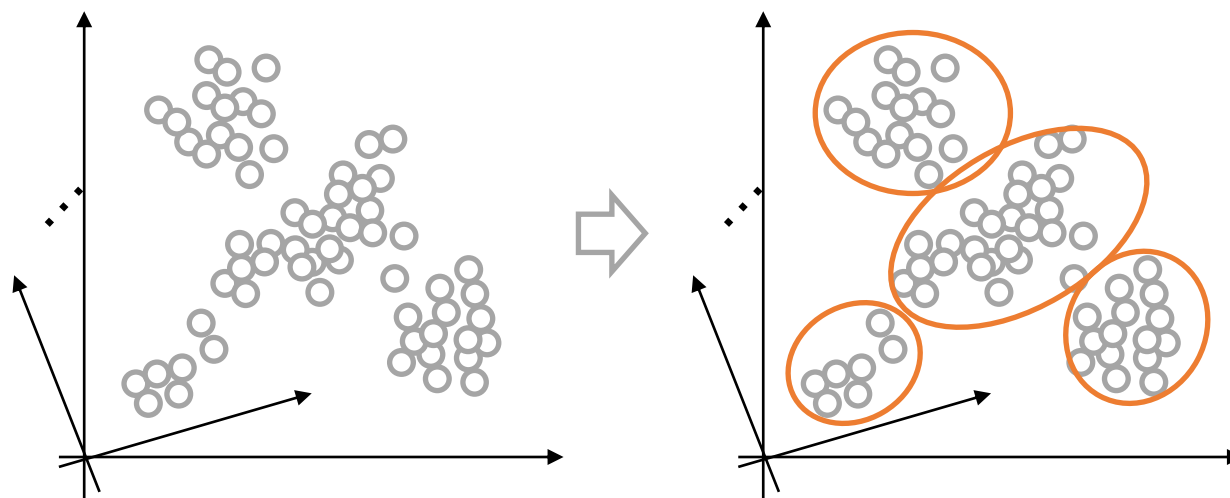
データ数が増える & 高次元になると、 「目で見て」では境界を作れない！



- なので、与えられたデータから、自動でクラス境界を定めたい！
- そこで「機械学習」を利用
 - 最近傍法
 - 識別関数法

参考：以前学んだ「クラスタリング」は、 パターン認識とはちょっと違う

- クラスタリング：似たデータを同じグループにわけ手法
 - 各データが何(=どのクラス)のグループかは不明

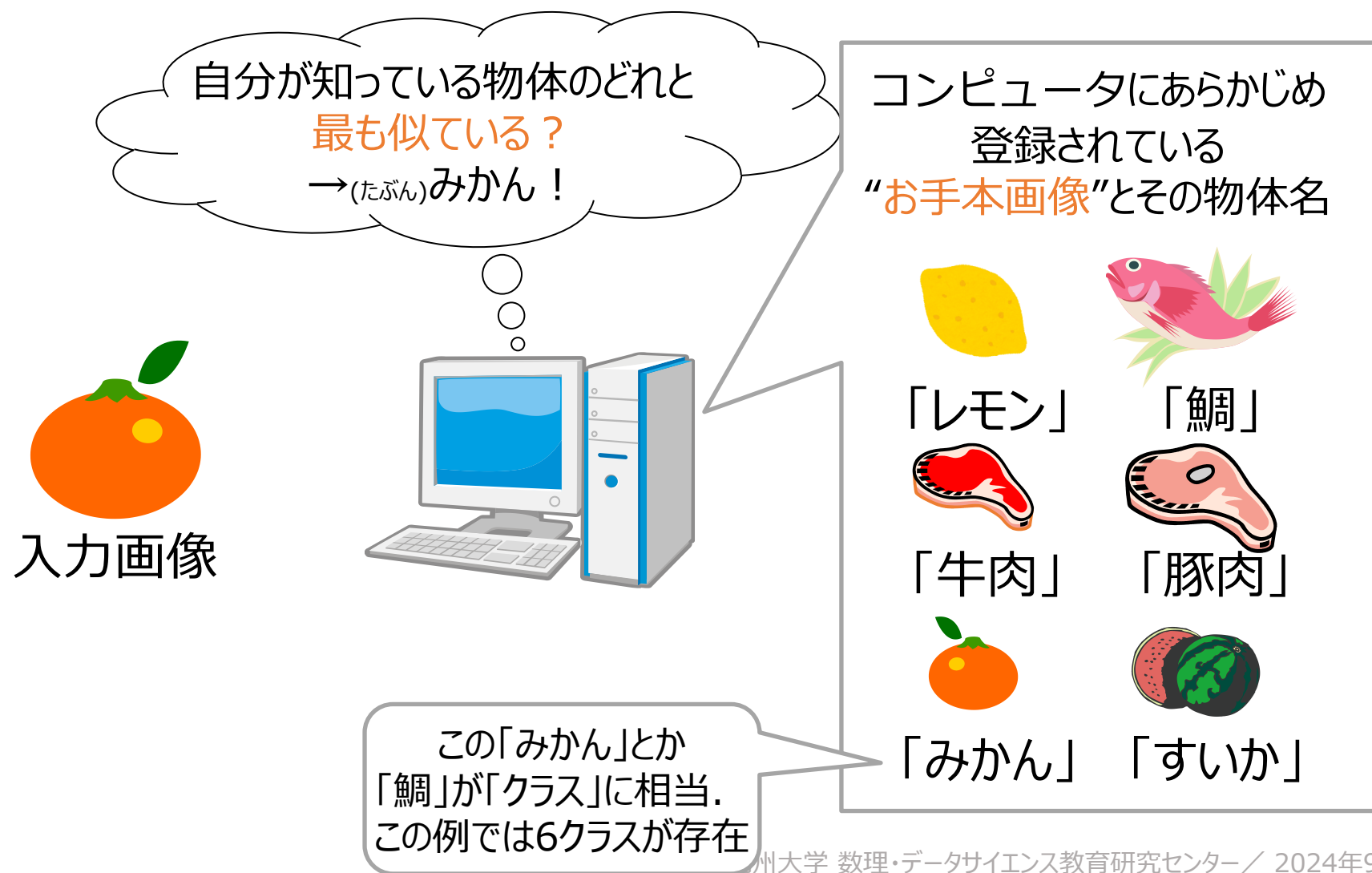


- パターン認識：どのクラスかわかっているデータをつかって境界線を作って、それを用いて未知データを分類

最近傍法： 最も基本的なパターン認識

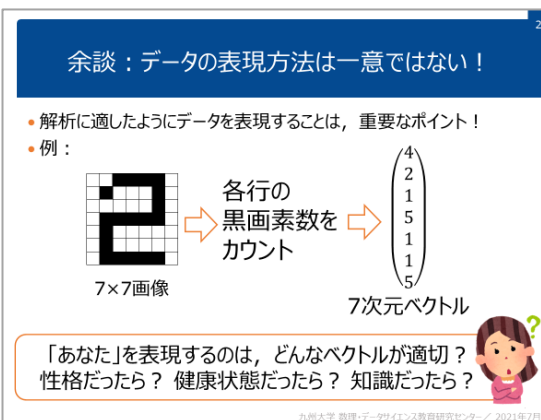
最も簡単な（？）機械学習！
単に「お手本」を計算機に覚えさせるだけ

コンピュータによる「パターン認識」の基本原理： 食べ物画像認識を例に

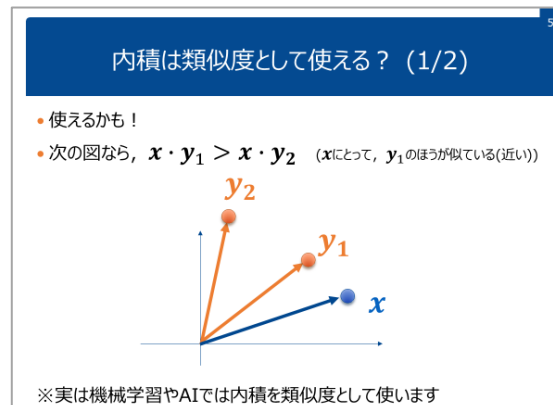
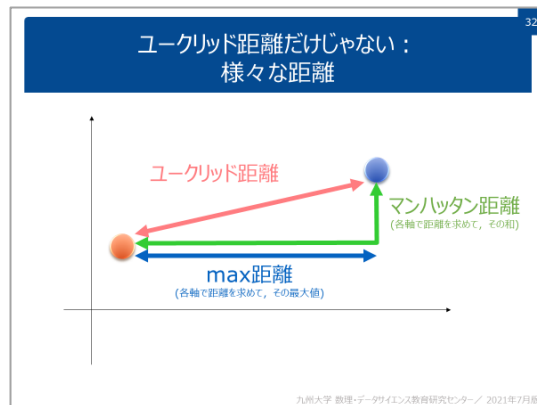


この基本原理について、もう少し踏み込んで考えると、 以前習ったことと関係してきます

● 画像のようなデータをどうやってコンピュータに扱わせる？



● 似てるとは何だ？



準備：データのベクトル表現(1/2)

テクスチャ
(模様)特徴

こんな感じで、各データは
ベクトルで表されているとします

豚肉=(色, 形, 模様)
=(10, 2.5, 4.3)

※これらの数字はテキトーです



形状特徴

色特徴

準備：データのベクトル表現(2/2)

テクスチャ
(模様)特徴

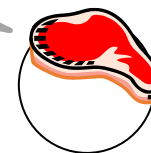
豚肉=(色, 形, 模様)
=(10, 2.5, 4.3)

※これらの数字はテキストです



牛肉=(色, 形, 模様)
=(8, 2.6, 0.9)

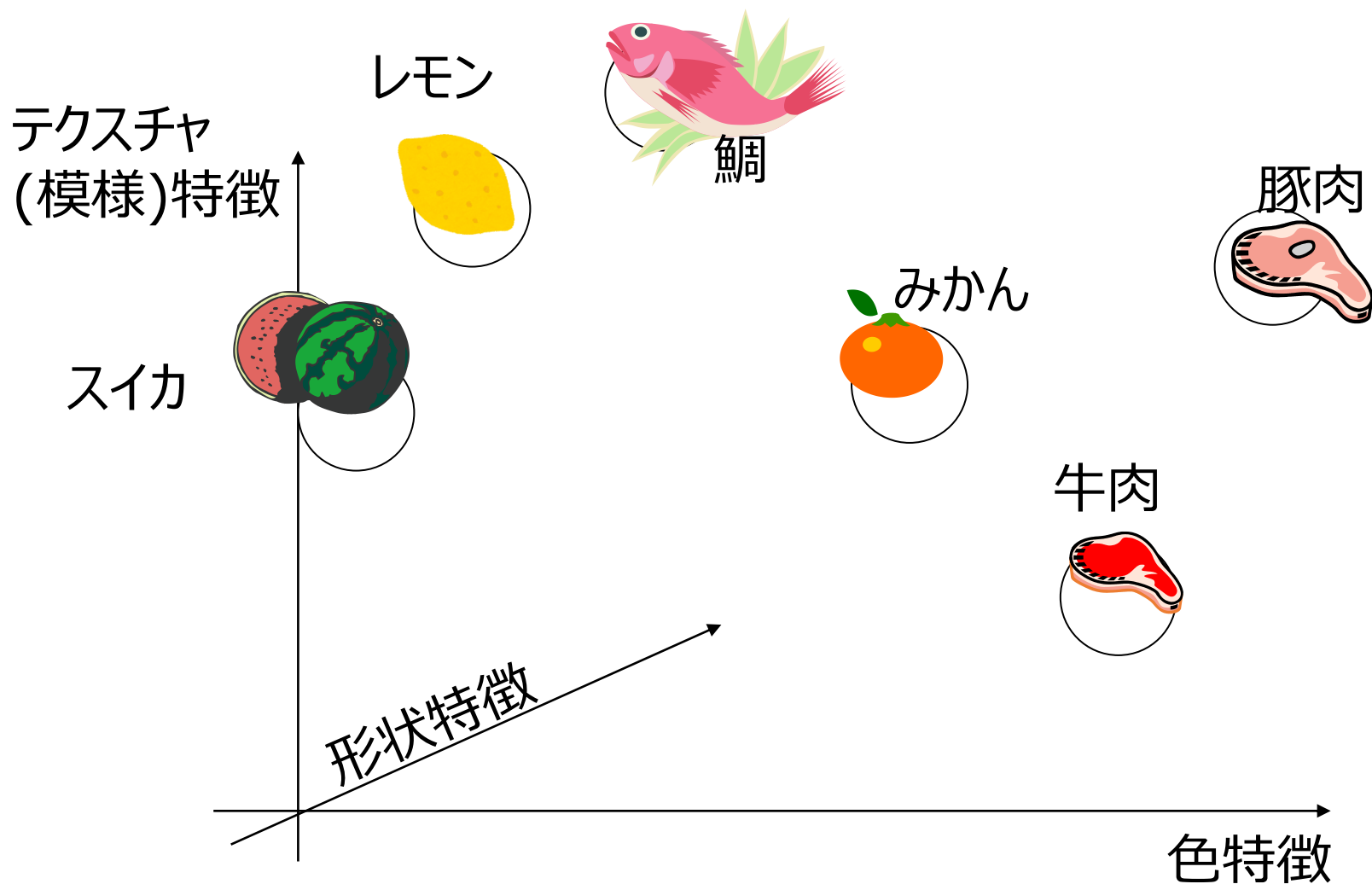
※これらの数字はテキストです



形状特徴

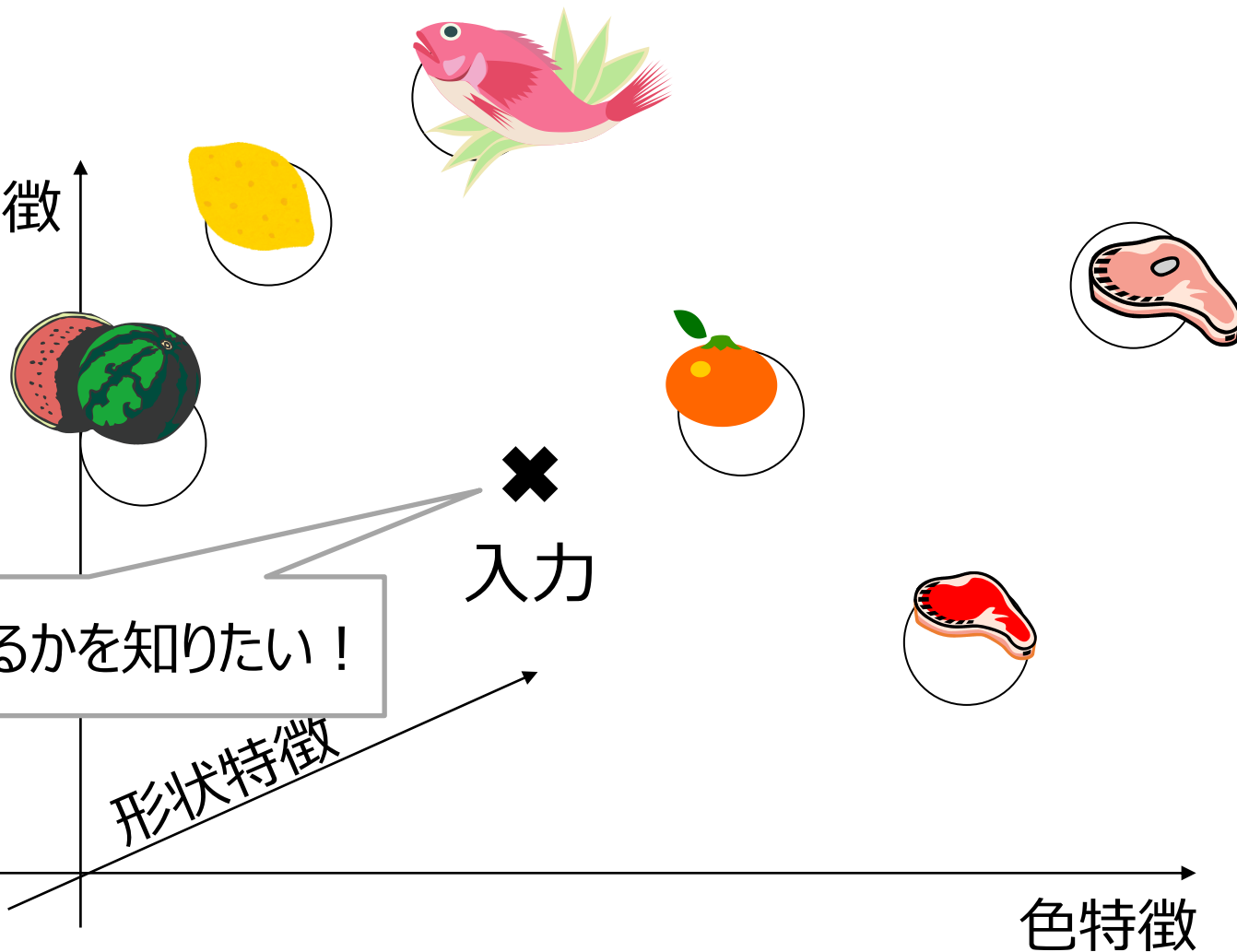
色特徴

計算機に(クラス付き)データを覚えさせる！

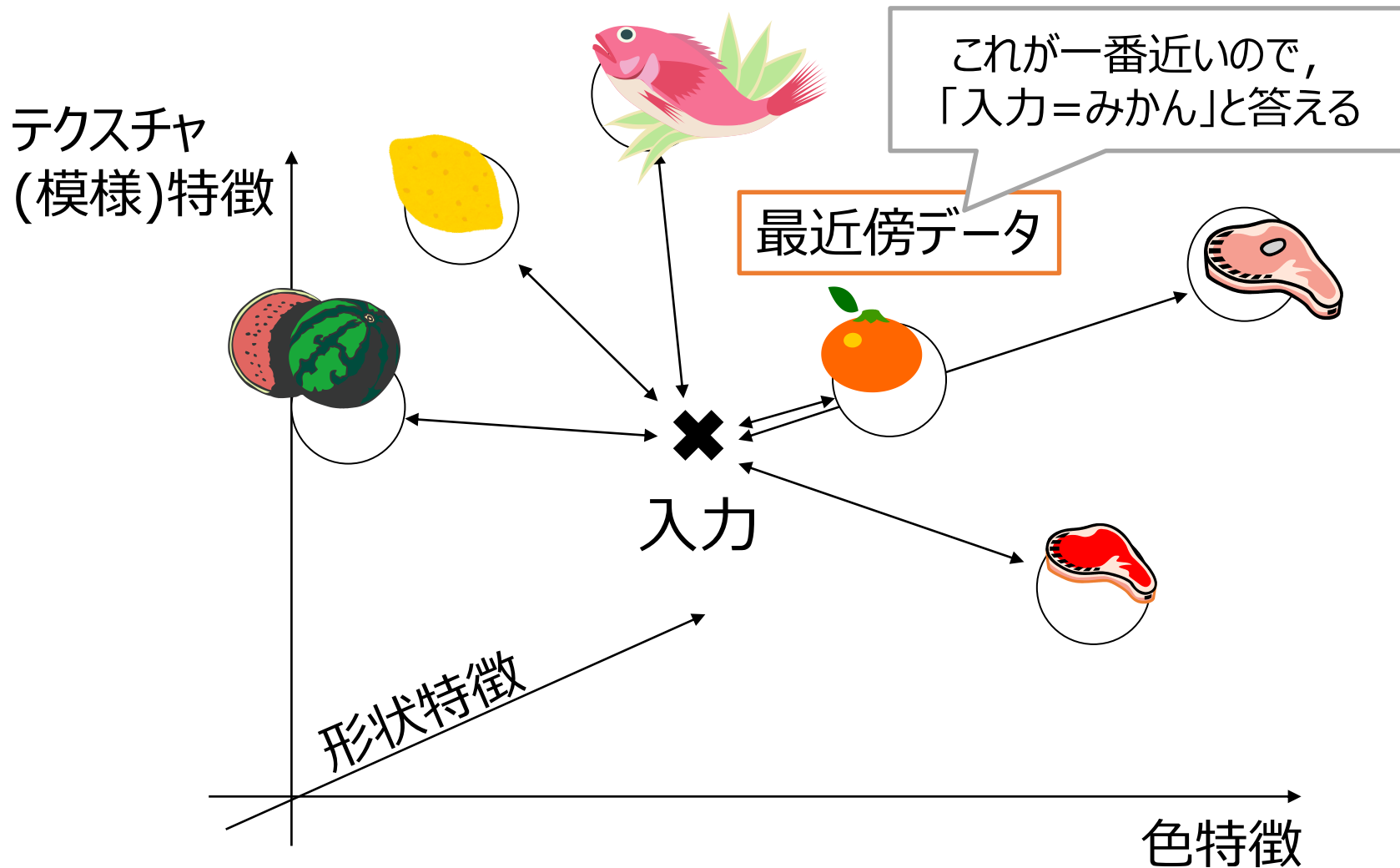


さて、クラス未知の入力データ

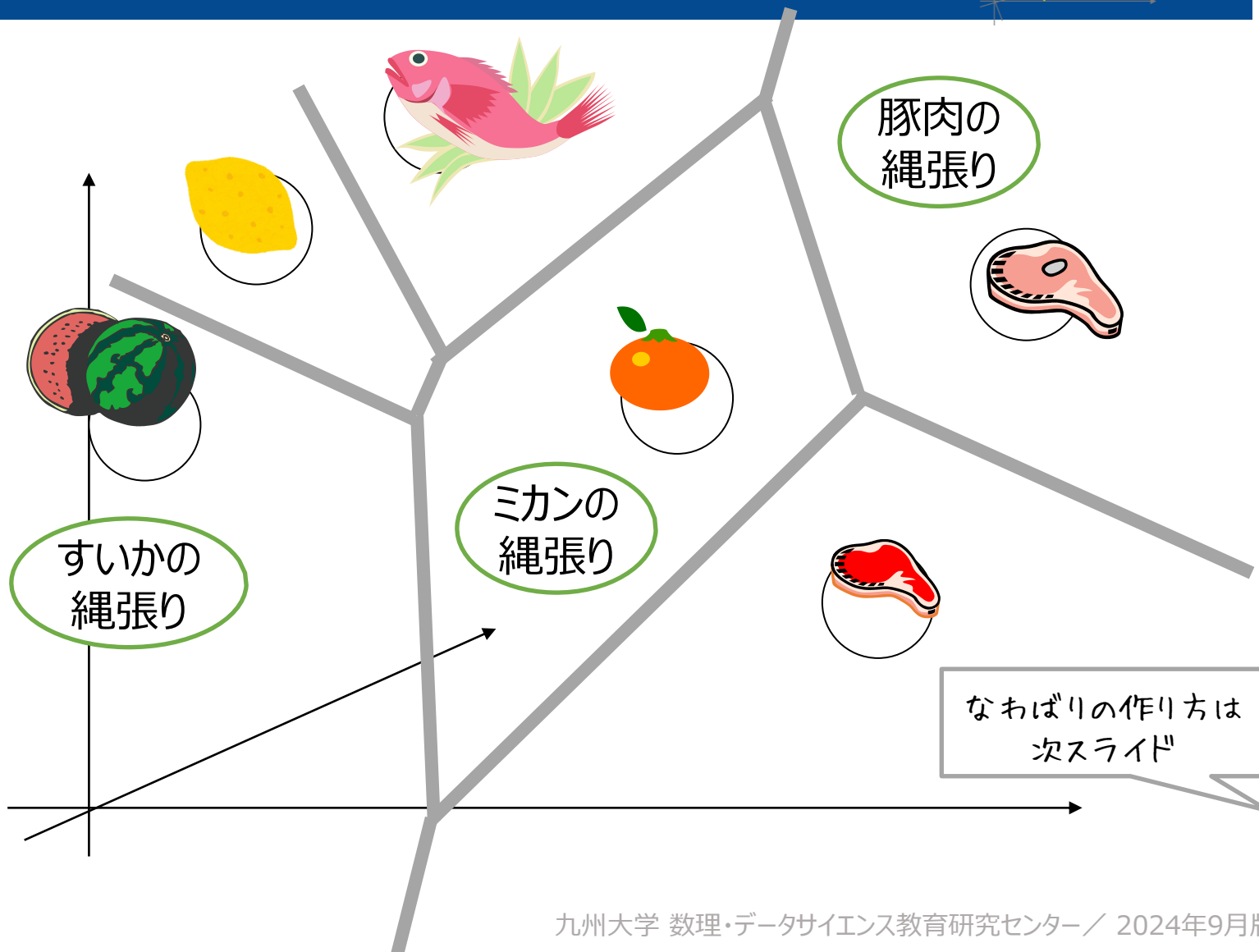
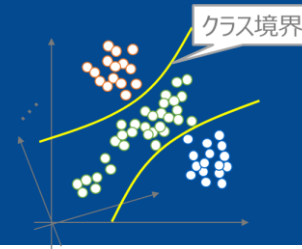
テクスチャ
(模様)特徴



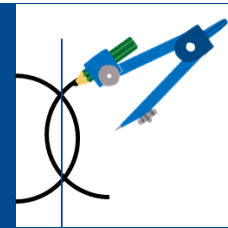
最近傍法： 距離最小(=類似度最大)のクラスに分類



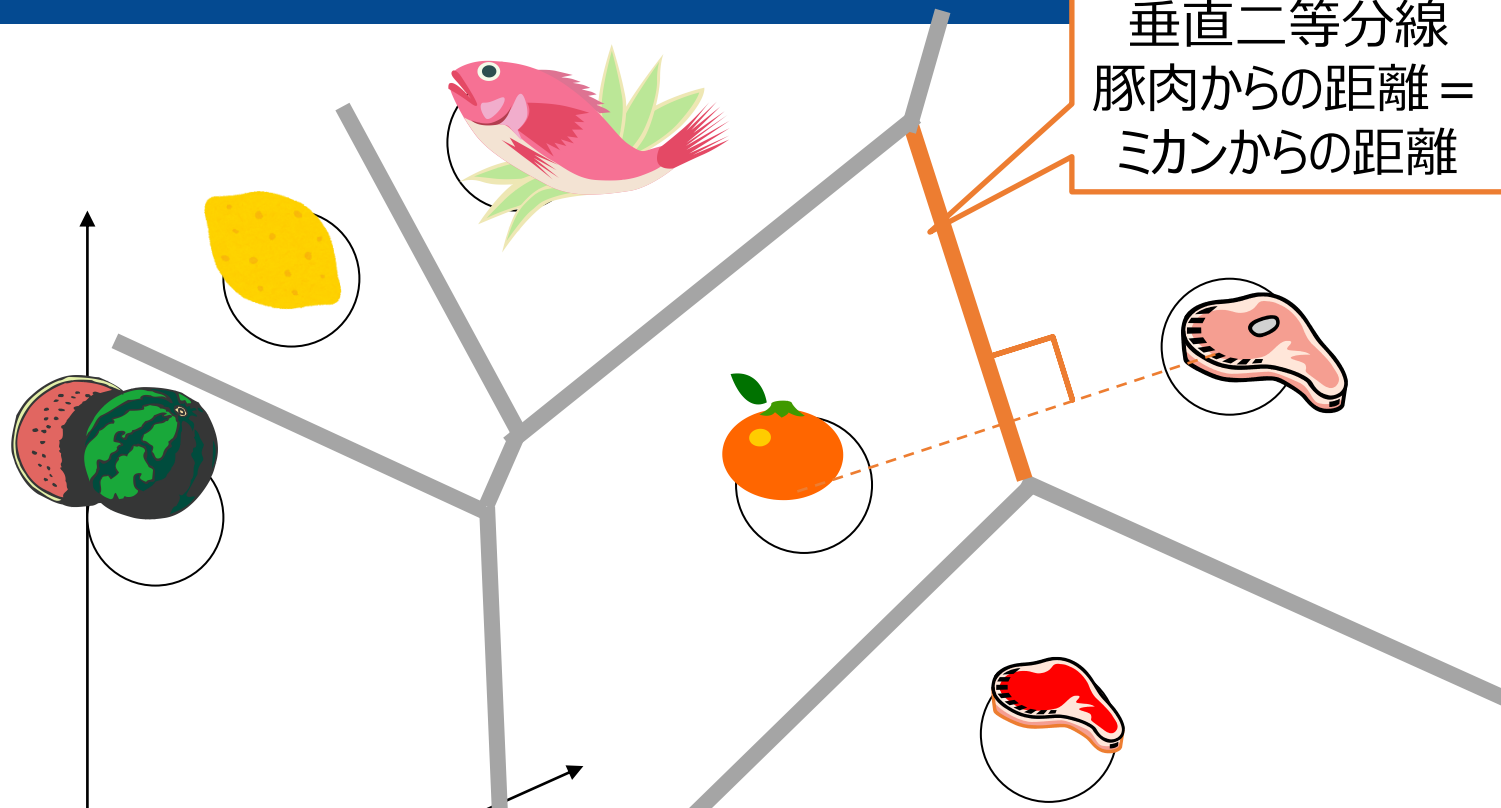
あれ？ 境界線は？ → 実は暗に「なわばり」がある



なわばりの作り方



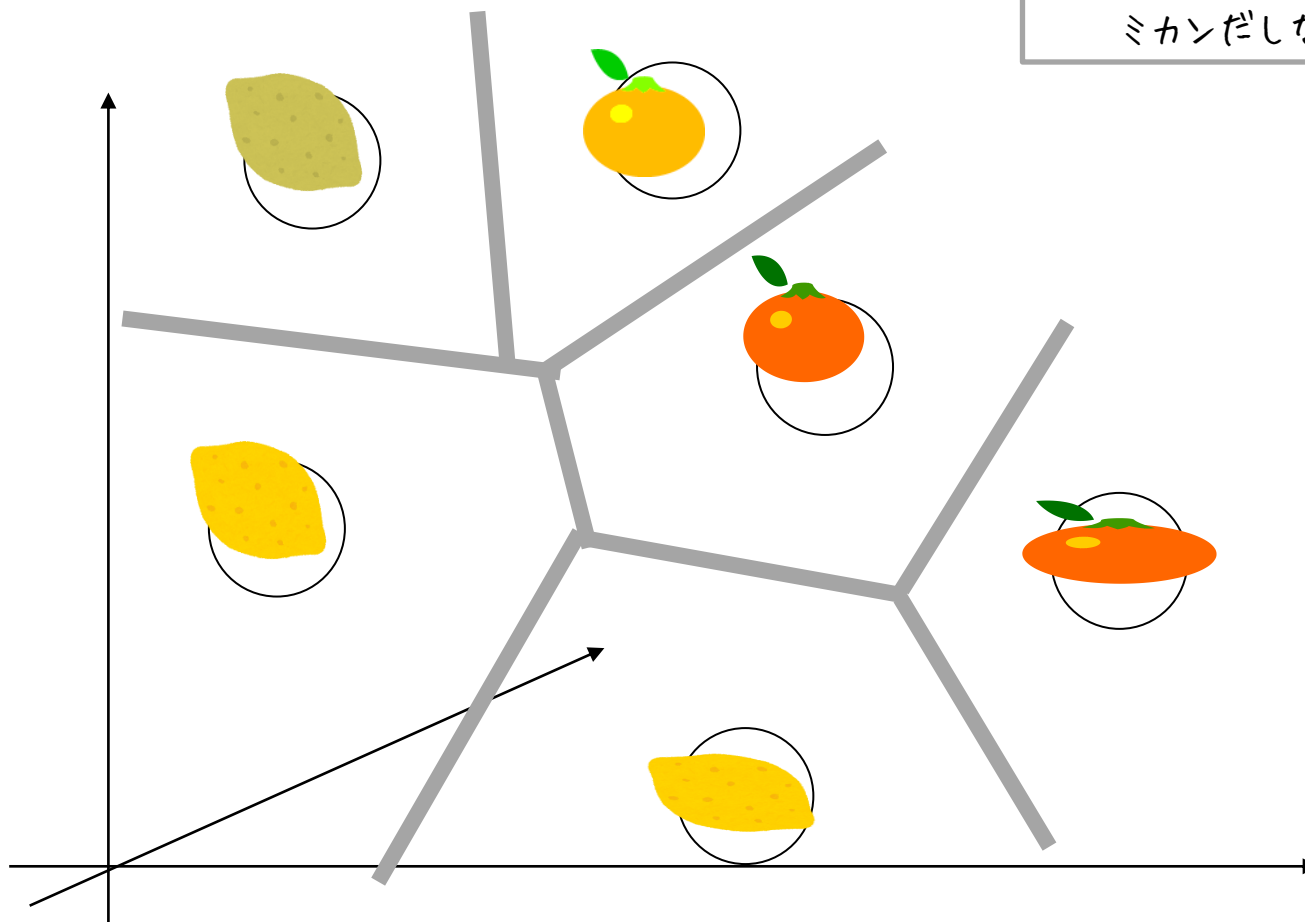
垂直二等分線
豚肉からの距離 =
ミカンからの距離



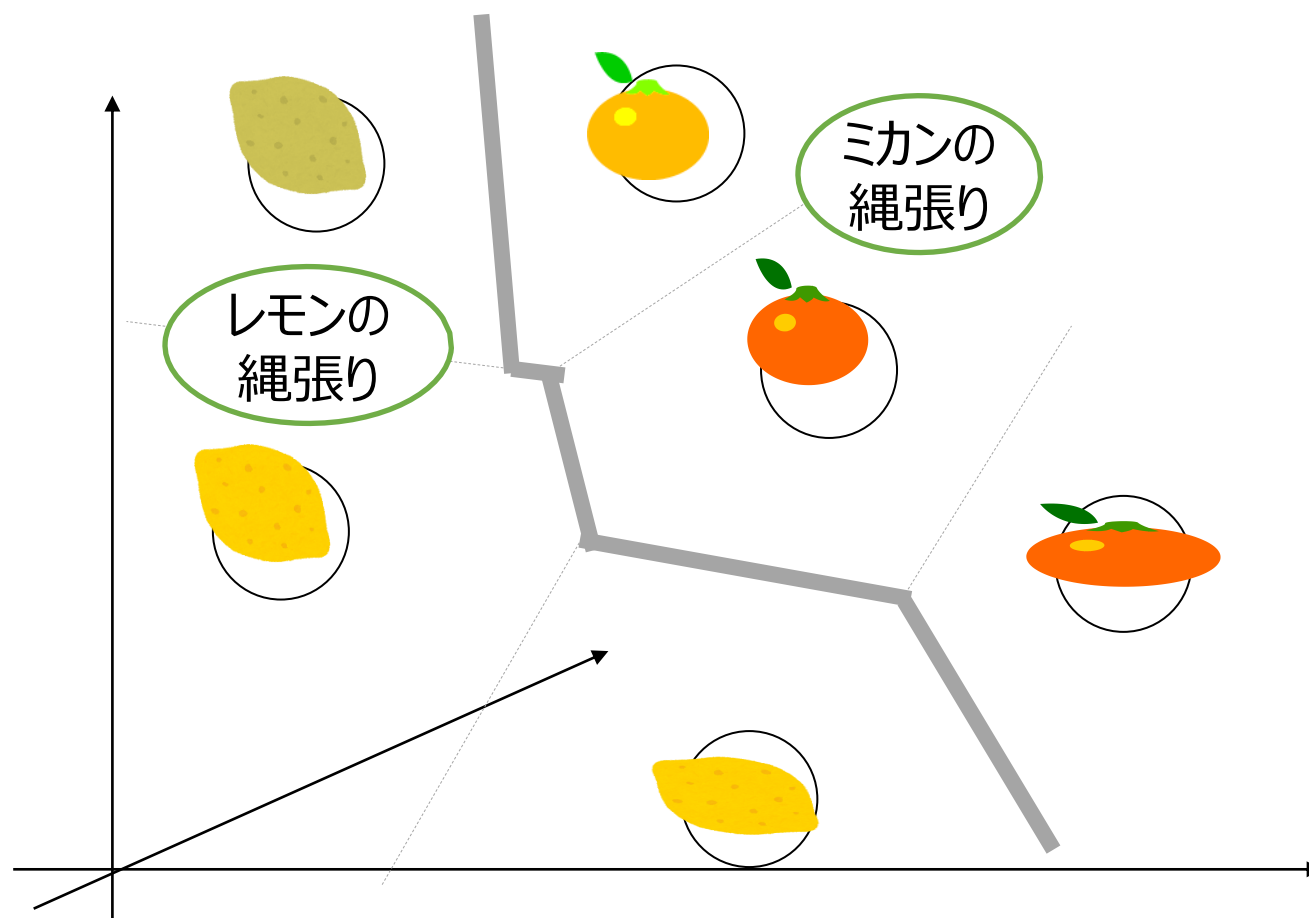
- ①この絵ではユークリッド距離で縄張りを作ってます
- ②この縄張りの様子は「ボロノイ図」と呼ばれます
(校区の設計とか, 分類以外でも色々使われます)
- ③本当は2次元ではないので, 境界「面」になります(正確には超平面)

1 クラスあたり，複数個を覚えさせてもOK! 「ミカン vs レモン」分類の場合

んー，でも，どのミカンも
ミカンだしなあ。。

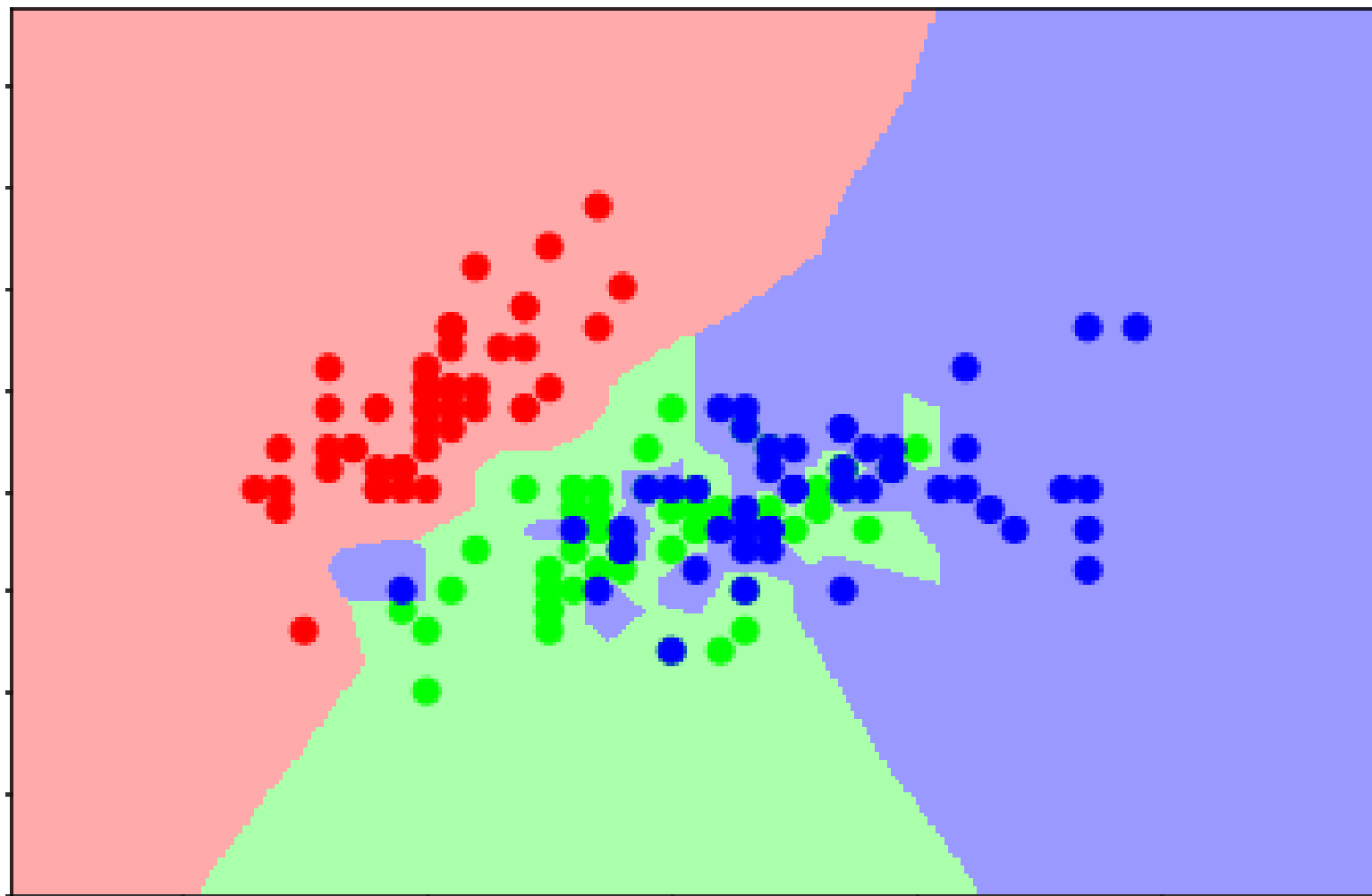


1 クラスあたり，複数個を覚えさせてもOK! 「ミカン vs レモン」分類の場合



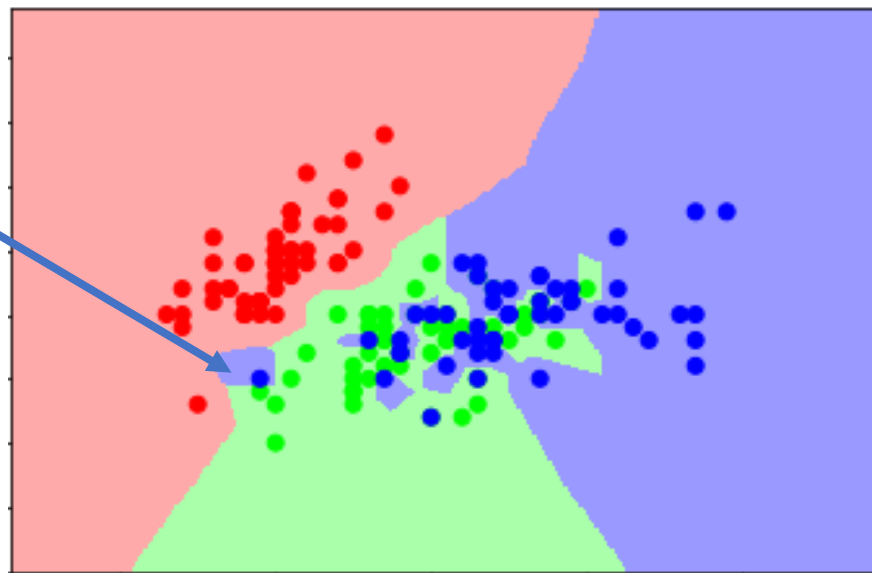
より複雑な分類境界が作られる！

3 クラスの場合の縄張りの様子 (2次元. 各クラスに多数のデータを登録)

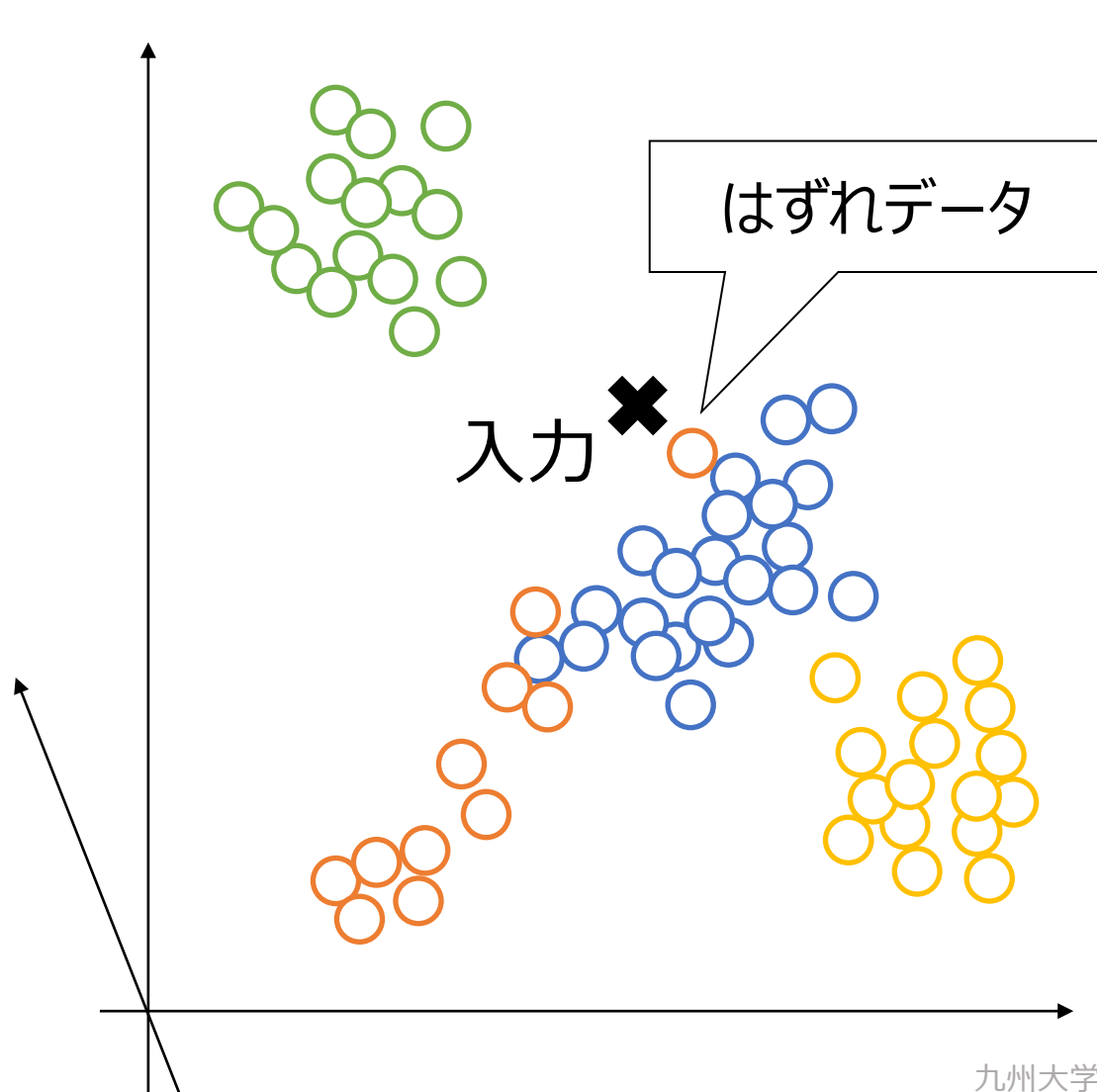


最近傍法の性質

- 非常に簡単！
 - 登録されているデータとの距離を測るだけで，分類可能
- 各クラスにつき複数のデータを登録することで，複雑な分類も可能
- 「はずれデータ」の影響大！



k 近傍法 (k-nearest neighbor, k-NN)



入力 \times

1番近い ○

2番近い ○

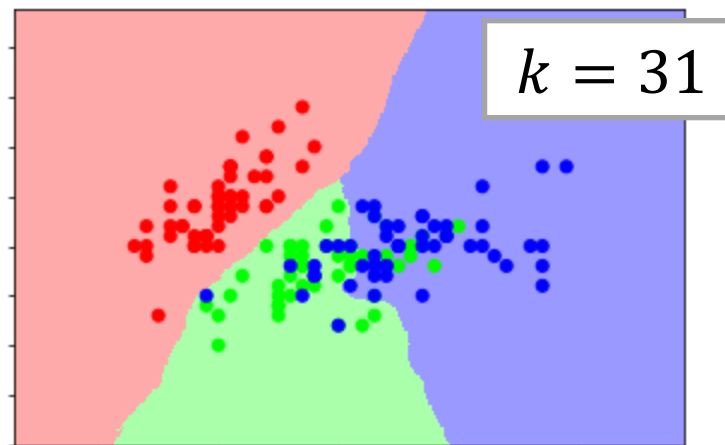
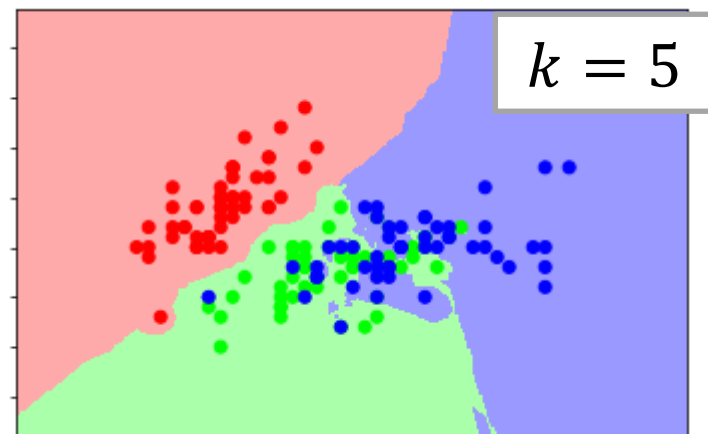
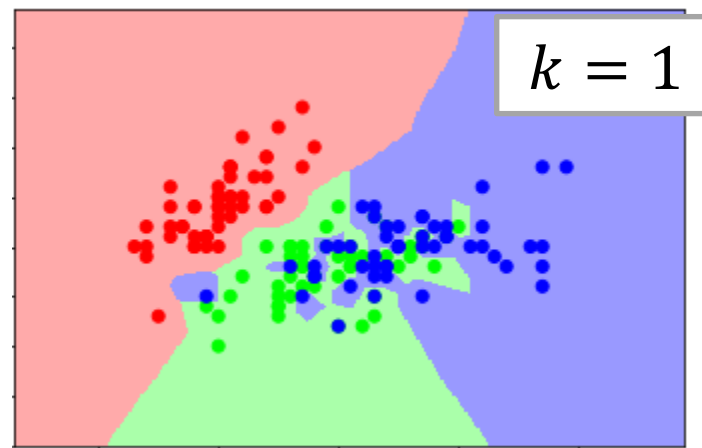
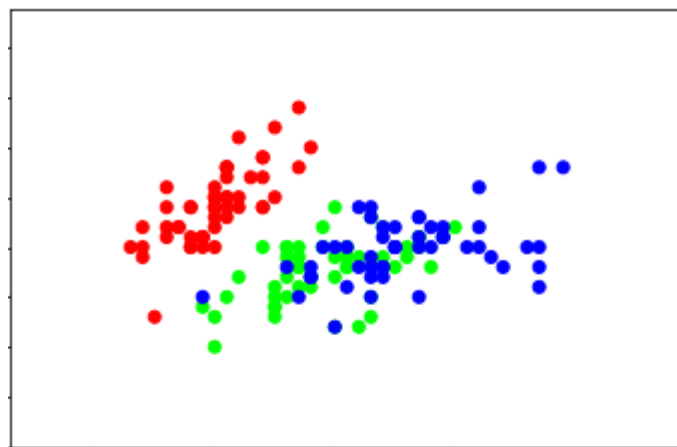
3番近い ○

多数決！

1NNだと $\times \rightarrow$ ○

3NNだと $\times \rightarrow$ ○

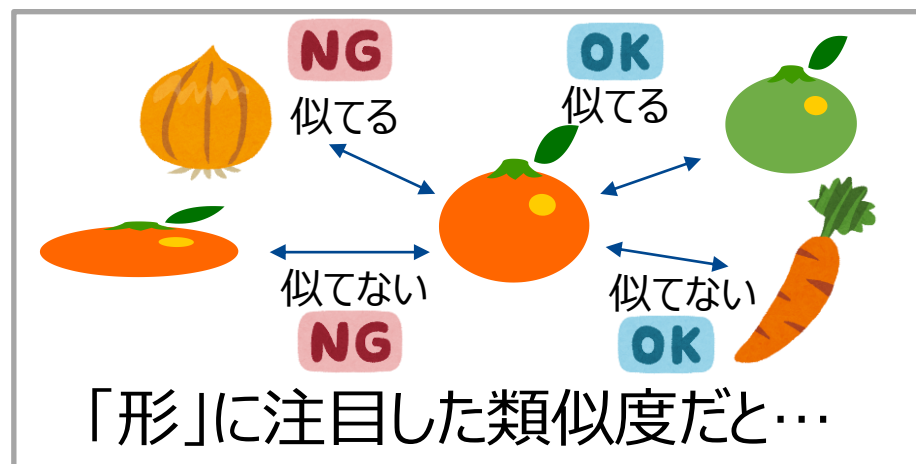
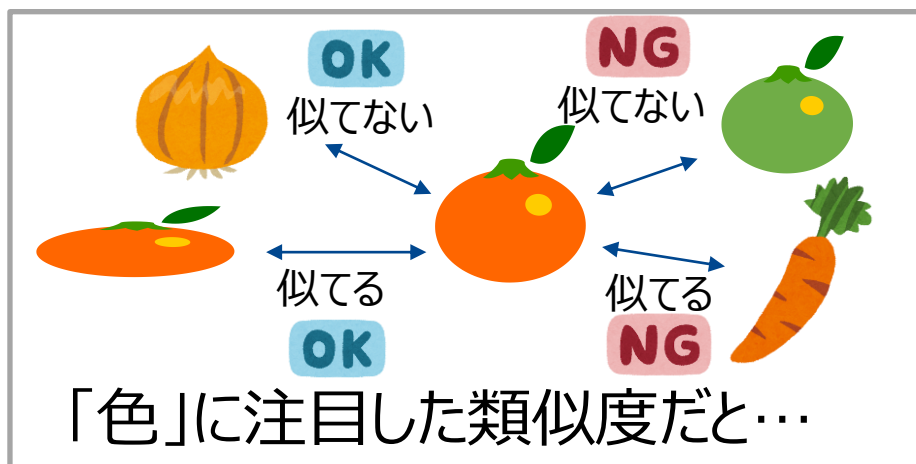
k 近傍法： k による境界の変化



- ① k を大きくすると、それだけ平滑な境界が求まる
- ② それが「分類結果に好影響を与える」とは限らない

パターン認識における 「類似度・距離」の考えどころ

- 「どこが」似ている？
 - 要するに「どのような特徴を使って類似度を評価するか」？




- 「どれぐらい」似ている？
 - 色の似てる具合をどう測る？
 - 形の似ている具合をどう測る？

数学っぽいので
「厳密な近さ」とか、
決まっていればいいのに、
意外と「あやふや」な
人間臭い話だなあ…



この辺、深入りするとハマります面白い

- そもそも「ミカン」って何だ？
- 食べられない（腐った）ミカンはミカンなのか？
- 食べ終わったミカンの皮はミカンなのか？
- ジュースにしたものはミカンなのか？
- 温州ミカンとオレンジは同じなのか？
- レモンとミカンの境界はあるのか？
- 「平均的なミカン」というのはどんなミカン？
- どうして  な絵で「ミカン」とわかるのか？
- どうして「ミ」+「カ」+「ン」の3文字が特定の果物と結びつくのか？
- ...
- などなど、認識論はギリシャ哲学時代からの難問です..
- 面白いのは「なぜか人間はそれでも生きていける」ところ...



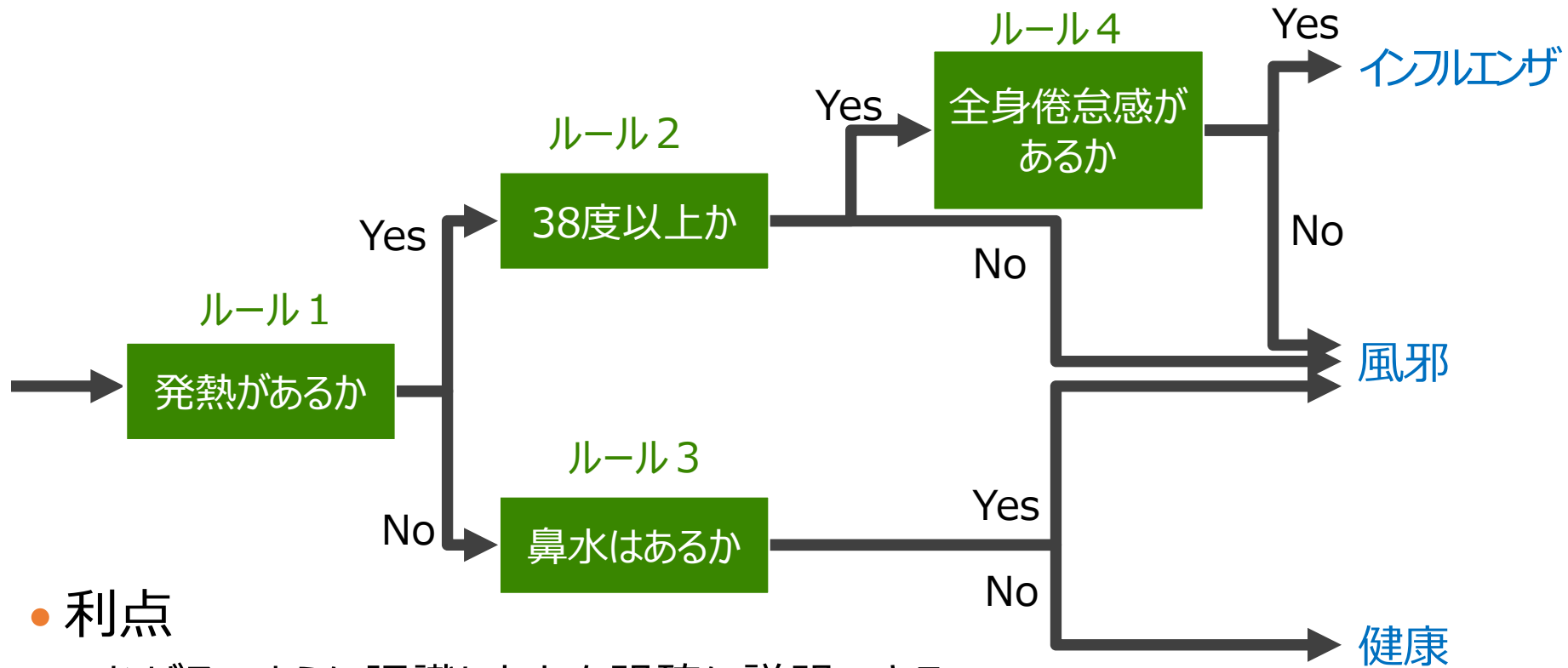
ミカんだ！



ルールベースのパターン認識

分類木とも呼ばれます

ルールベースのパターン認識



※模式図であり医学的根拠はありません

● 利点

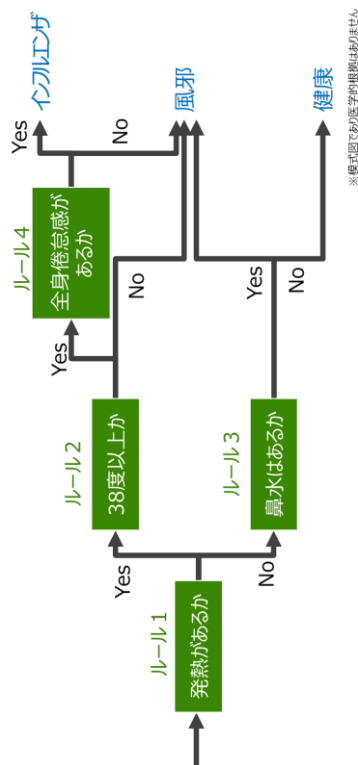
- なぜそのように認識したかを明確に説明できる

● 欠点

- どのようなルールを設定すればよいかが明らかではない
- 手動で設定したルールでは性能が出ない場合もある

先ほどのルールは「分類木」とも呼ばれます

- 90度回転させると、「木」に見える？



木の葉



木の根

- 「木」の構造は、情報学等、様々なところで活用されています！
 - グラフ理論，ネットワーク理論でも活用される，データ構造の基本なのです

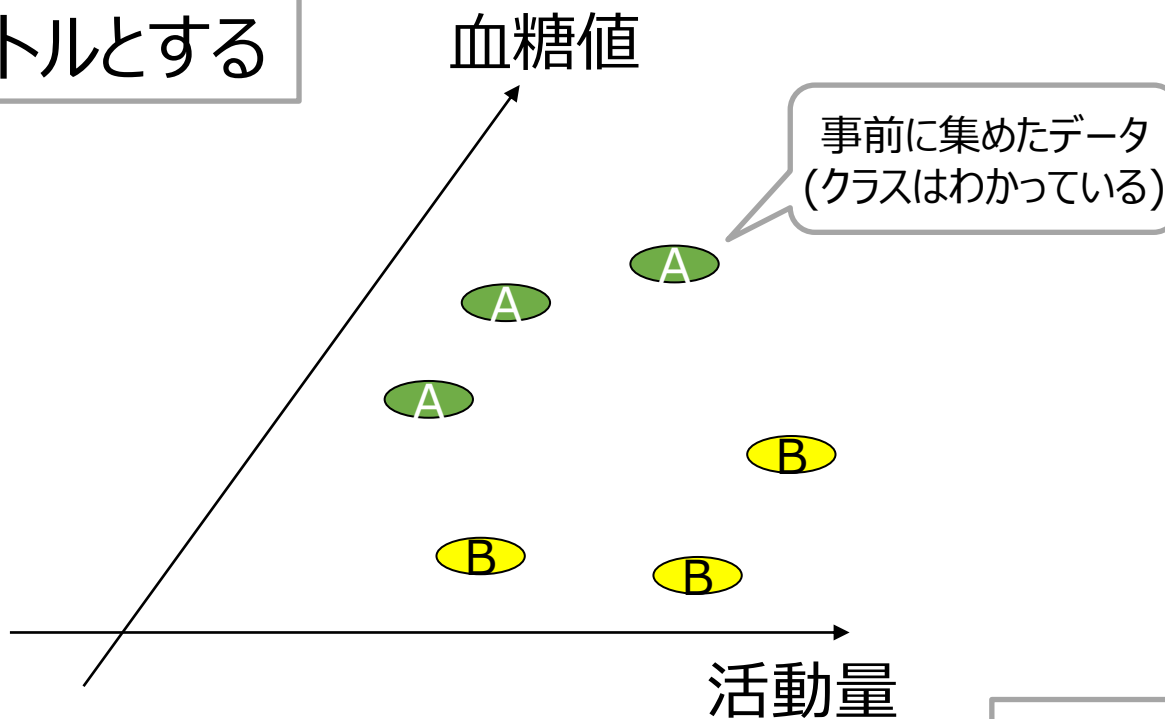
識別関数法

関数 = 「なんか入れた, なんか出すもの」
識別関数 = 「データ入れたら, その〇〇らしさを出すもの」

識別関数とは？

例：診断マシンを作りたい！

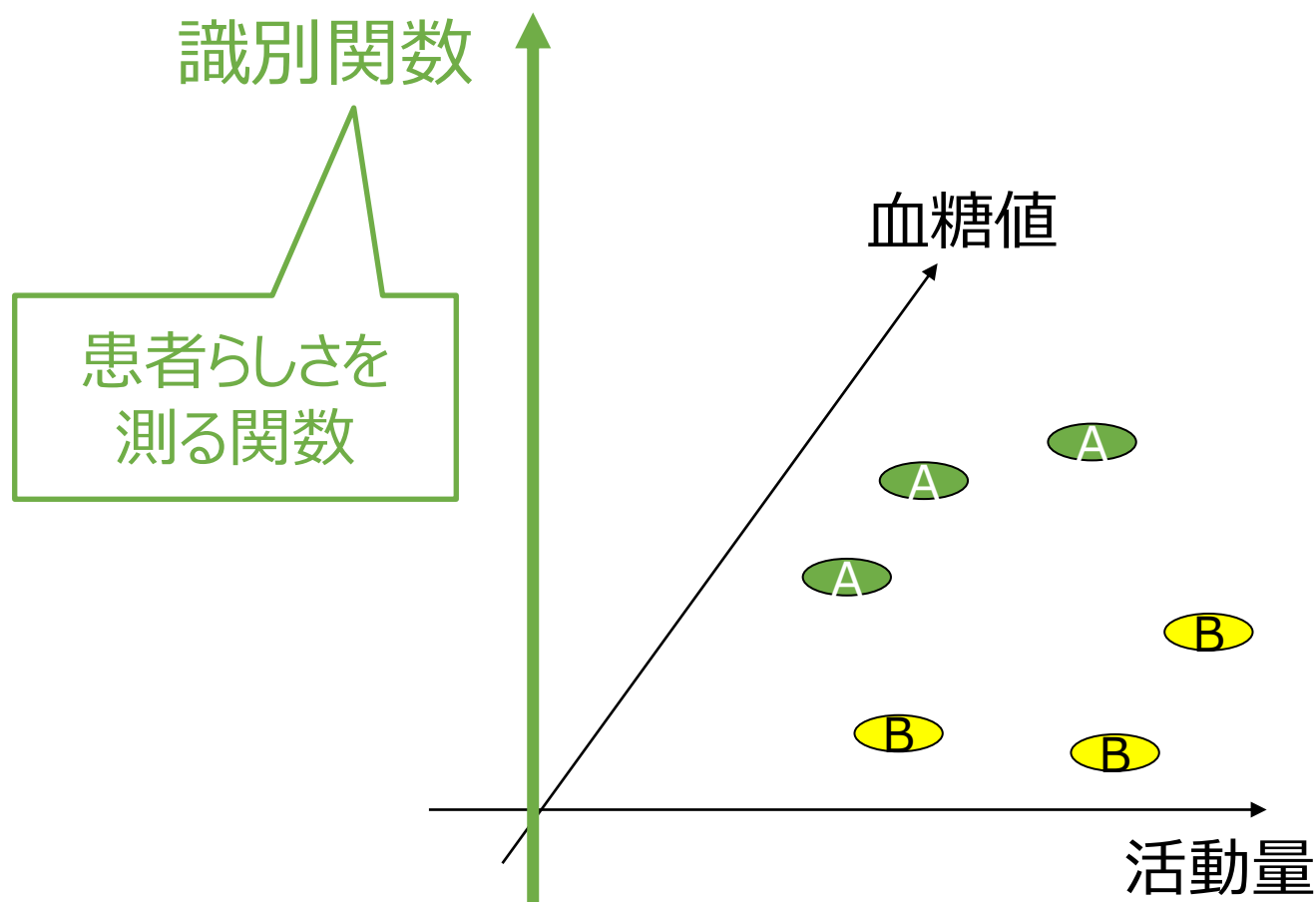
各データ＝
2次元ベクトルとする



A: 患者(糖尿病)
B: 健康

識別関数とは？

例：診断マシンを作りたい！



識別関数とは？

例：診断マシンを作りたい！

識別関数

識別関数 =
患者らしさを測る関数

血糖値

患者3の識別
関数値 = 10
(患者らしさ大)

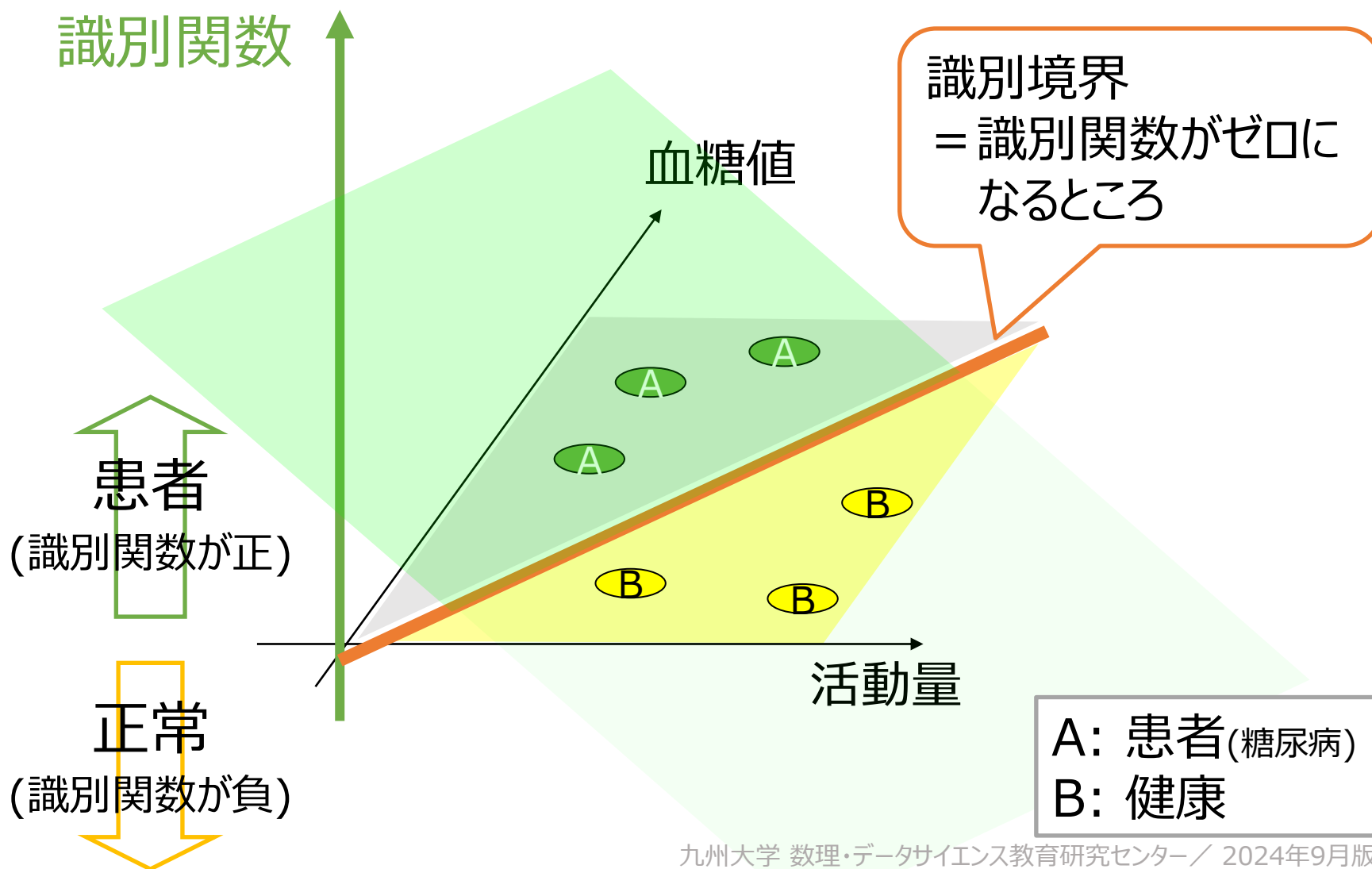
健常者3の識別
関数値 = -1
(患者らしさ小)

活動量

A: 患者(糖尿病)
B: 健康

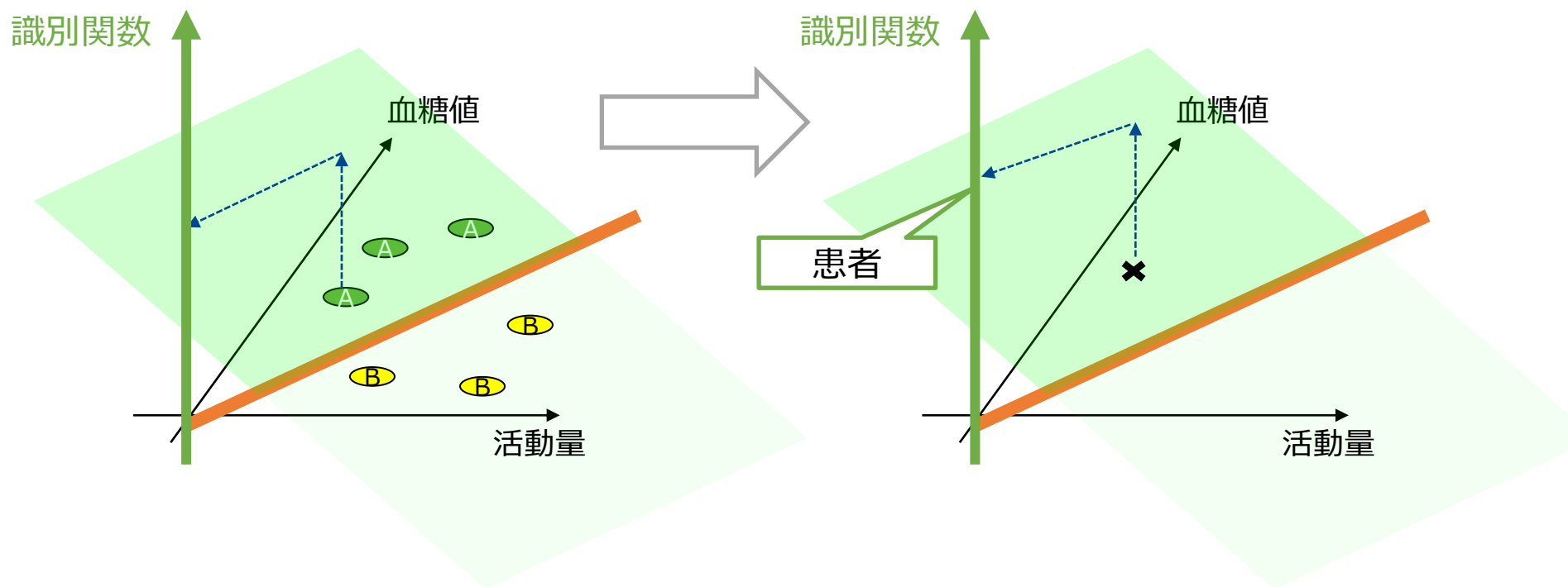
識別関数とは？

例：診断マシンを作りたい！



識別関数によるデータ分類の手順

- 【学習】クラスがわかったデータを用いて識別関数を決定する
- 【利用】未知データを識別関数に入力し、関数値を求めることで、分類を行う（以下の例では、関数値が正ならば患者）



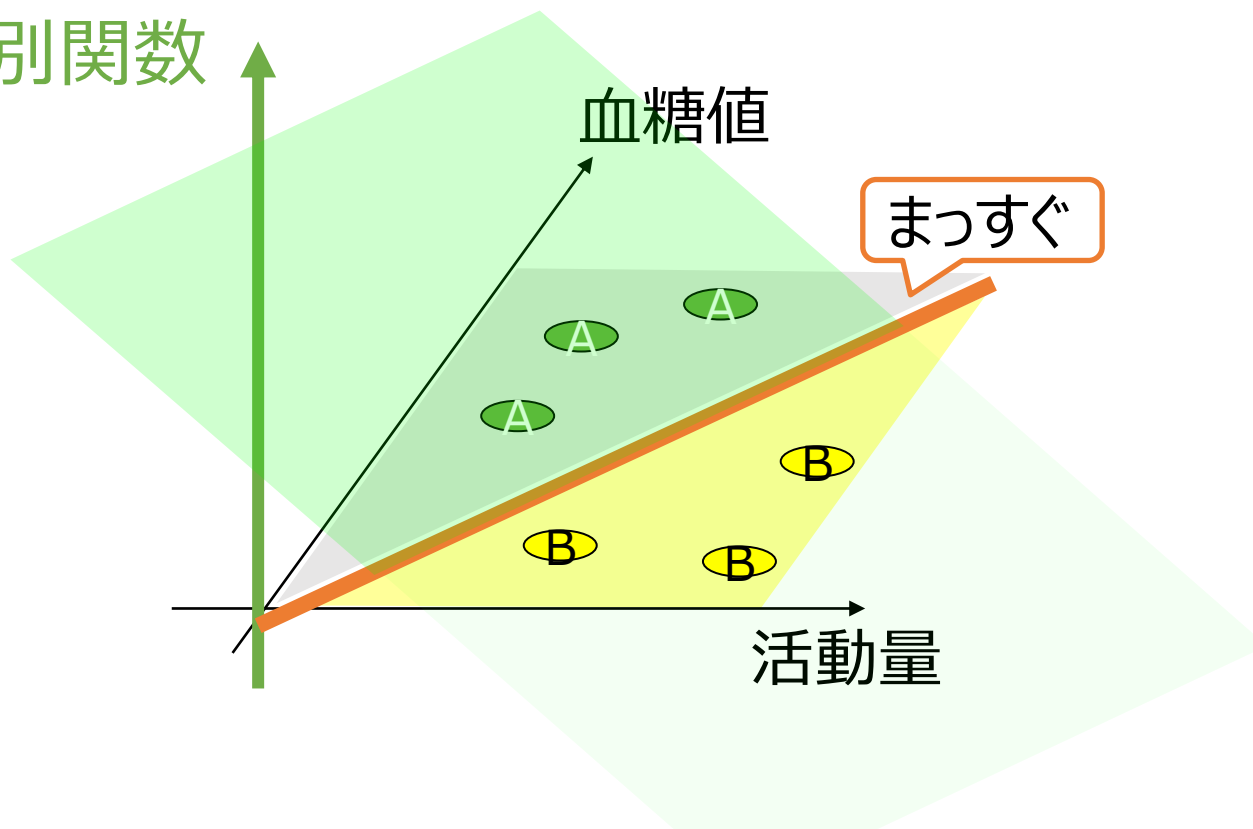
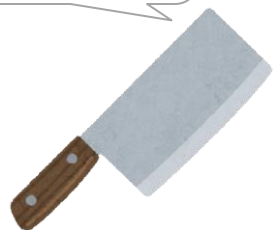
こんなまっすぐな境界でいいのか？(1/2)

識別関数

血糖値

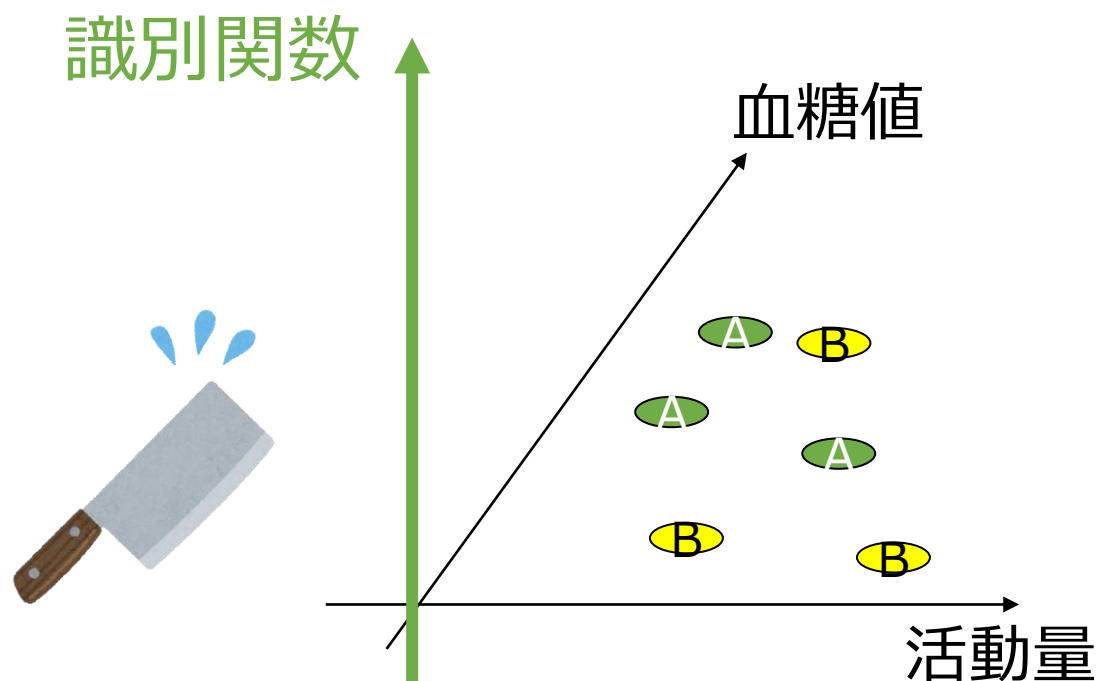
まっすぐ

まっすぐな包丁による
切り口は
やっぱりまっすぐ



- この「曲がっていない平面的な識別関数」を**線形識別関数**と呼ぶ
- 線形識別関数による境界も、やはりまっすぐに

こんなまっすぐな境界でいいのか？(2/2)

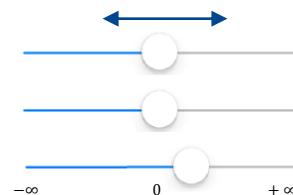


- もう線形識別関数では完全分離できない！
- 対策
 - 完全分離はあきらめる
 - より柔軟な識別関数（ex. ニューラルネットワーク（後述））を使う

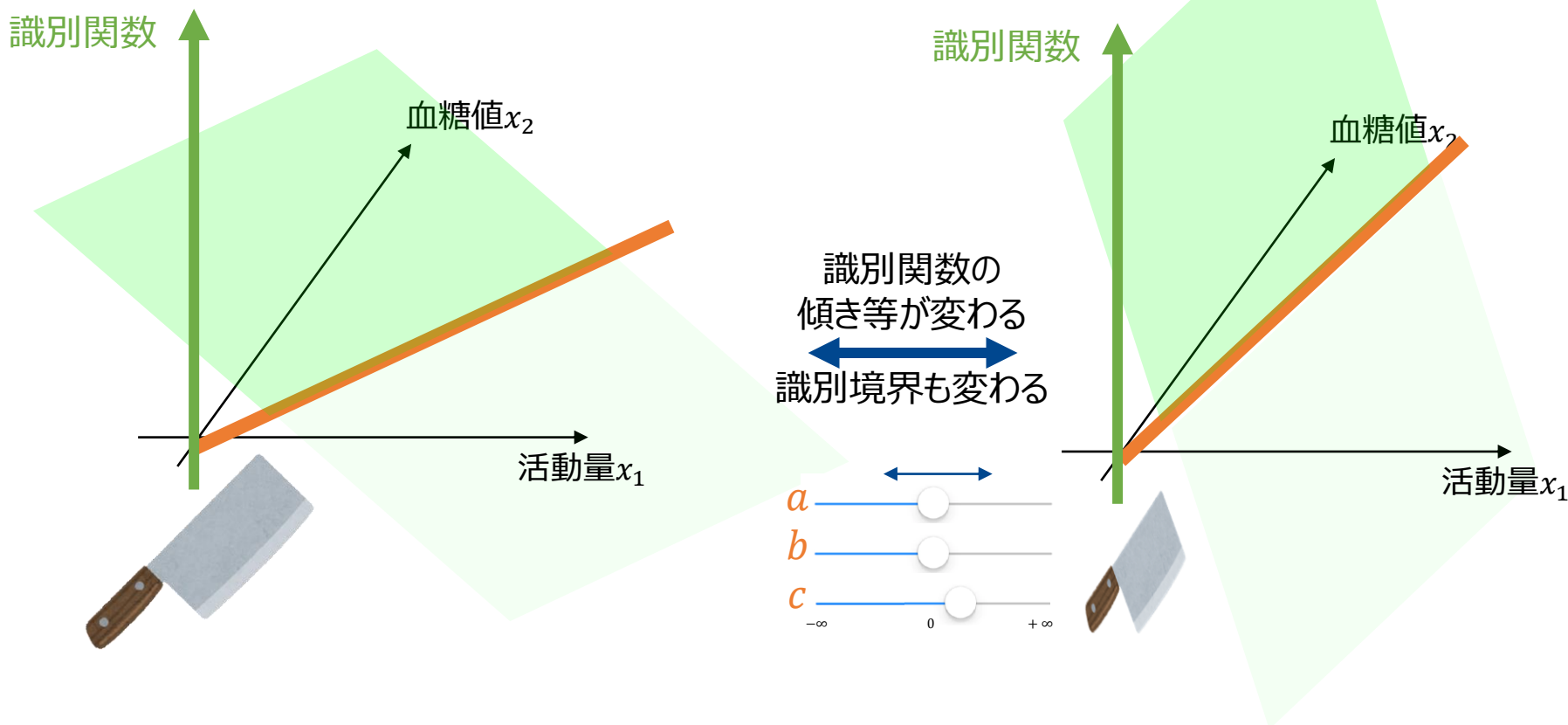
なぜ識別関数？ 最近傍法でいいのでは？

識別関数のメリット：

- 境界の形を，好きなようにコントロールできる
 - まっすぐ（線形）から，ぐにやぐにやまで
- はずれデータの影響を小さくできる
- 最適性など，理論的な性能保証がある
- パターン認識の問題を「識別関数のパラメータを決める問題」と捉えることができる！
 - パラメータ＝「調節スライダー」



パラメータを決める問題？



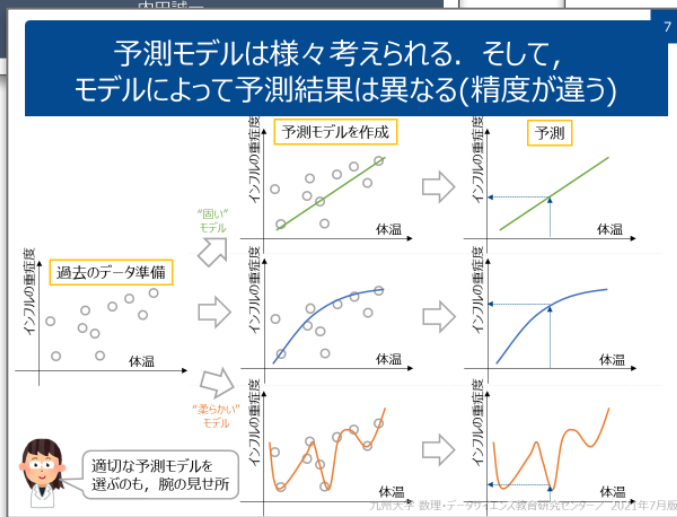
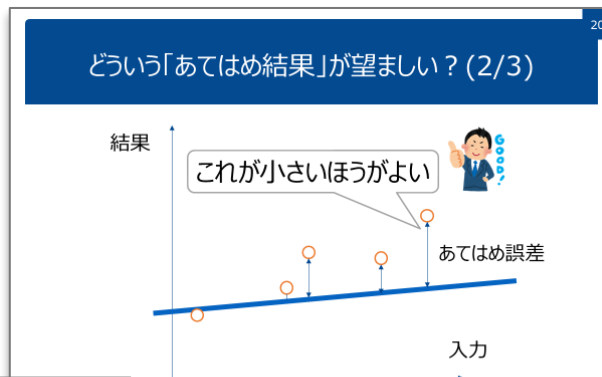
- 集めたデータが正しく分類できるように、**スライダー**(パラメータ)を調節
 - 上の線形識別関数は $y = ax_1 + bx_2 + c$ なので、パラメータは a, b, c の3つ

回帰分析の話を思い出していただけると嬉しい (パターン認識は回帰分析とも関係しているのです)

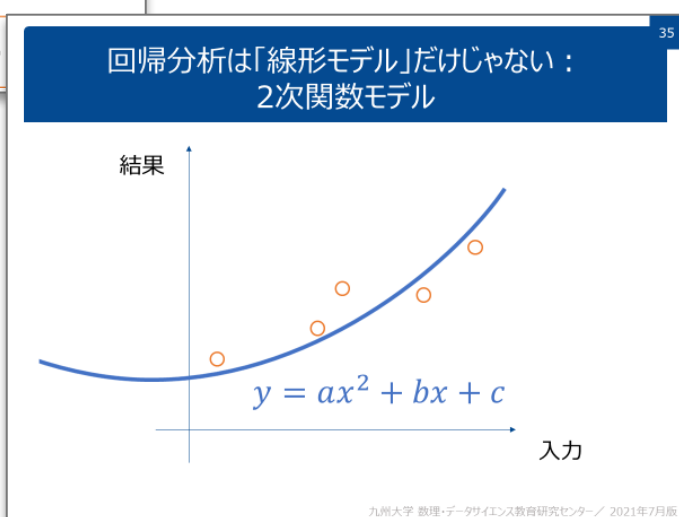
電気情報工学科
データサイエンス序論

回帰分析

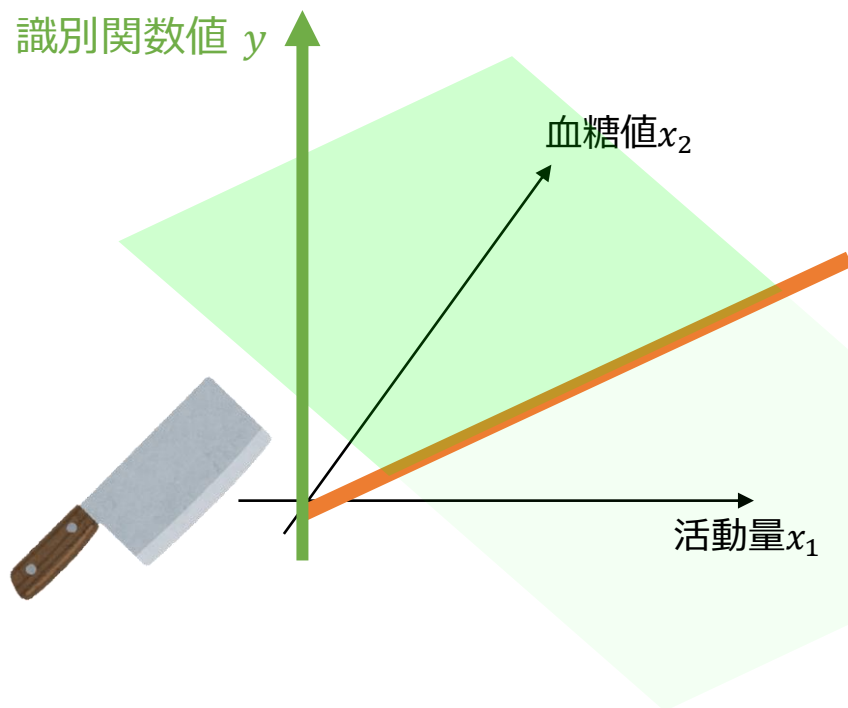
システム情報科学研究院情報知能工学部門
内田 誠一



$$\sum_i \text{第} i \text{データのあてはめ誤差} \rightarrow \text{最小化}$$



おや, どこかで見たような...



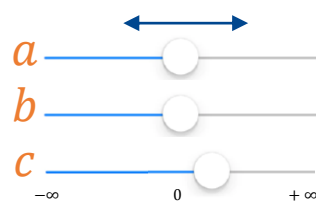
線形識別関数値

$$y = ax_1 + bx_2 + c$$

2つの3次元ベクトルを使うと, こう書ける

$$y = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

同じ要素どうしを掛けて, 全部足す

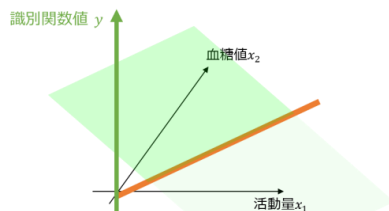


識別関数の
傾きなどを決める
パラメータ

識別したいデータ
(x_1, x_2)

...にオマケの1が付いた

というわけで、線形識別関数 = 内積



$$y = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

線形識別関数



以前やった内積

思い出そう：内積

内積の書き方4種
(教科書で違ってたりする)

$x \cdot y$
 (x, y)
 $\langle x, y \rangle$
 $x^T y$

• ご存じのはず

$$x = \begin{pmatrix} 3 \\ 5 \end{pmatrix}, y = \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{の内積} \rightarrow x \cdot y = 3 \times 6 + 5 \times 1 = 23$$

• 要するに、「要素どうしの積をとって、全部足す」

• その原理で、何次元ベクトルでも計算可能

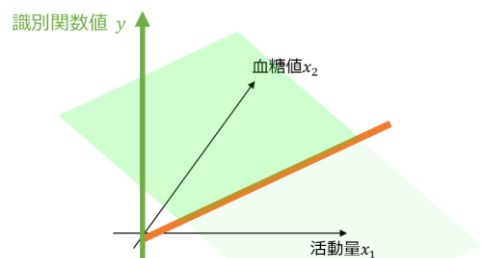
$$\begin{pmatrix} 3 \\ 5 \end{pmatrix} \text{と} \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{の内積} \Rightarrow \left. \begin{array}{l} \begin{pmatrix} 3 \\ 5 \end{pmatrix} \times \begin{pmatrix} 6 \\ 1 \end{pmatrix} = 18 \\ \phantom{\begin{pmatrix} 3 \\ 5 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \end{pmatrix}} = 5 \end{array} \right\} 18 + 5 = 23$$

$$\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix} \text{と} \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix} \text{の内積} \Rightarrow \left. \begin{array}{l} \begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix} \times \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix} = 18 \\ \phantom{\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix}} = 5 \\ \phantom{\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix}} = 4 \end{array} \right\} 18 + 5 + 4 = 27$$

※この調子で、4次元でも、100万次元でも可能

内積は、成分量計算だけでなく、平面の方程式にも表れるすごいやつ...

内積なので、線形識別関数はベクトル表現を使ってシンプルに書ける



線形識別関数



識別関数の
傾きなどを決める
パラメータ

ベクトル \mathbf{v} と書く

$$y = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

識別したいデータ
(x_1, x_2)

…にオマケの1が付いた

ベクトル \mathbf{x} と書く

とりあえず
オマケは気にせず

$$y = \mathbf{v} \cdot \mathbf{x}$$

$y > 0$ なら患者, $y < 0$ なら正常

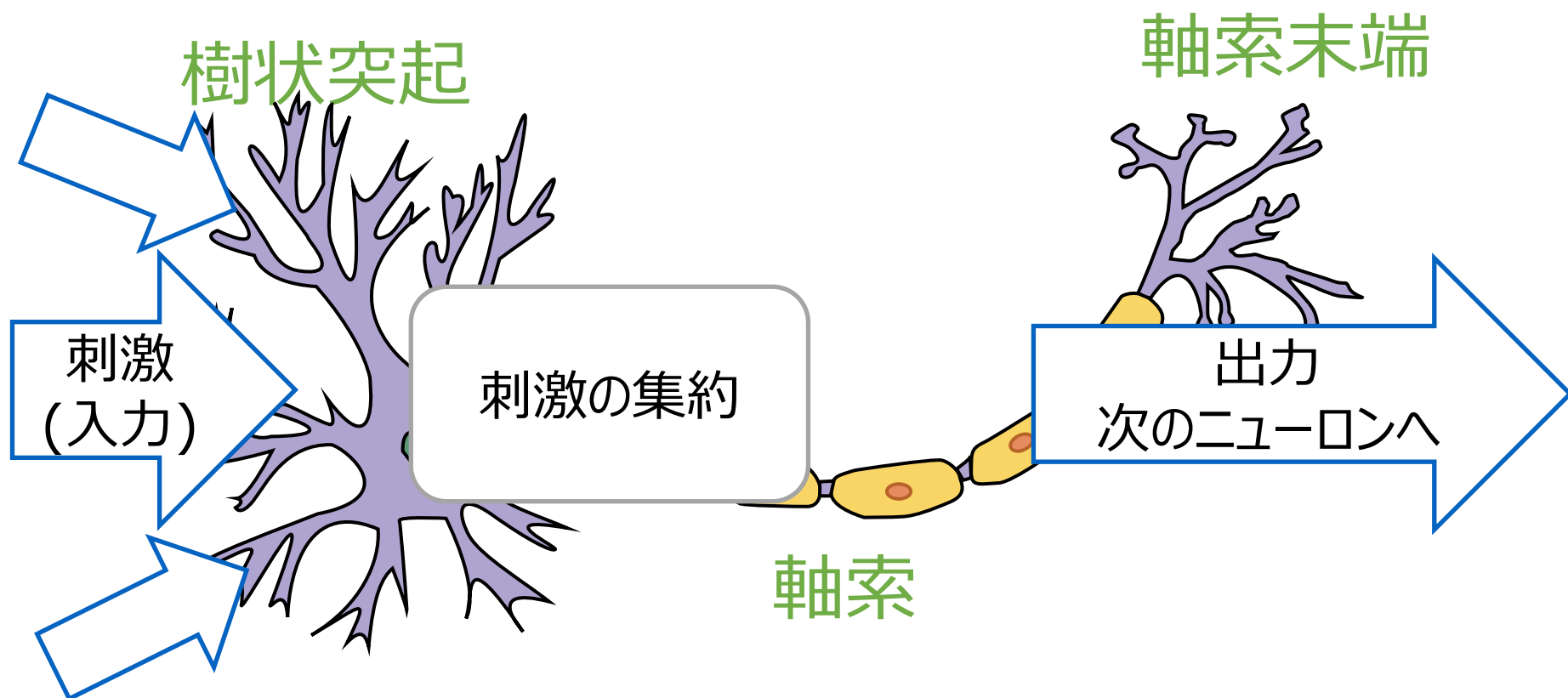
※この例では、データが(x_1, x_2)の2次元ですが、例によって何次元でもOKです

またもや

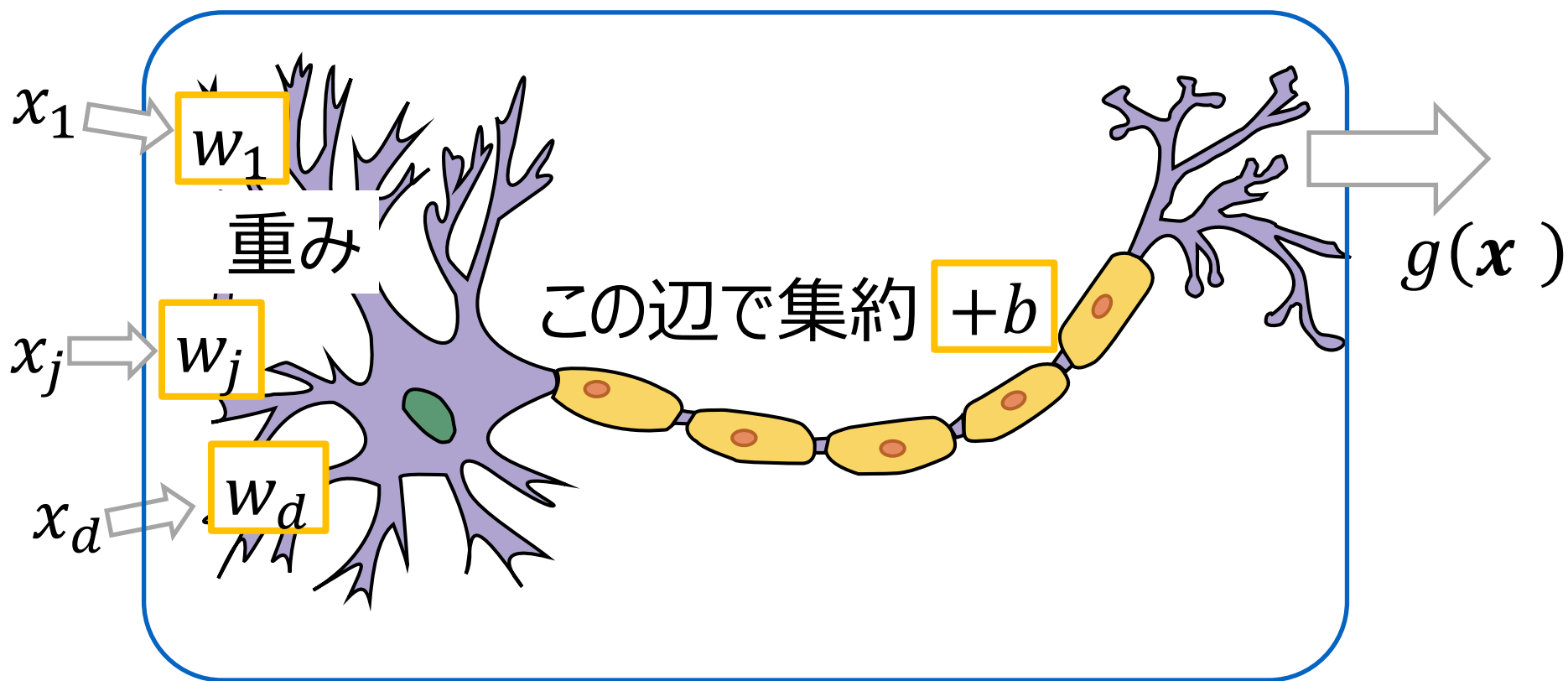
ニューロンと内積

深層ニューラルネットワークによる
パターン認識①

神経細胞（ニューロン）



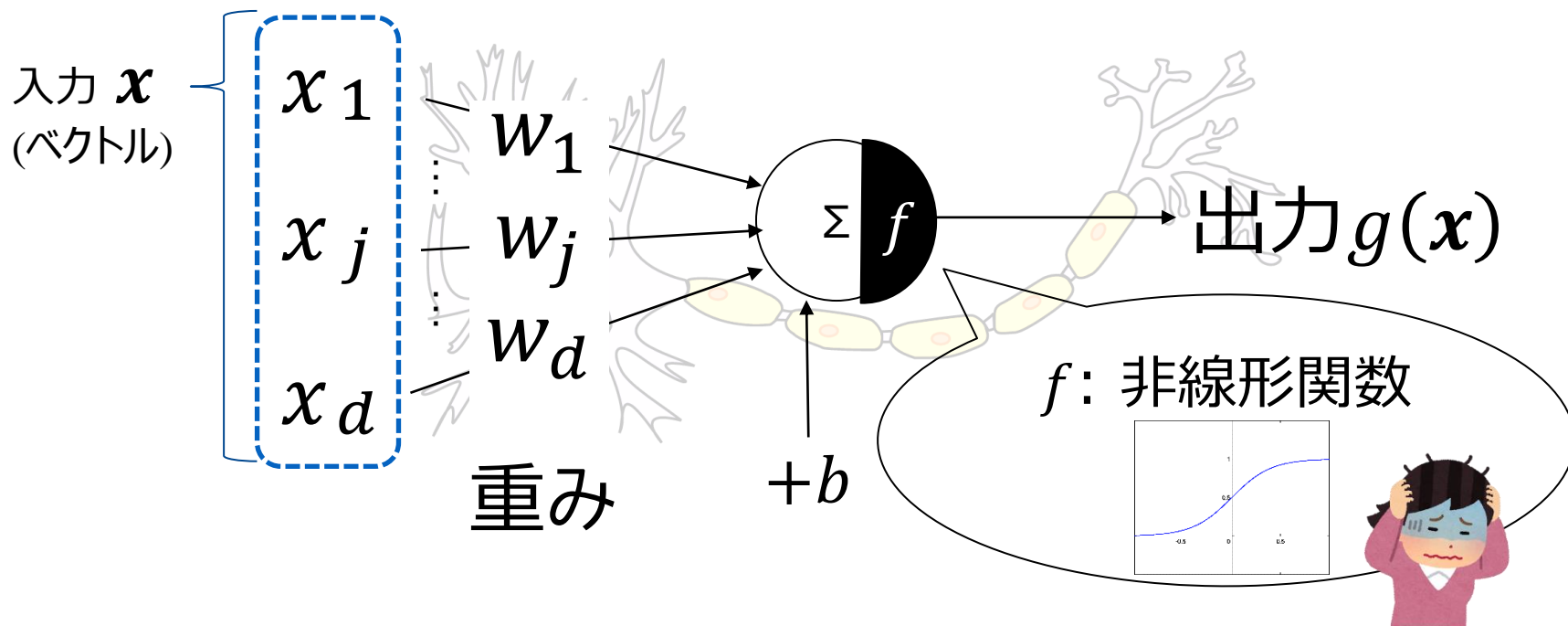
ニューロンの計算モデル



ニューロンの計算モデル

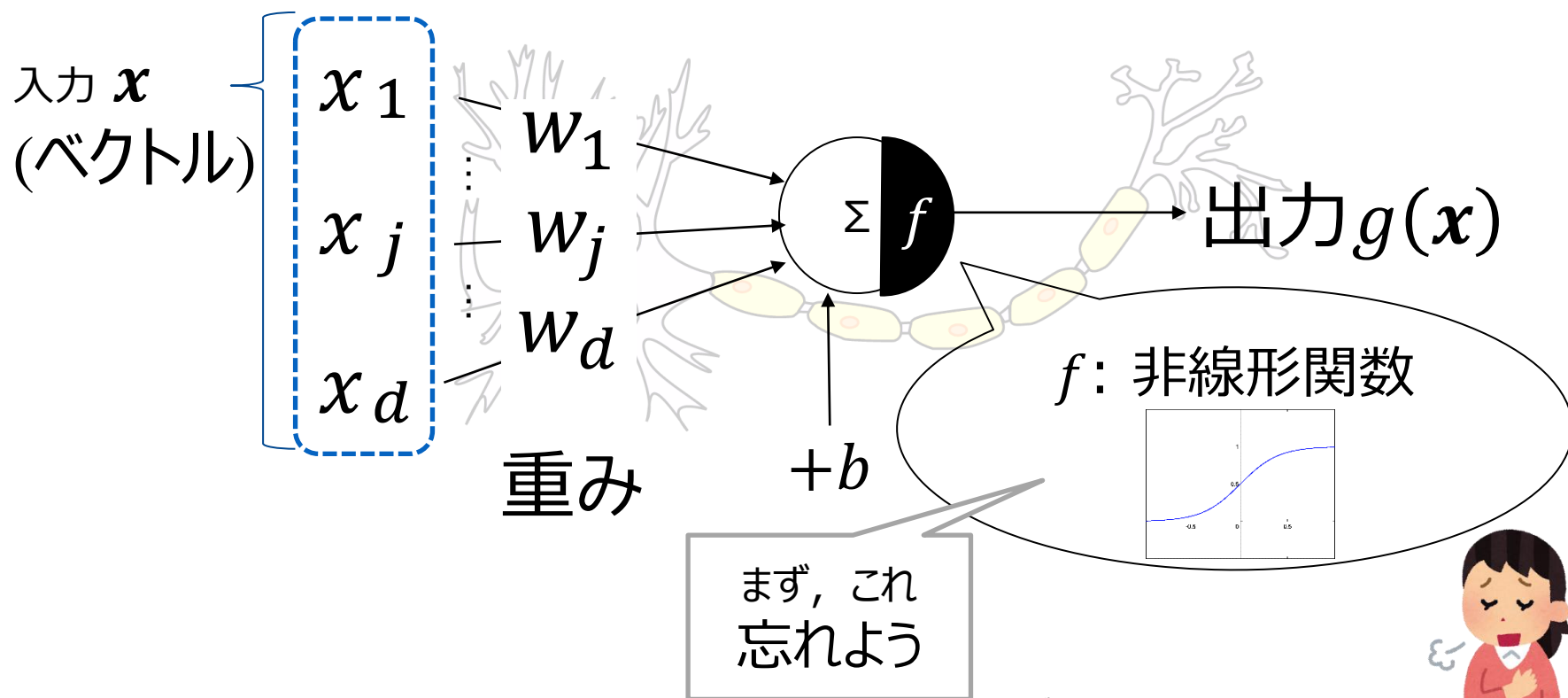
$$g(x) = f\left(\sum_j w_j x_j + b\right)$$

何じゃこりゃ



これが何なのか，単純化してみる

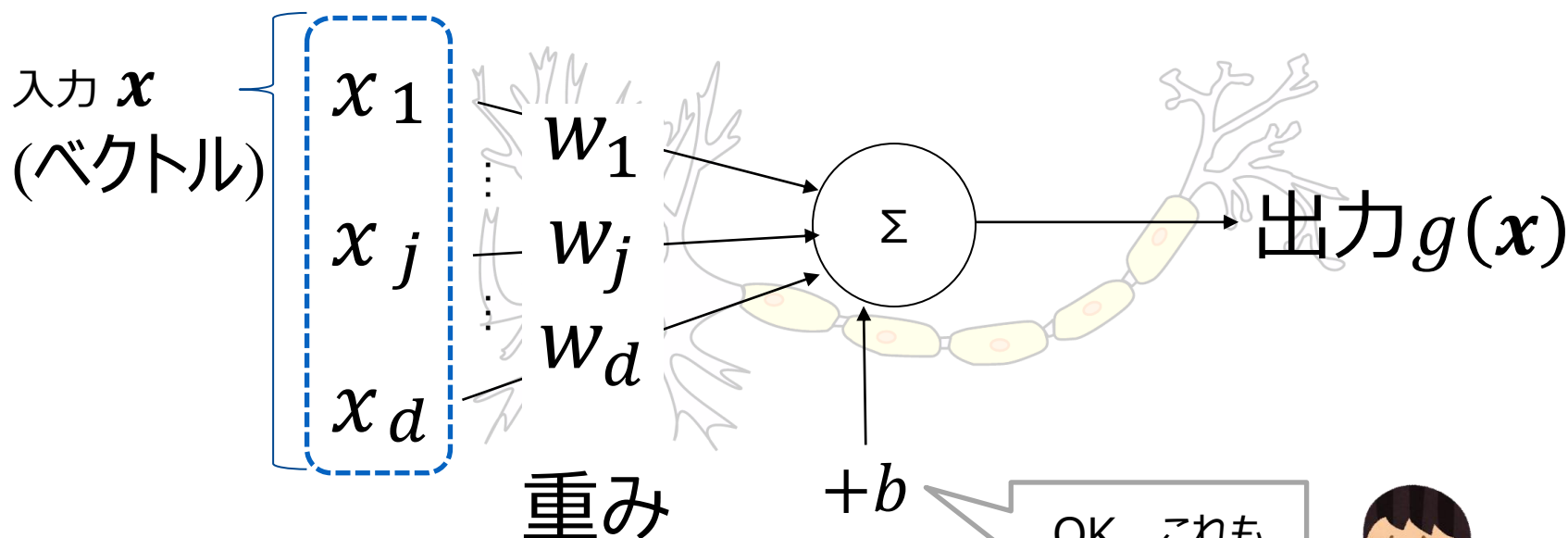
$$g(x) = f\left(\sum_j w_j x_j + b\right)$$



少しは簡単になったが...

$$g(x) = \sum_j w_j x_j + b$$

まだよくわからん

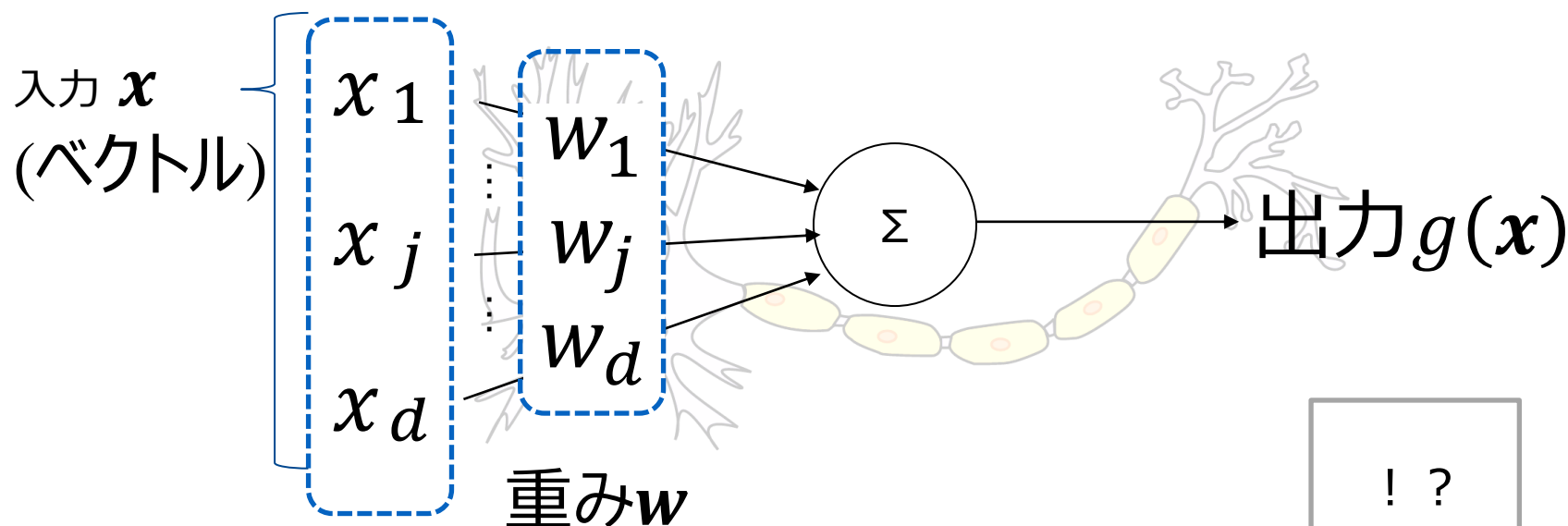


OK, これも
忘れよう

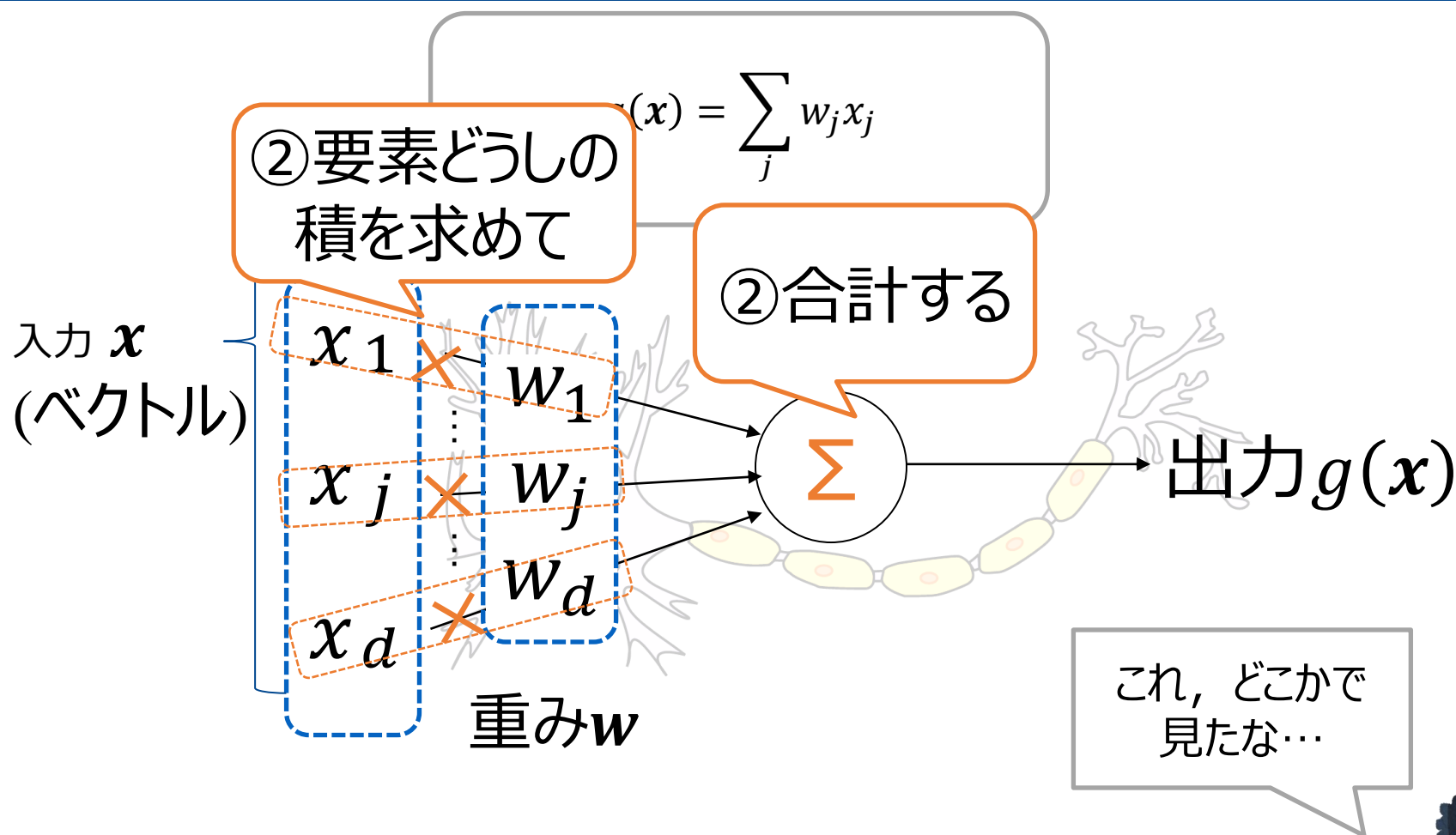


そしてその結果…

$$g(\mathbf{x}) = \sum_j w_j x_j$$



そしてその結果…



というわけで、ニューロン≡内積

$$g(x) = \sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x}$$

ベクトルの
かけ算だ

以前やったやつ

「内積」を使えば成分量が計れる！

$\mathbf{x} = \begin{pmatrix} 2 \\ 6 \end{pmatrix}$ の中に $\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ はどれくらい入っている？

$$\mathbf{x} \cdot \mathbf{e}_1 = \begin{pmatrix} 2 \\ 6 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2$$

内積で \mathbf{x} に含まれる \mathbf{e}_1 成分量が測れる！

思い出そう：内積

内積の書き方4種
(教科書で違ったりする)

$\mathbf{x} \cdot \mathbf{y}$
 (\mathbf{x}, \mathbf{y})
 $\langle \mathbf{x}, \mathbf{y} \rangle$
 $\mathbf{x}^T \mathbf{y}$

• ご存じのはず

$$\mathbf{x} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}, \mathbf{y} = \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{の内積} \rightarrow \mathbf{x} \cdot \mathbf{y} = 3 \times 6 + 5 \times 1 = 23$$

• 要するに、「要素どうしの積をとって、全部足す」

• その原理で、何次元ベクトルでも計算可能

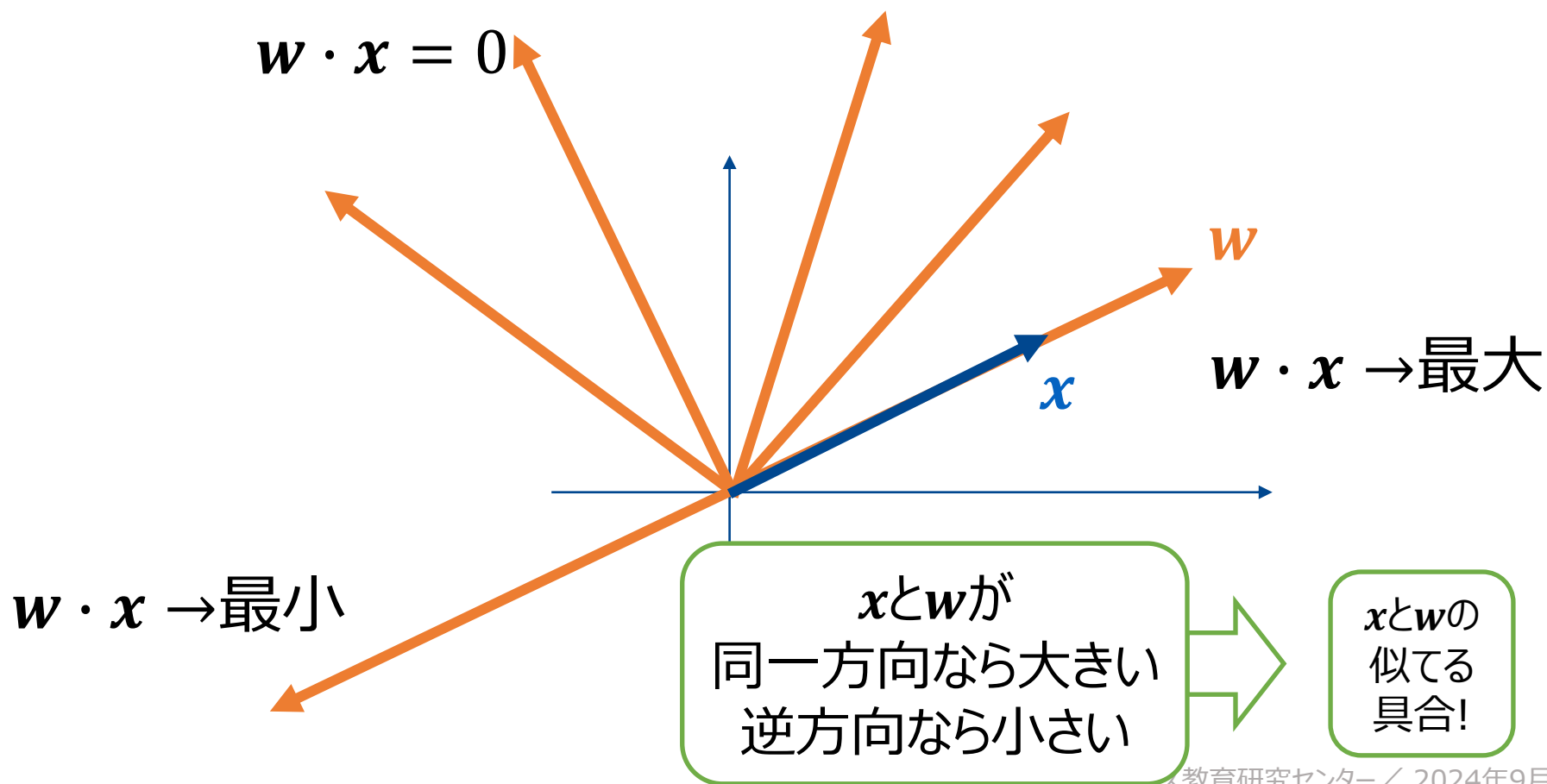
$$\begin{pmatrix} 3 \\ 5 \end{pmatrix} \text{と} \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{の内積} \Rightarrow \left\{ \begin{array}{l} \begin{pmatrix} 3 \\ 5 \end{pmatrix} \times \begin{pmatrix} 6 \\ 1 \end{pmatrix} = 18 \\ \phantom{\begin{pmatrix} 3 \\ 5 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \end{pmatrix}} = 5 \end{array} \right\} 18 + 5 = 23$$

$$\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix} \text{と} \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix} \text{の内積} \Rightarrow \left\{ \begin{array}{l} \begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix} \times \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix} = 18 \\ \phantom{\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix}} = 5 \\ \phantom{\begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}} \times \phantom{\begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix}} = 4 \end{array} \right\} 18 + 5 + 4 = 27$$

※この調子で、4次元でも、100万次元でも可能

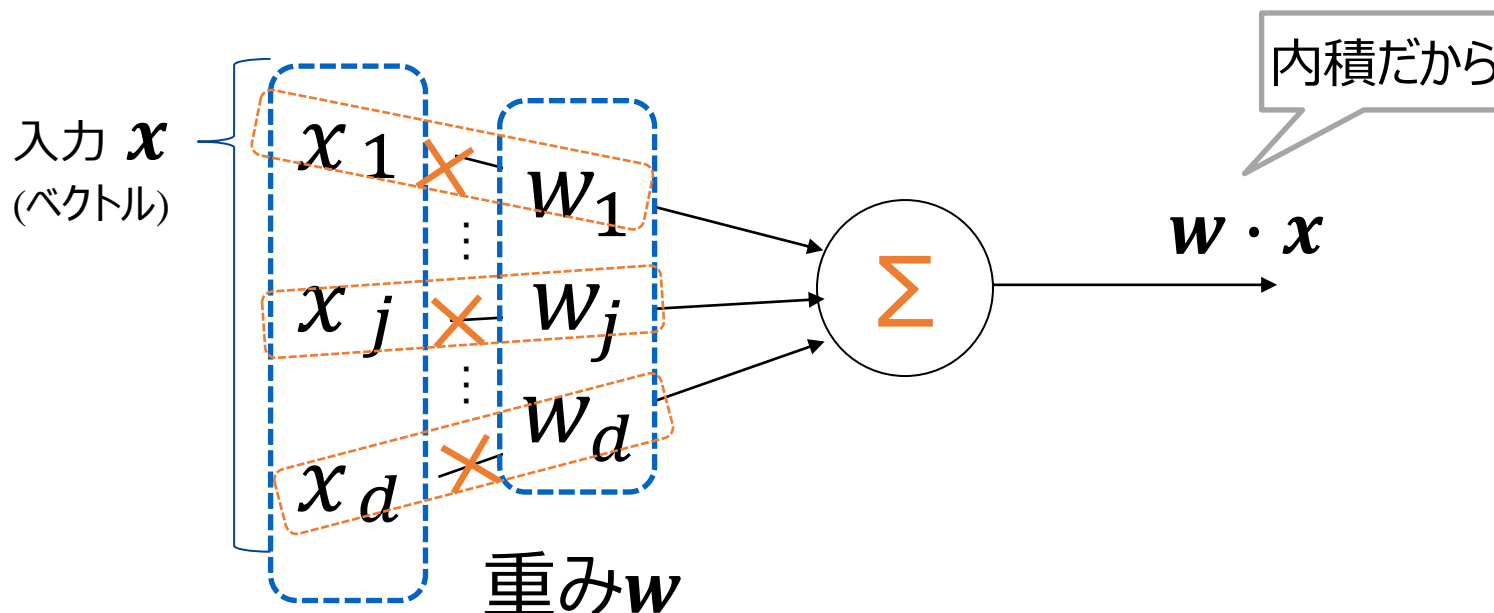
内積の復習

- ベクトル w を回転させながら x と内積をとると...



というわけで、ニューロンは 「自分好みの」入力に反応する

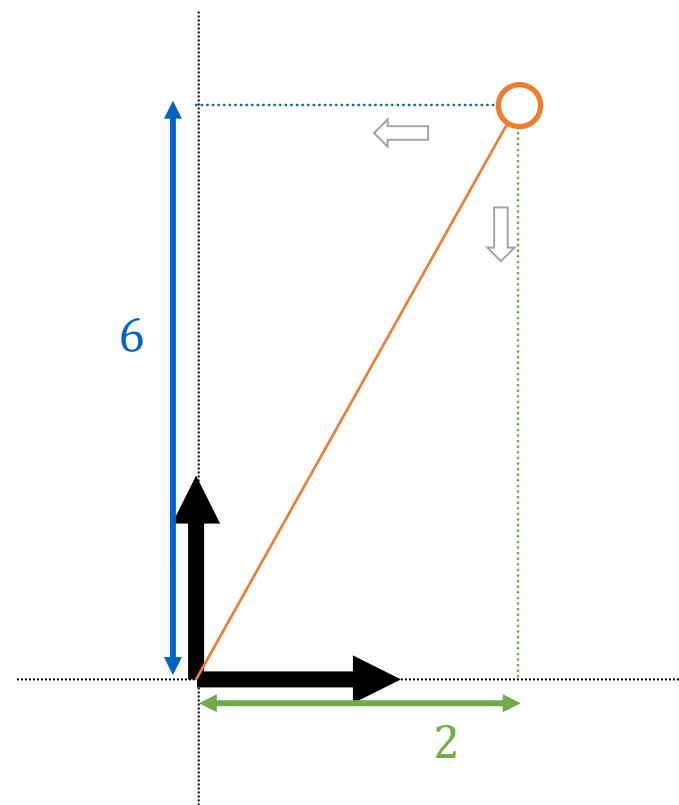
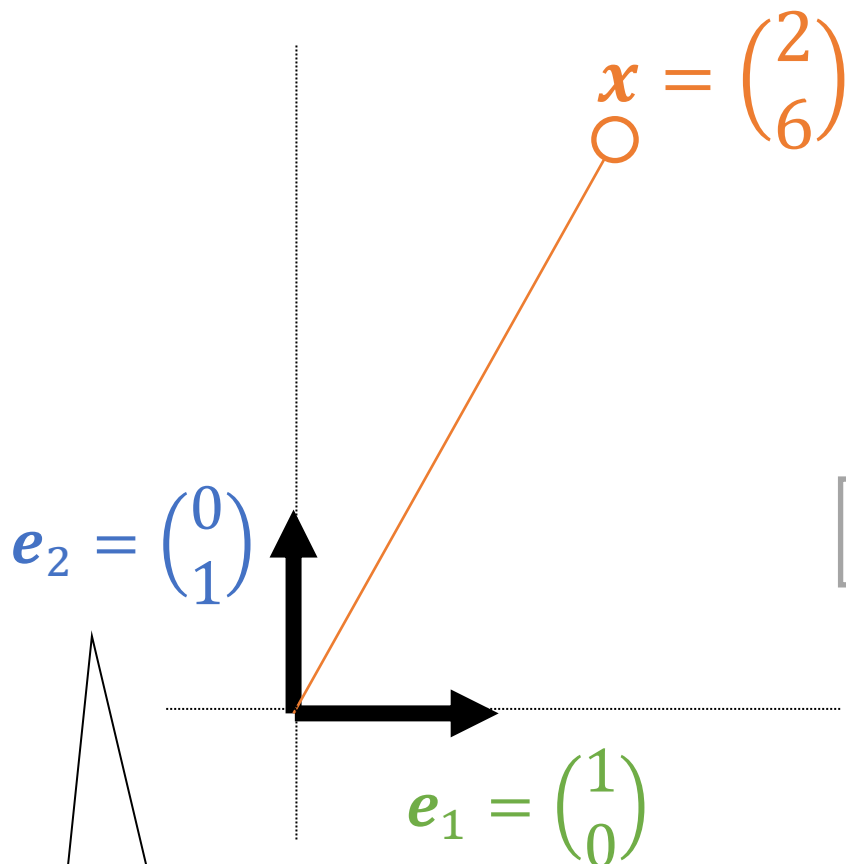
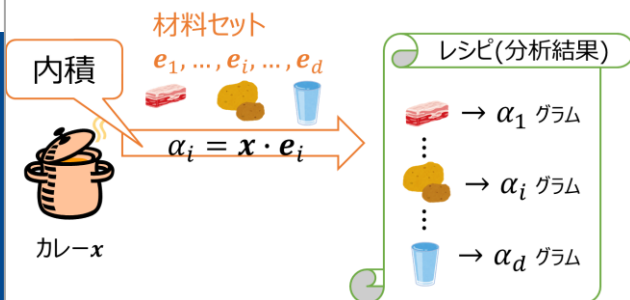
- ニューロン自身が持っている重み w と似たベクトル x が入力されると、大きく反応



複数のニューロンを用いることで データに別の見方を与える

深層ニューラルネットワークによる
パターン認識②

もう一度，内積の復習

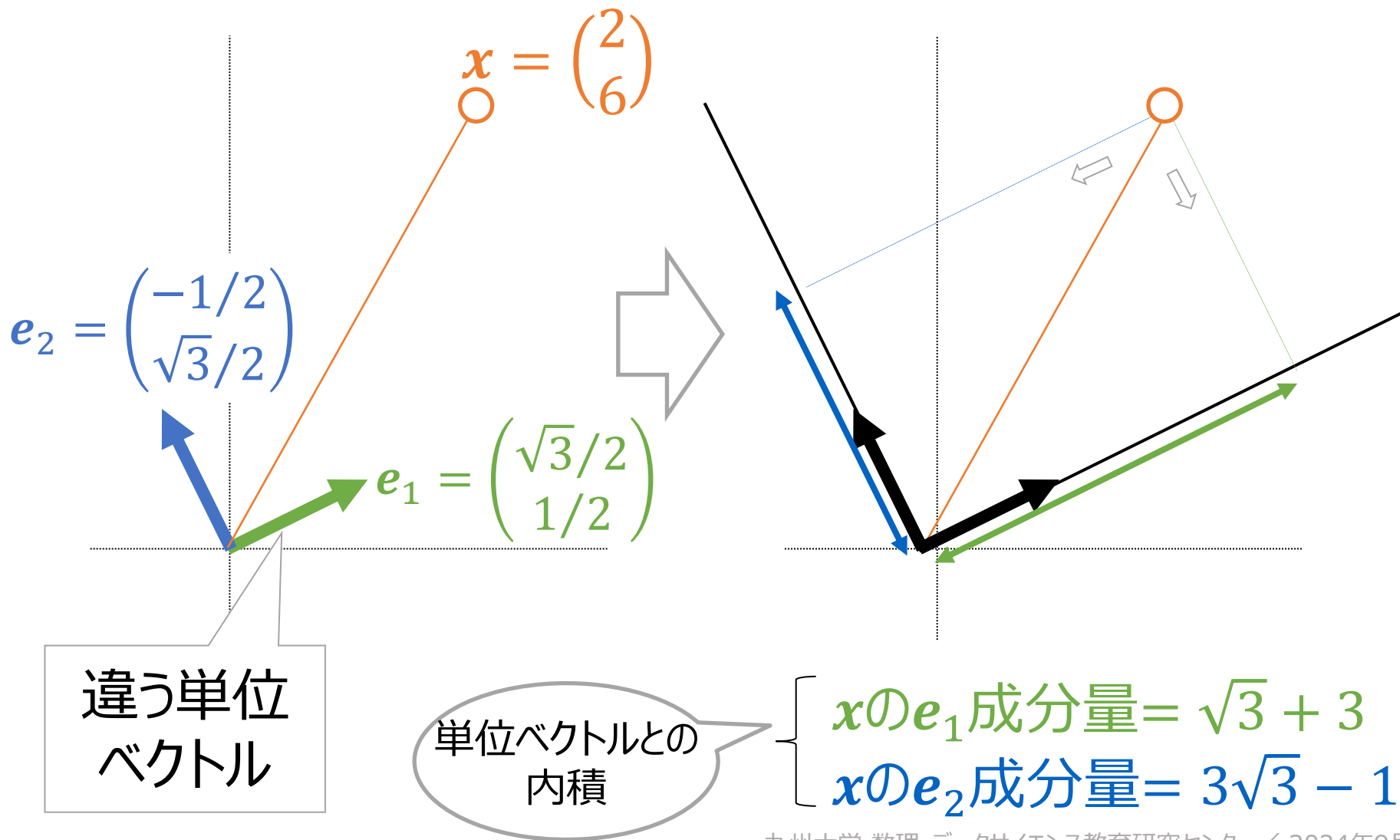
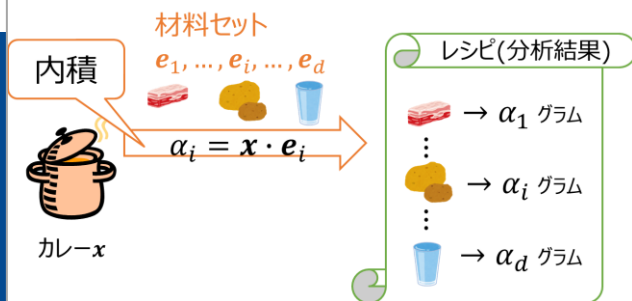


単位
ベクトル

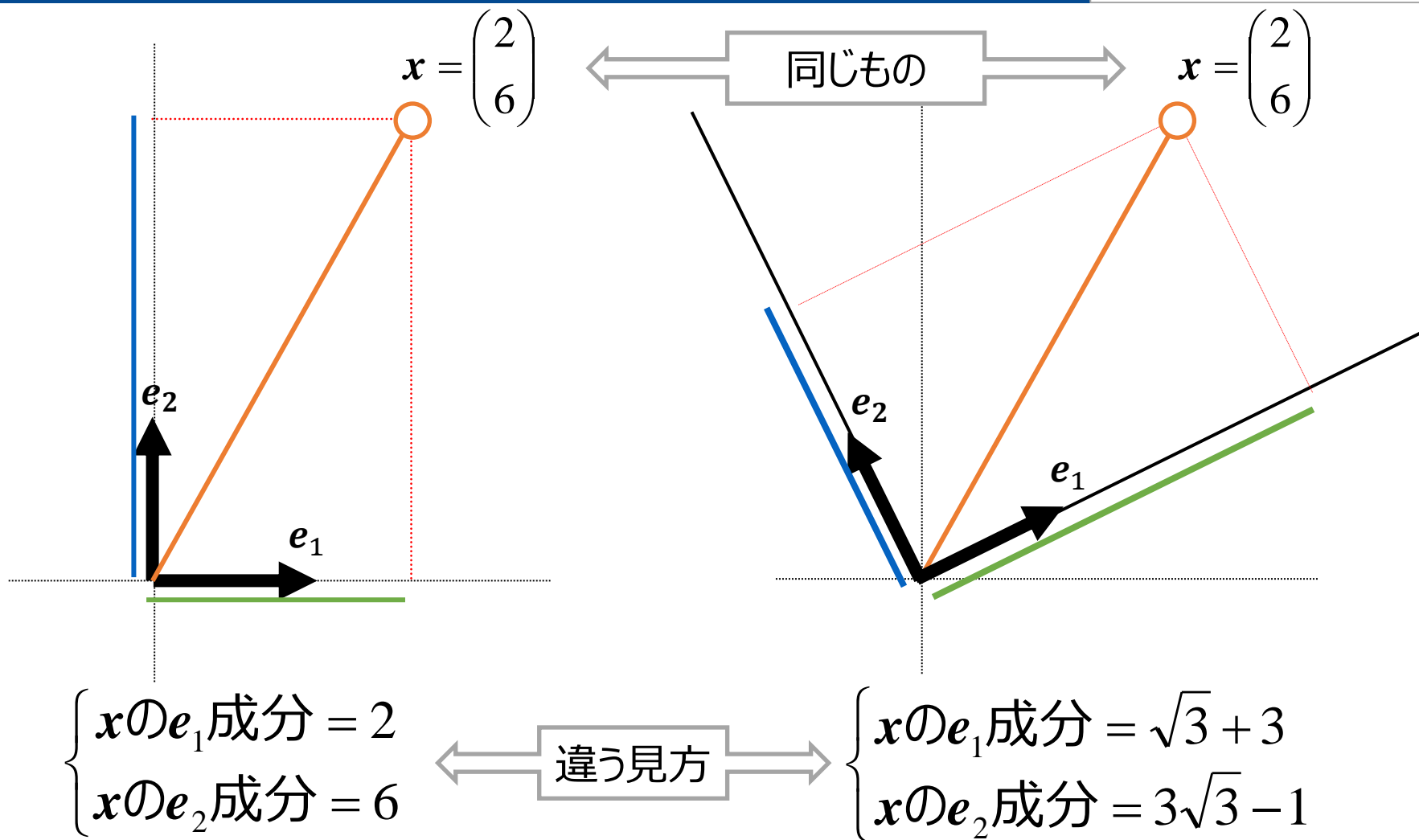
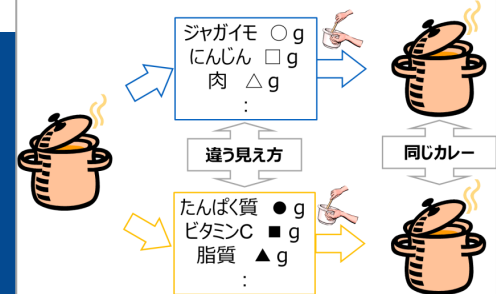
単位ベクトルとの
内積

x の e_1 成分量 = 2
 x の e_2 成分量 = 6

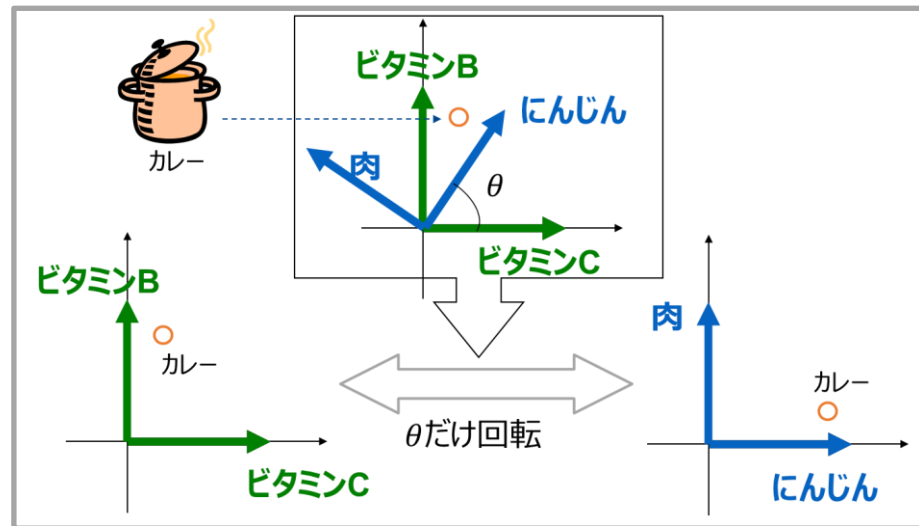
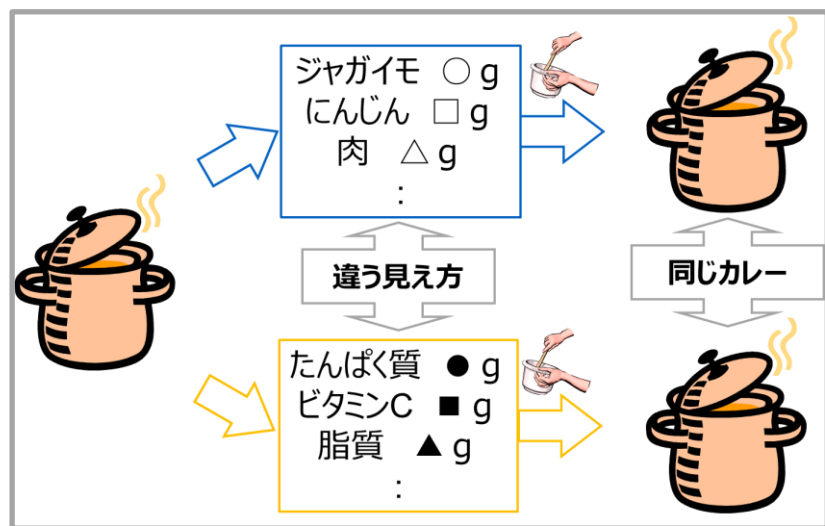
もう一度，内積の復習



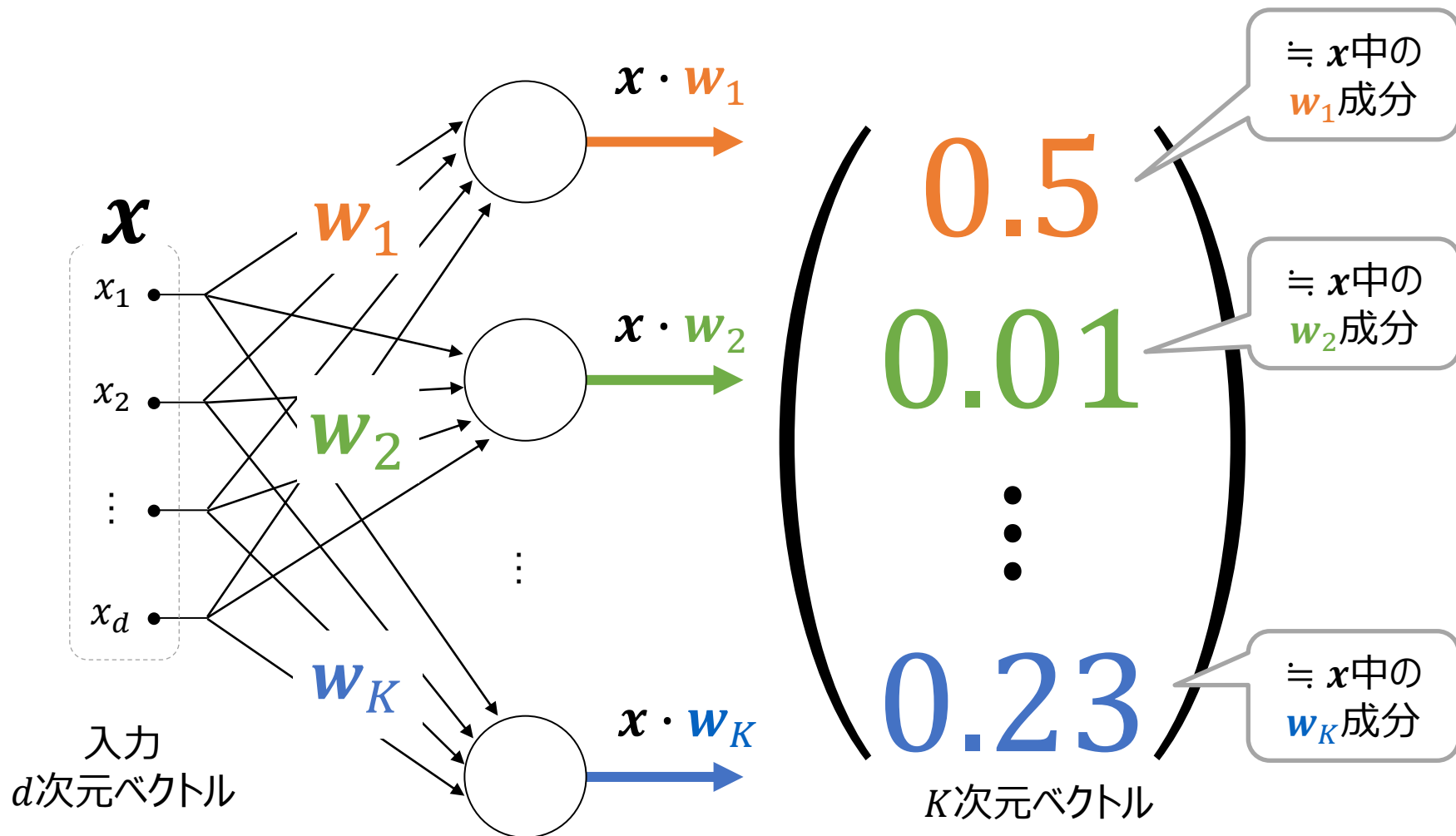
もう一度，内積の復習



こんな話もありました： 「同じものが違って見える」

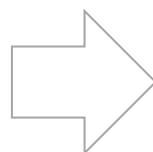
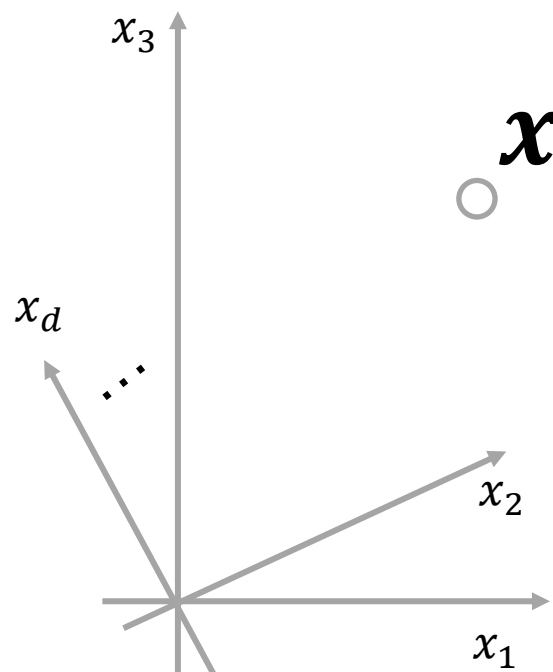


ところで、ニューロンを K 個並べると…

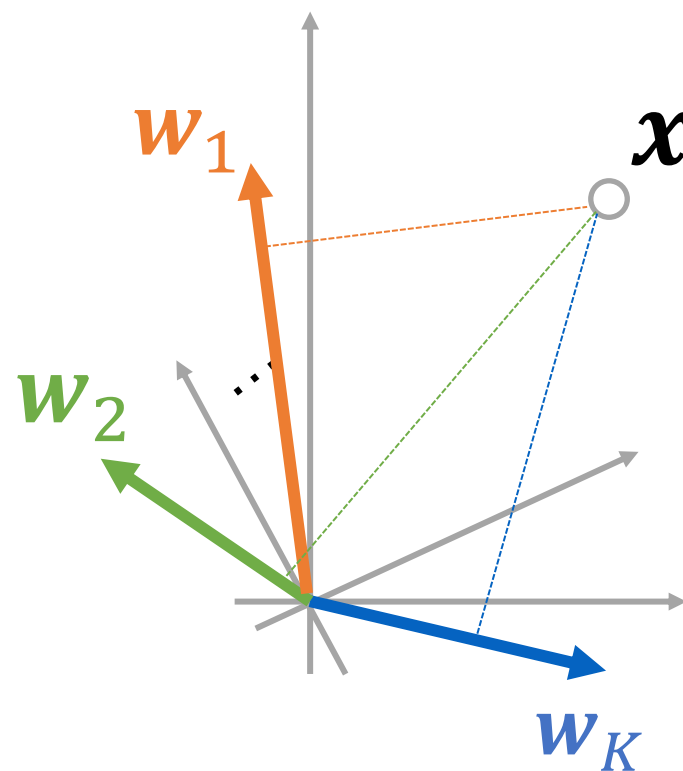


ということ？ (1/2)

元々の状態



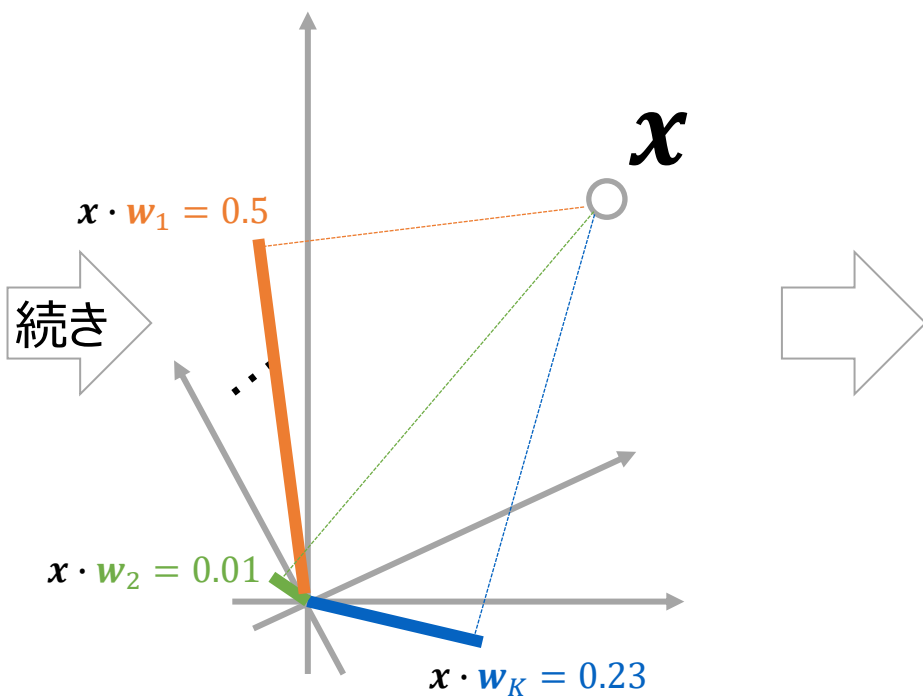
新しい「 K 本の座標軸」
(新しい x の見方)



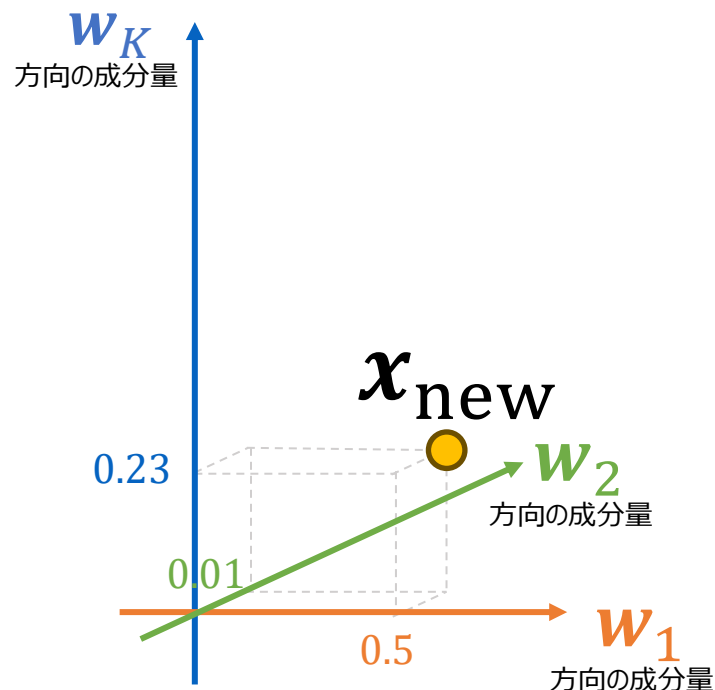
続く

ということ？ (2/2)

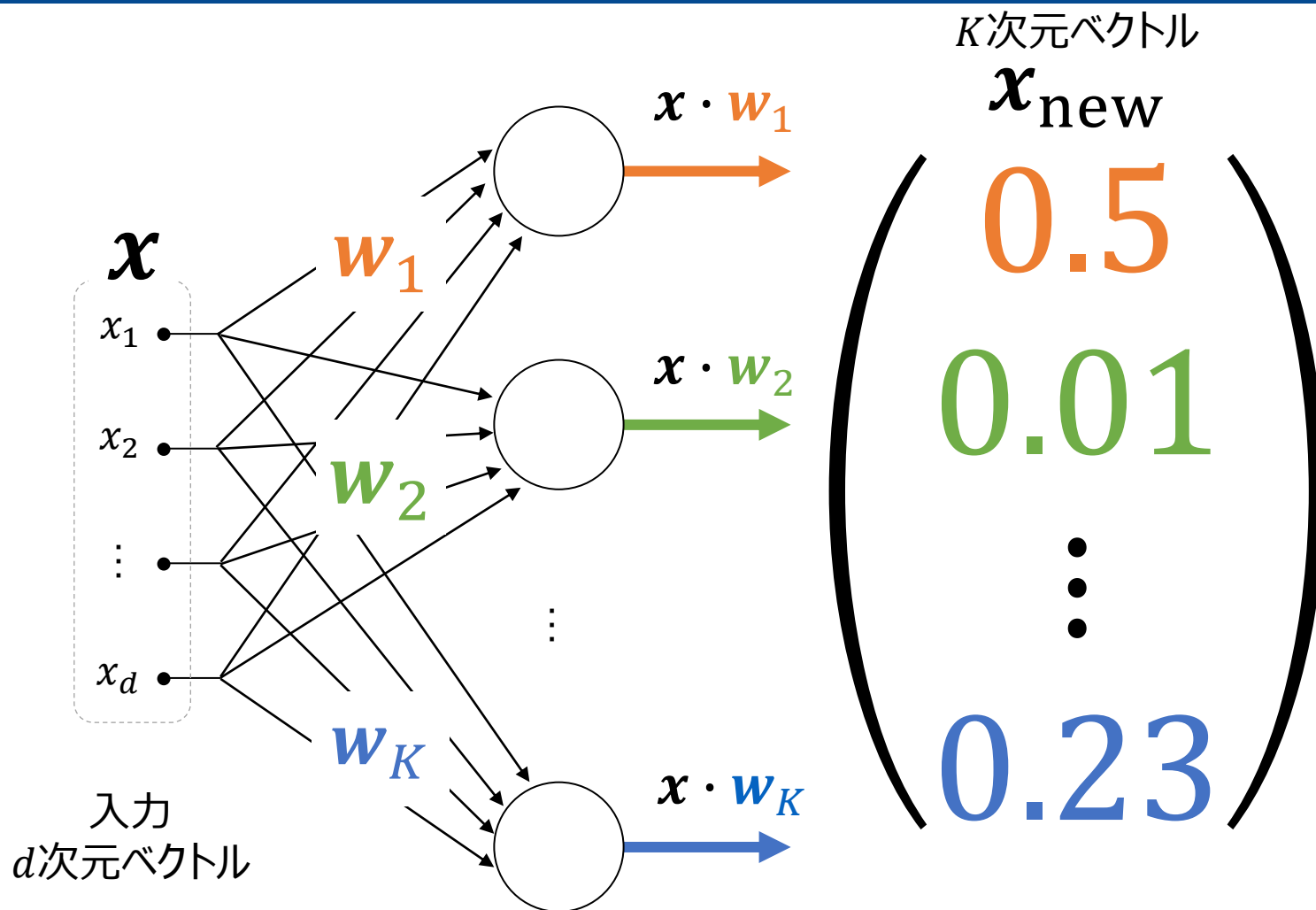
新しい「 K 本の座標軸」との内積



新しい x の見える方



なので、ニューロンを K 個並べると、 x の新しい見え方 x_{new} が得られる

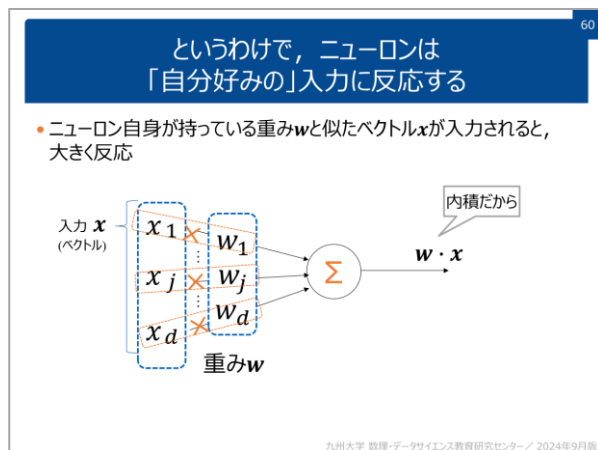


ニューラルネットワーク

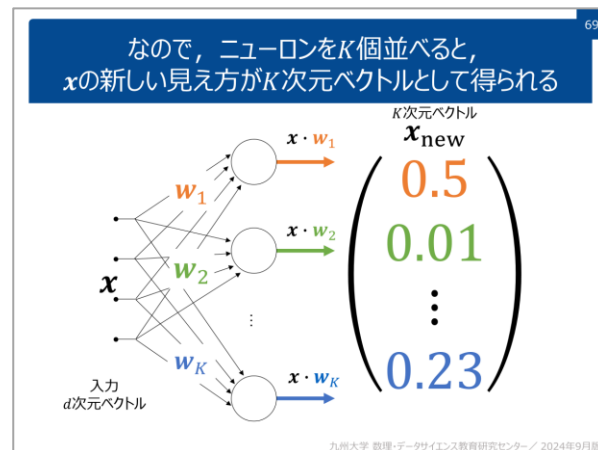
深層ニューラルネットワークによる
パターン認識③

ここまで出てきた「内積」の活躍ぶりを振り返る

● 内積はニューロンだ

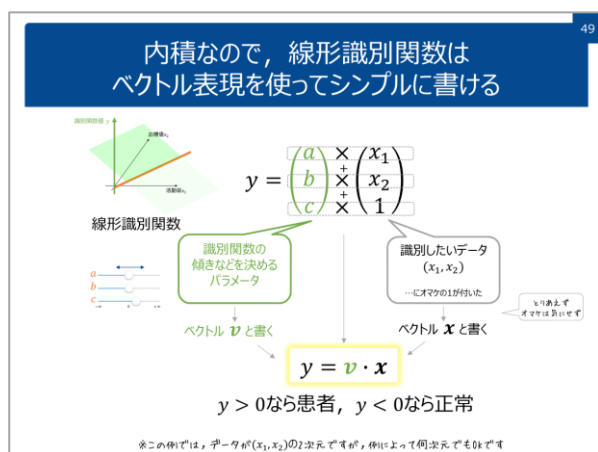


● 内積は「新しい表現を与える」!



K 個のニューロンを使うと...

● 内積は線形識別関数だ!



単なる掛け算と
足し算なのに...



思い出そう：内積

● 保存するのは

$$x = \begin{pmatrix} 3 \\ 5 \end{pmatrix}, y = \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{ の内積 } \rightarrow x \cdot y = 3 \times 6 + 5 \times 1 = 23$$

● 要するに、「要素どうしの積をとって、全部足す」

● その原理で、何次元ベクトルでも計算可能

$$\begin{pmatrix} 3 \\ 5 \end{pmatrix} \text{ と } \begin{pmatrix} 6 \\ 1 \end{pmatrix} \text{ の内積 } \Rightarrow \begin{pmatrix} 3 & 6 \\ 5 & 1 \end{pmatrix} \begin{matrix} = 18 \\ = 5 \end{matrix} \rightarrow 18 + 5 = 23$$

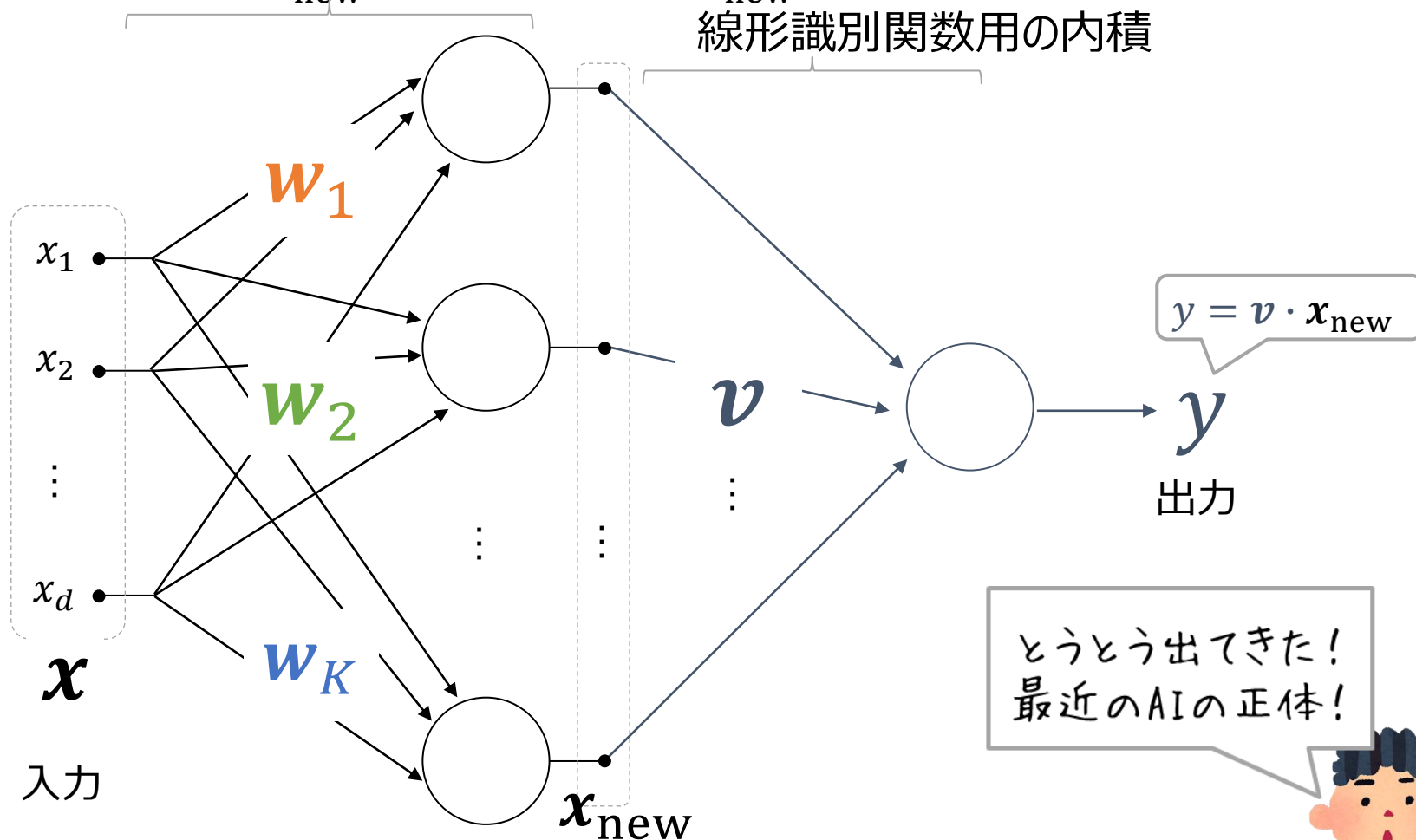
$$\begin{pmatrix} 3 \\ 5 \end{pmatrix} \text{ と } \begin{pmatrix} 6 \\ 2 \end{pmatrix} \text{ の内積 } \Rightarrow \begin{pmatrix} 3 & 6 \\ 5 & 2 \end{pmatrix} \begin{matrix} = 18 \\ = 10 \end{matrix} \rightarrow 18 + 5 + 4 = 27$$

※この調子で、4次元でも、100次元でも可能

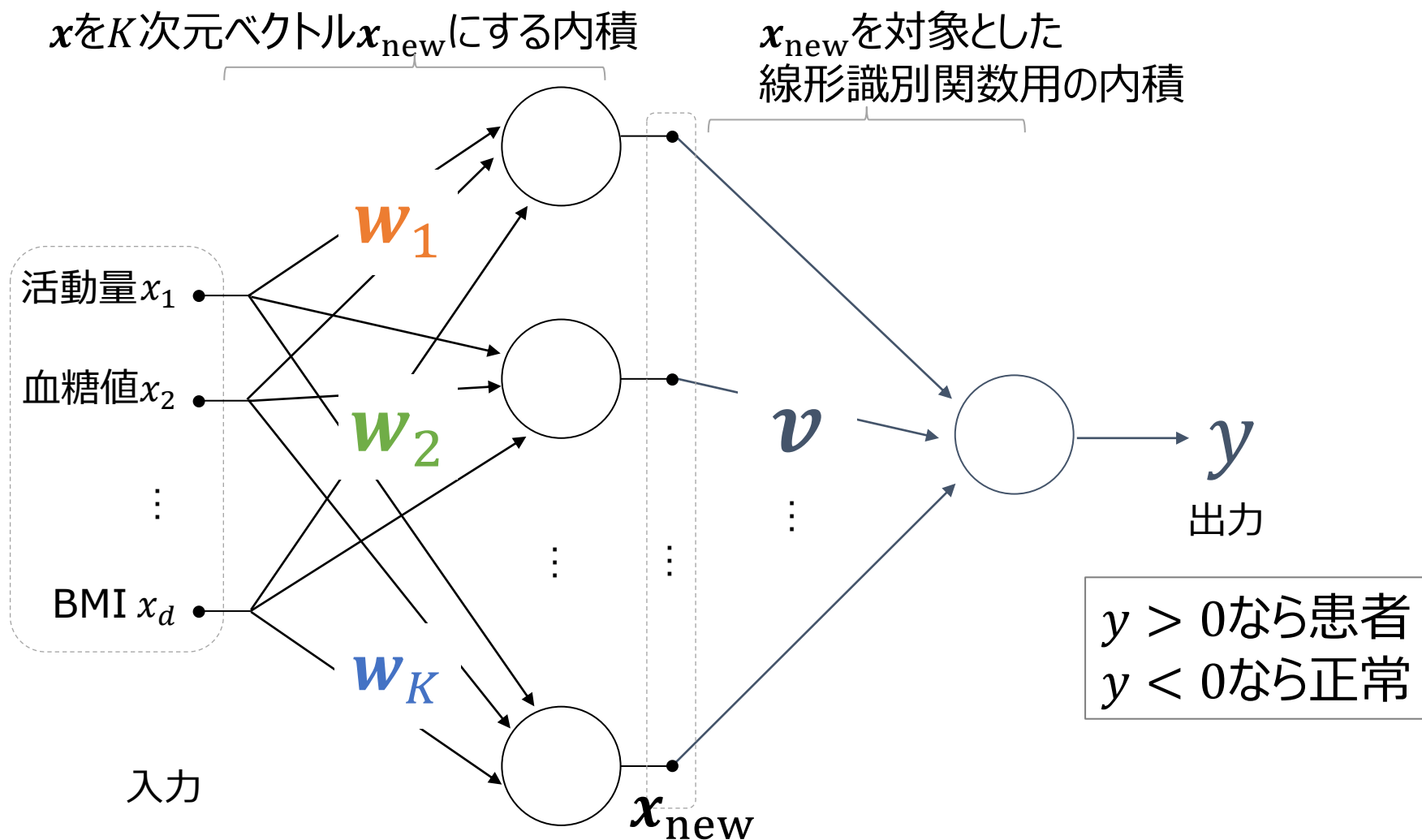
ニューロン(=内積)を以下のように重ねたものが ニューラルネットワーク (の基本形)

x を K 次元ベクトル x_{new} にする内積

x_{new} を対象とした
線形識別関数用の内積



例：健康診断なら

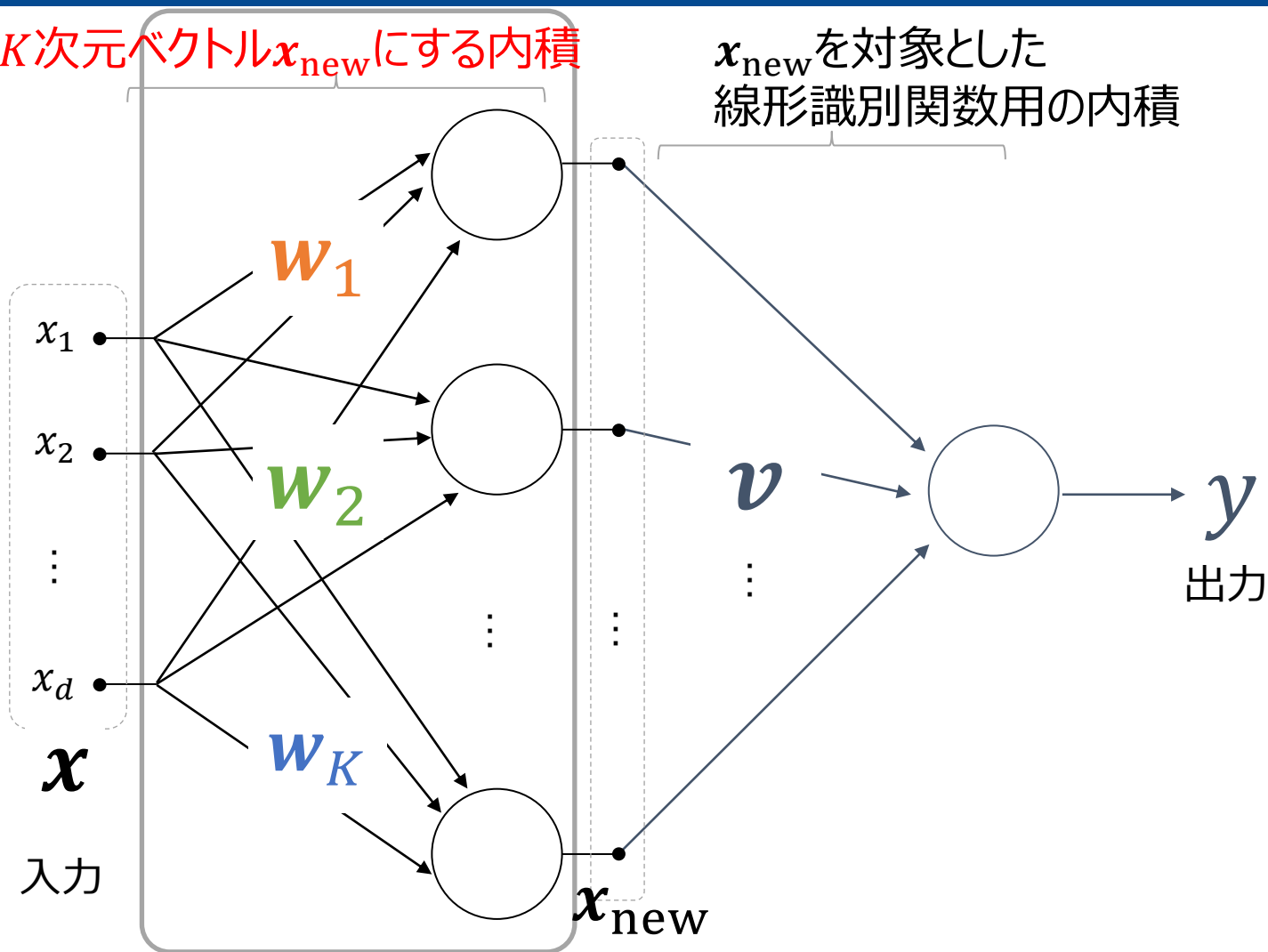


ん？ ということ？

特に線形識別の前に x を x_{new} にするのはなぜ？

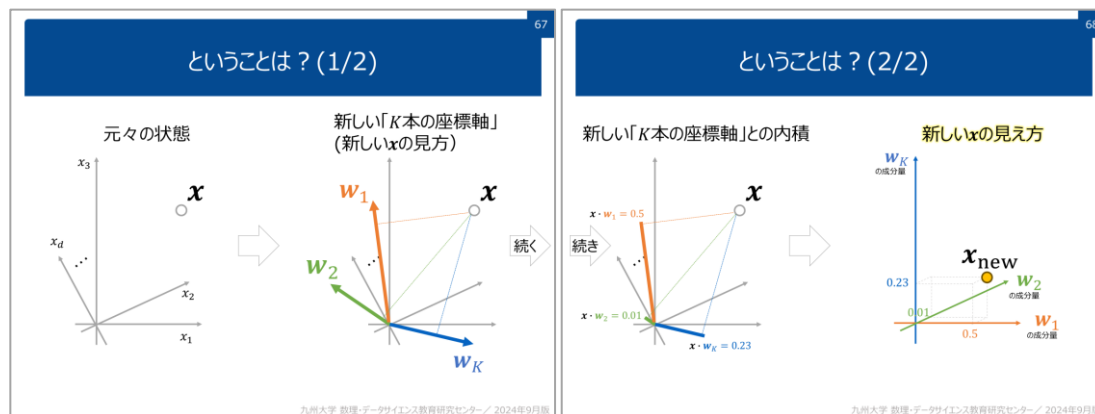
x を K 次元ベクトル x_{new} にする内積

x_{new} を対象とした
線形識別関数用の内積

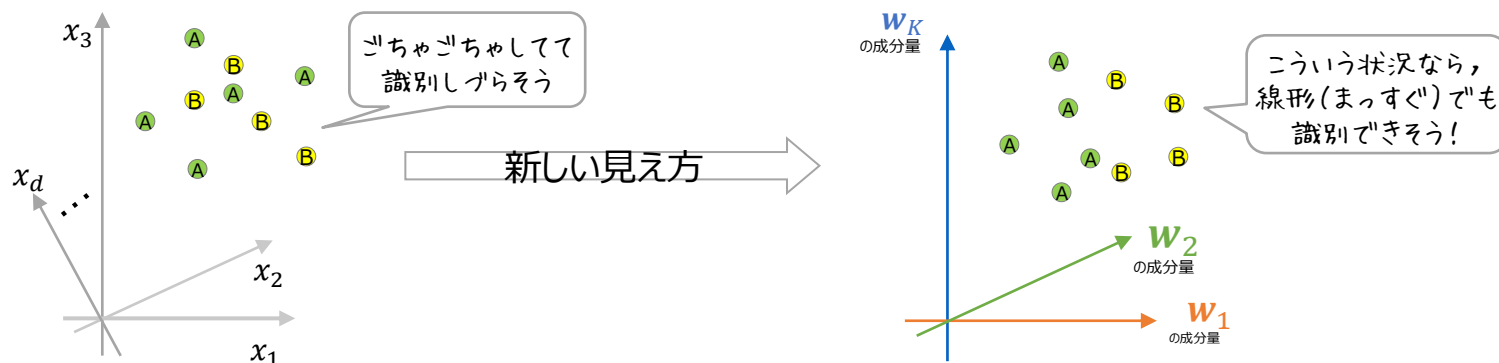


実は「 x を x_{new} にする」のはものすごく大事(1/2)

- x を x_{new} にするのはこんな感じでしたよね

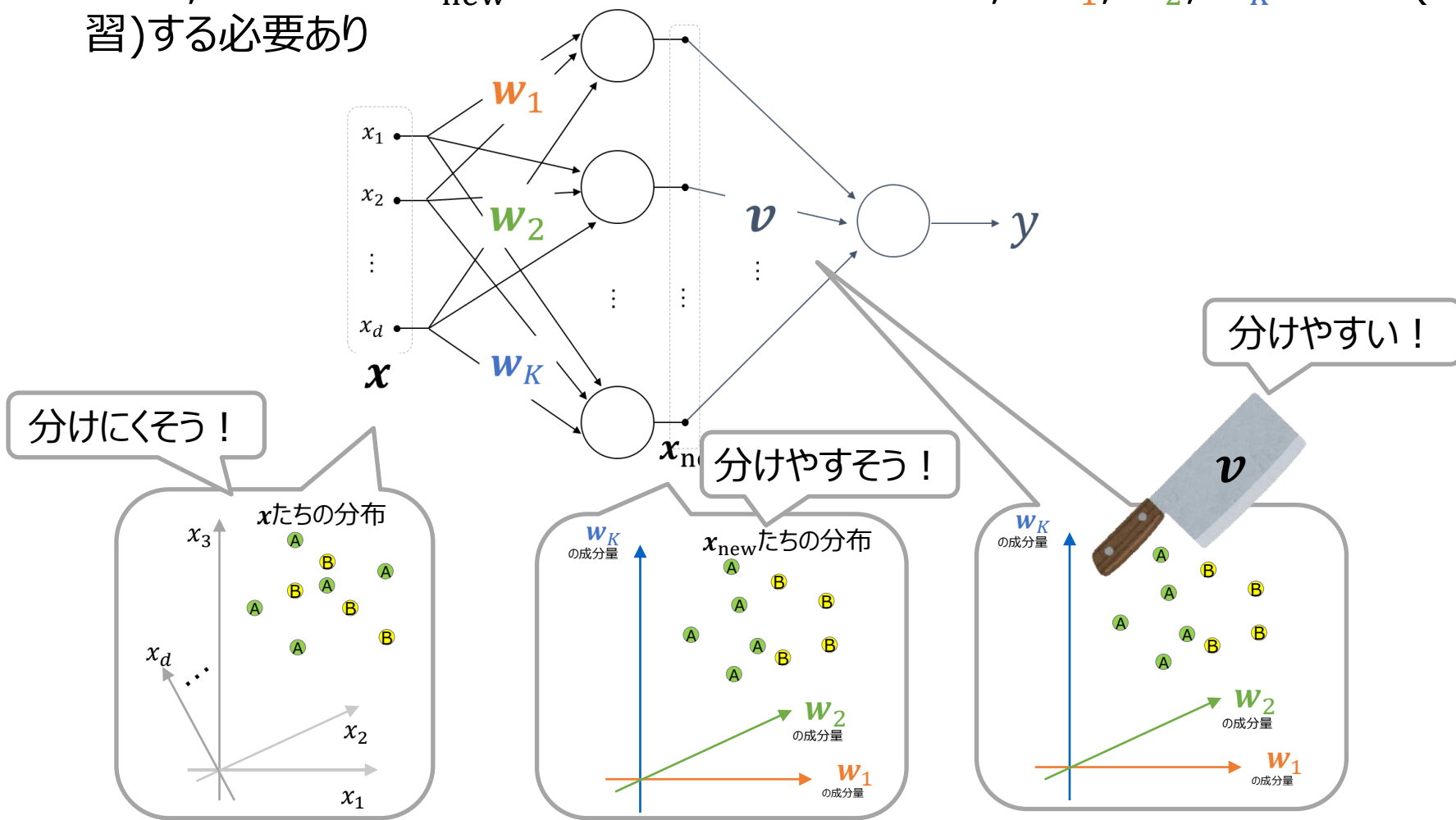


- もし識別したいデータが、より識別しやすくなるような「新しい見え方」があれば…

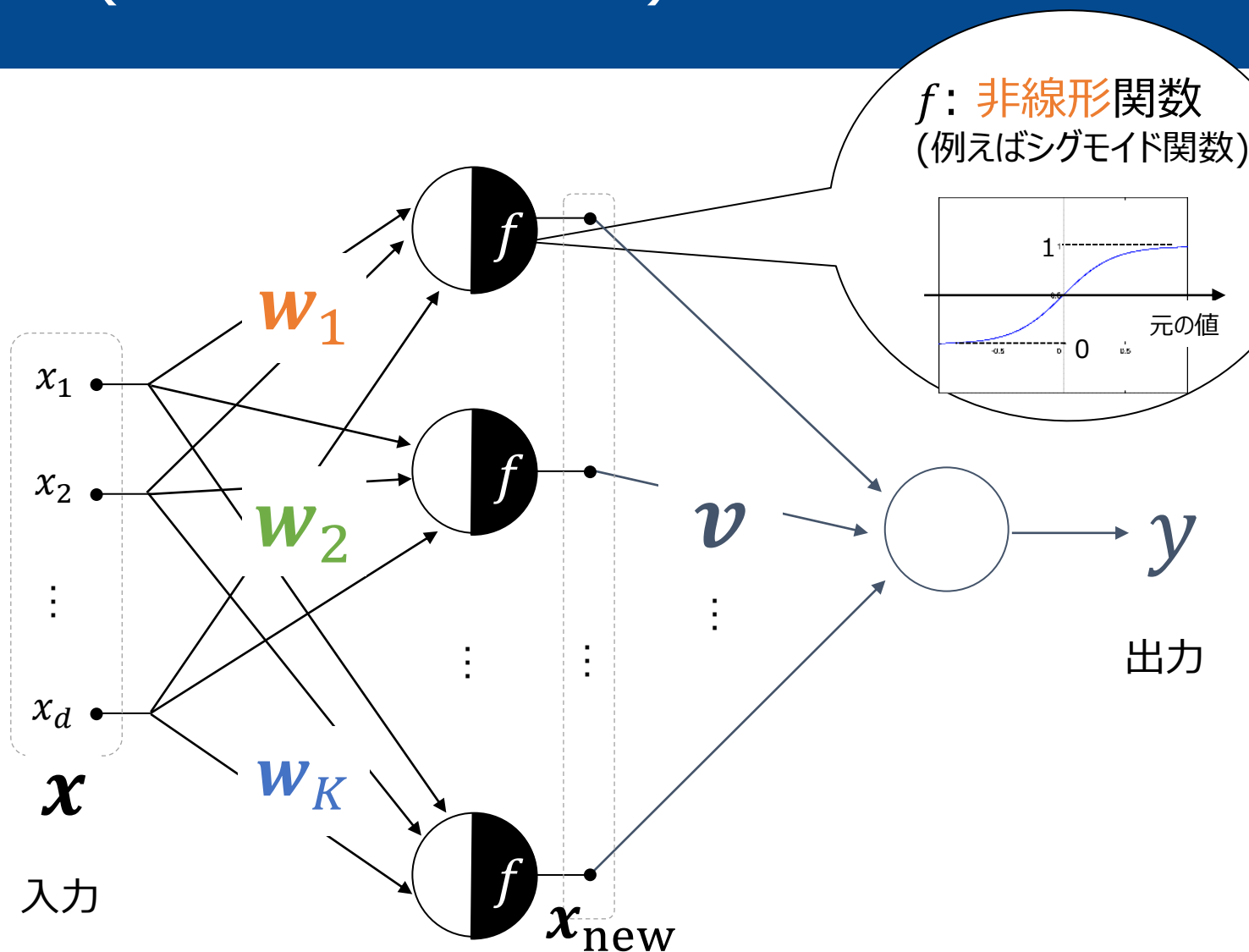


実は「 x を x_{new} にする」のはものすごく大事(2/2)

- ただし, x のままより x_{new} がより識別しやすくなるように, w_1, w_2, w_K を設定(学習)する必要あり

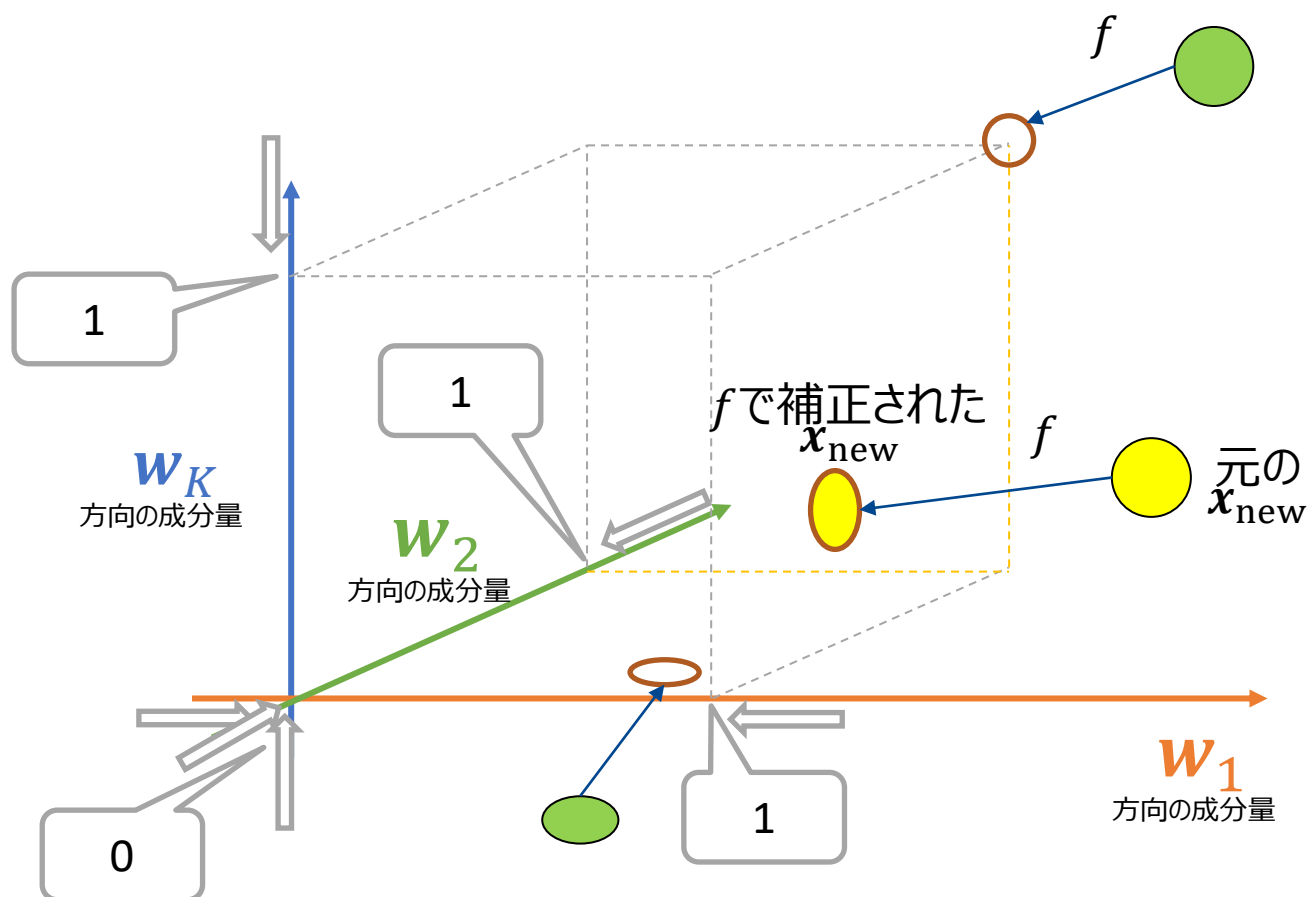
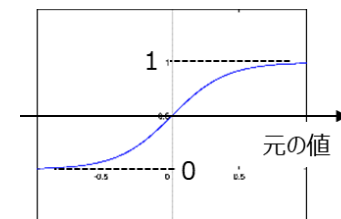


(とりあえず無視した)非線形関数は？



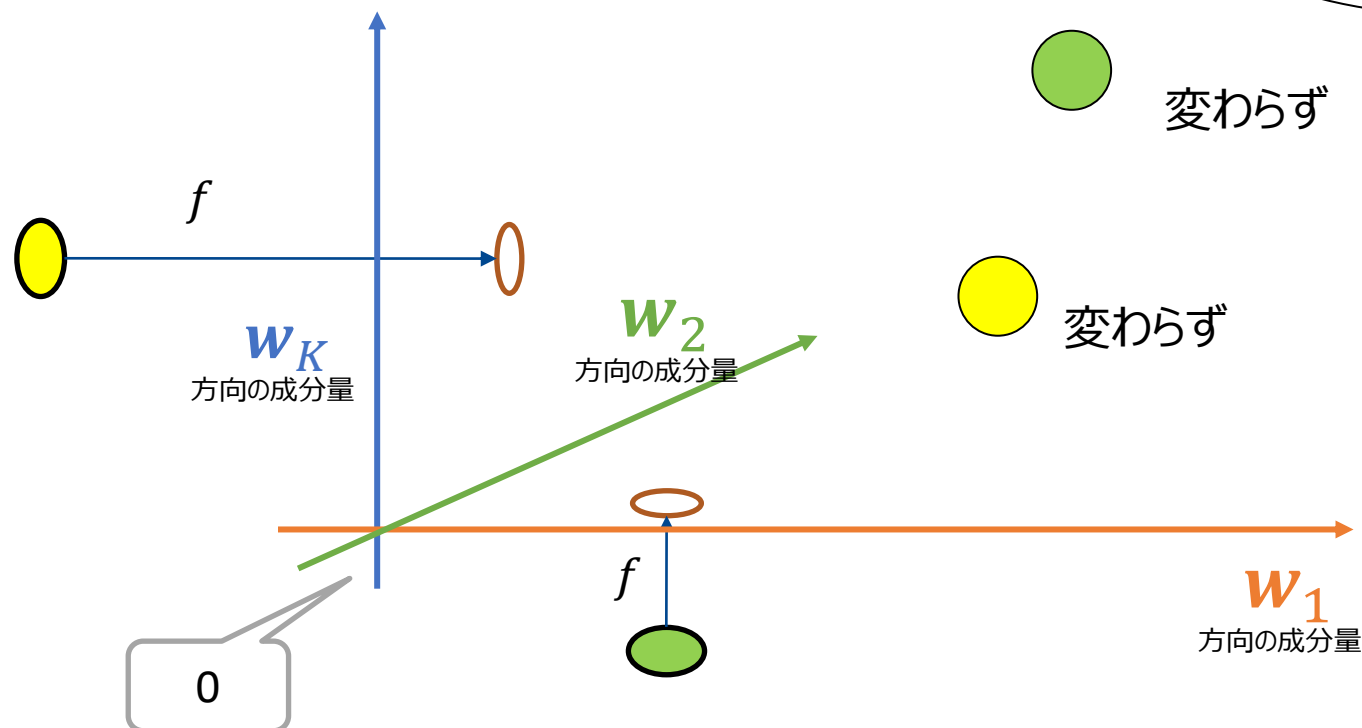
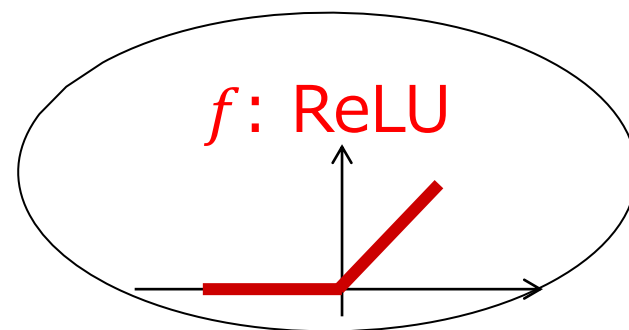
非線形関数 f の効果の直感的理解(1/2)

- シグモイド関数→ x_{new} が $[0,1]$ 区間に入るように補正



非線形関数の効果の直感的理解(2/2)

- ReLU $\rightarrow x_{\text{new}}$ が $[0, \infty]$ 区間に入るように
 - 要は「 x_{new} にマイナス成分があるとそれをゼロに」



参考：なぜ f なんて必要なの？

- 実は、 w との単純な内積だけで x を x_{new} にしても、分布の状況を大きくは変えられない

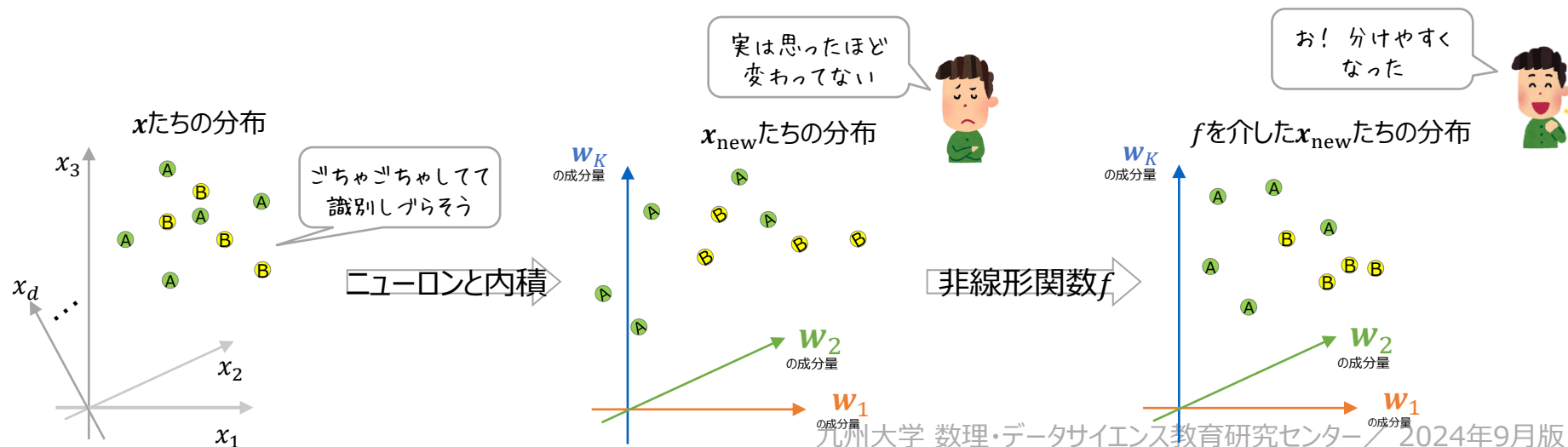


- 専門的には：「内積だけだと線形変換にしかない」

- f による追加処理が入ると、内積だけでは実現できないような、より望ましい x_{new} の分布に変化できる

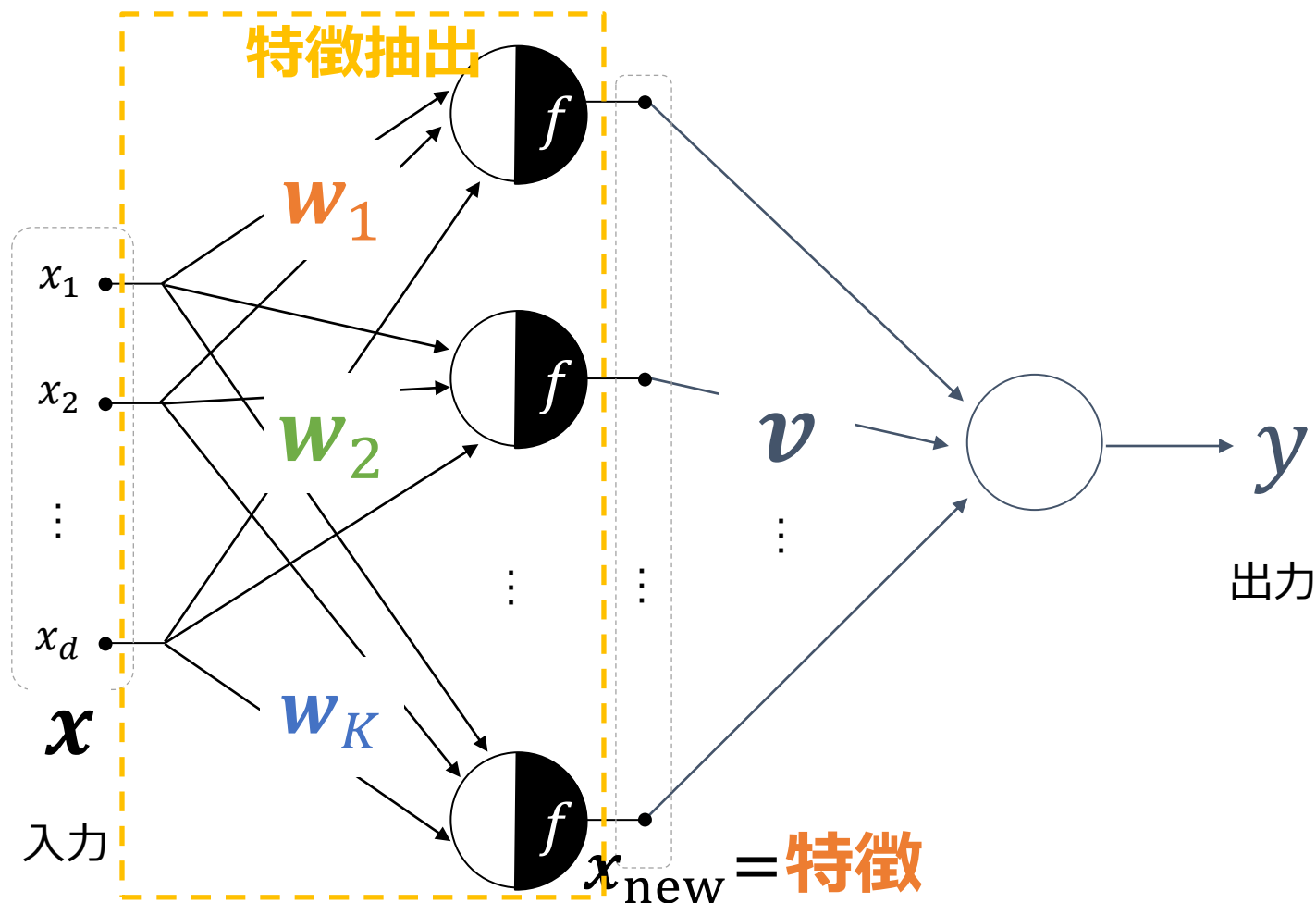


- 専門的には：「 f により非線形変換が実現」

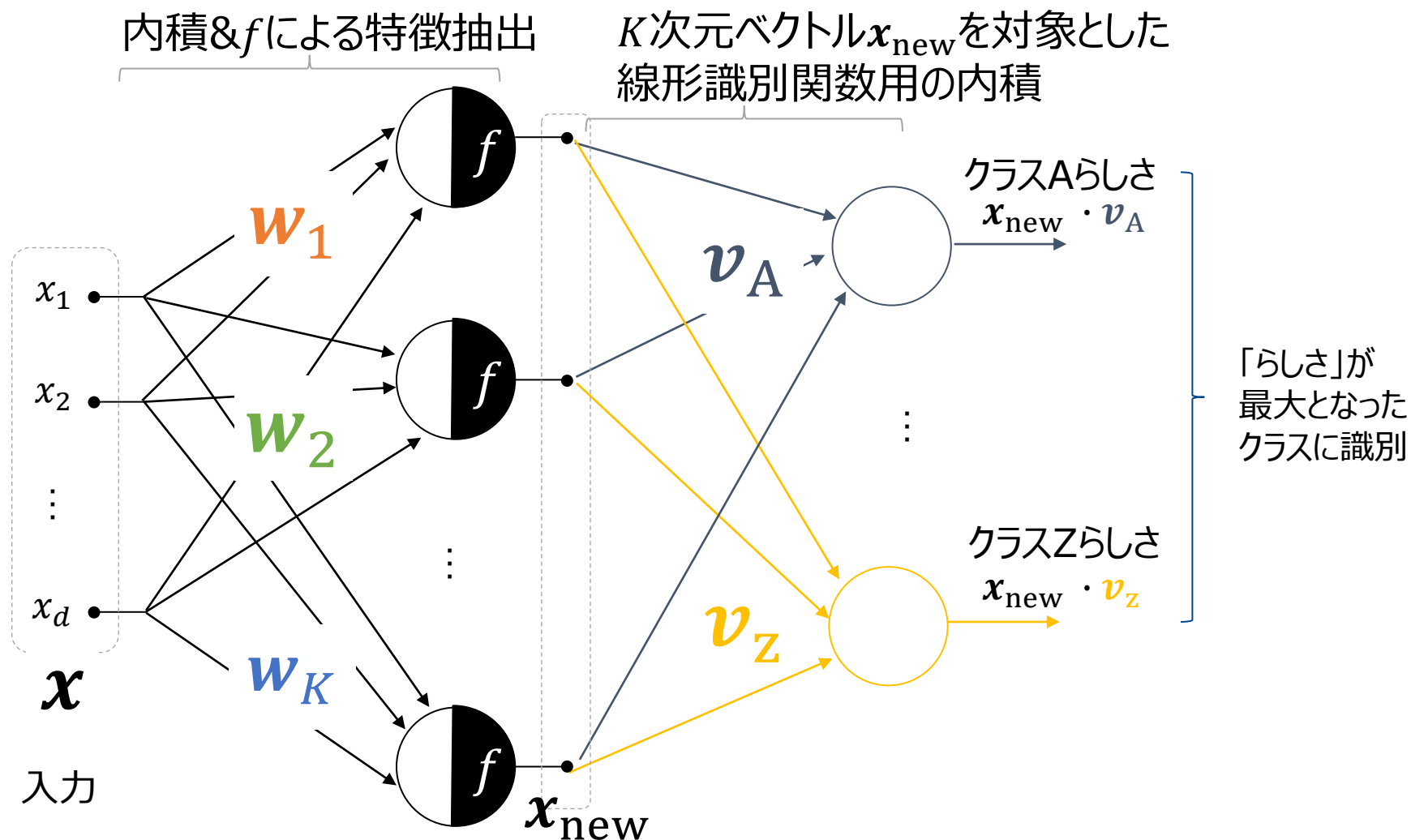


「 x を x_{new} にする」= 特徴抽出, と呼びます

- 元の x から, より識別に適した**特徴**として x_{new} を抽出するから



発展：識別関数を 多数のクラス分準備することもできます！



深層ニューラルネットワーク



深層ニューラルネットワークによる
パターン認識③

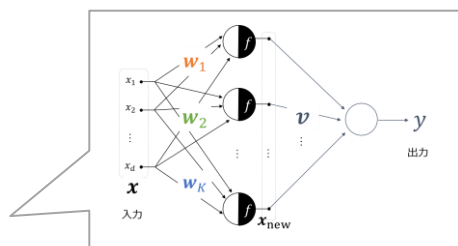


ニューラルネットワークから 深層ニューラルネットワークへ

1970
ごろ



1990
ごろ



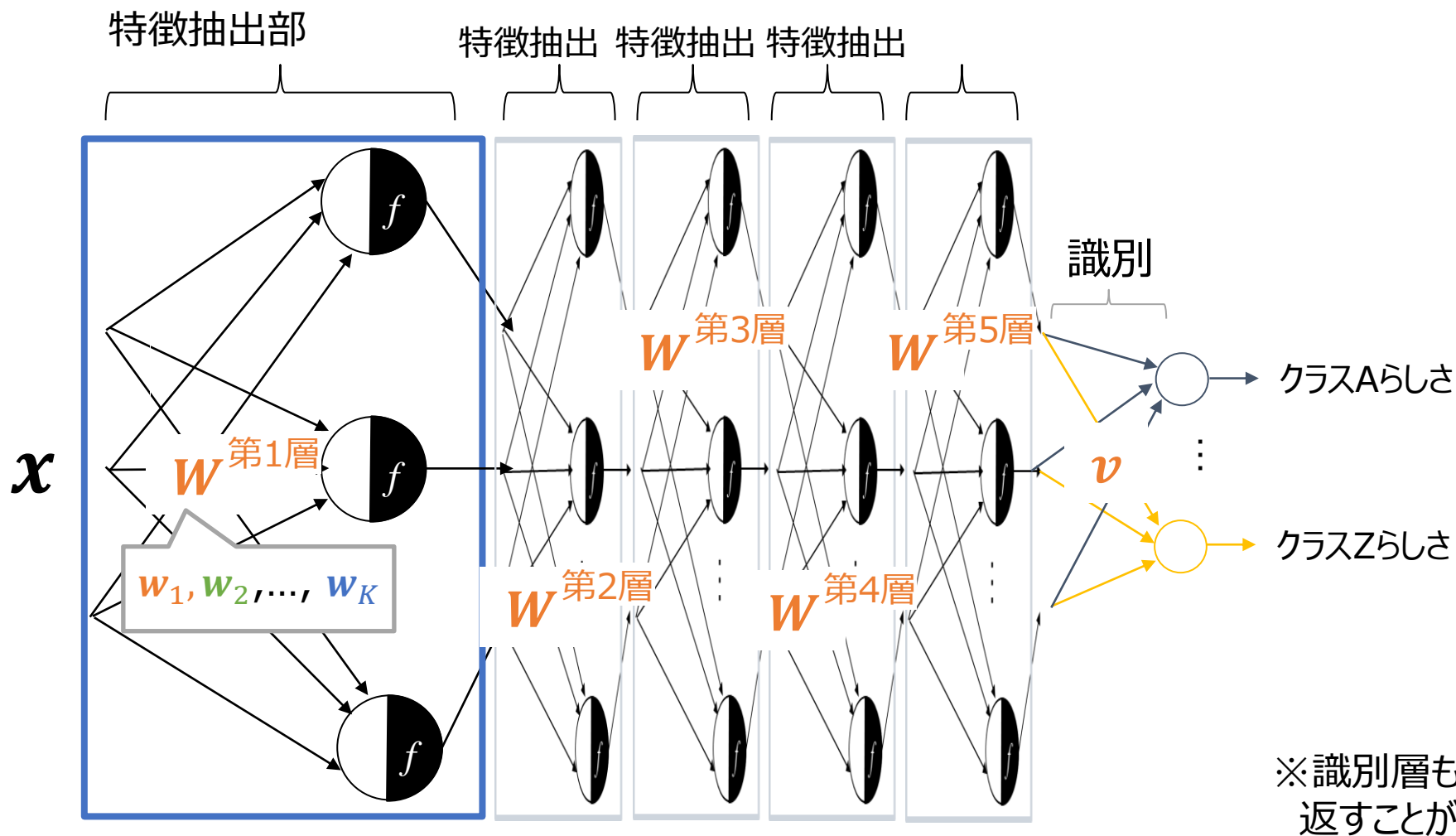
2015
ごろ



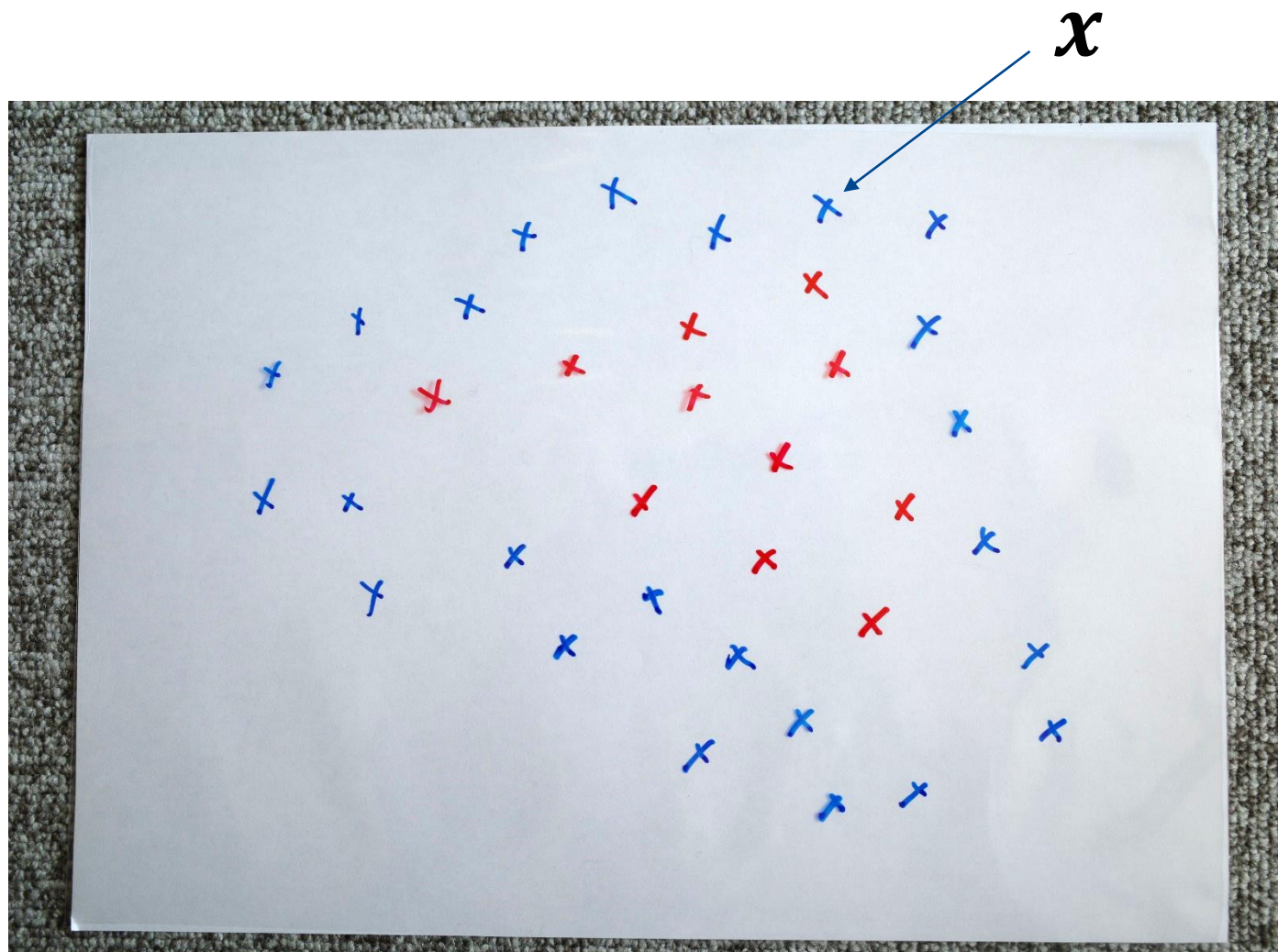
その後
...



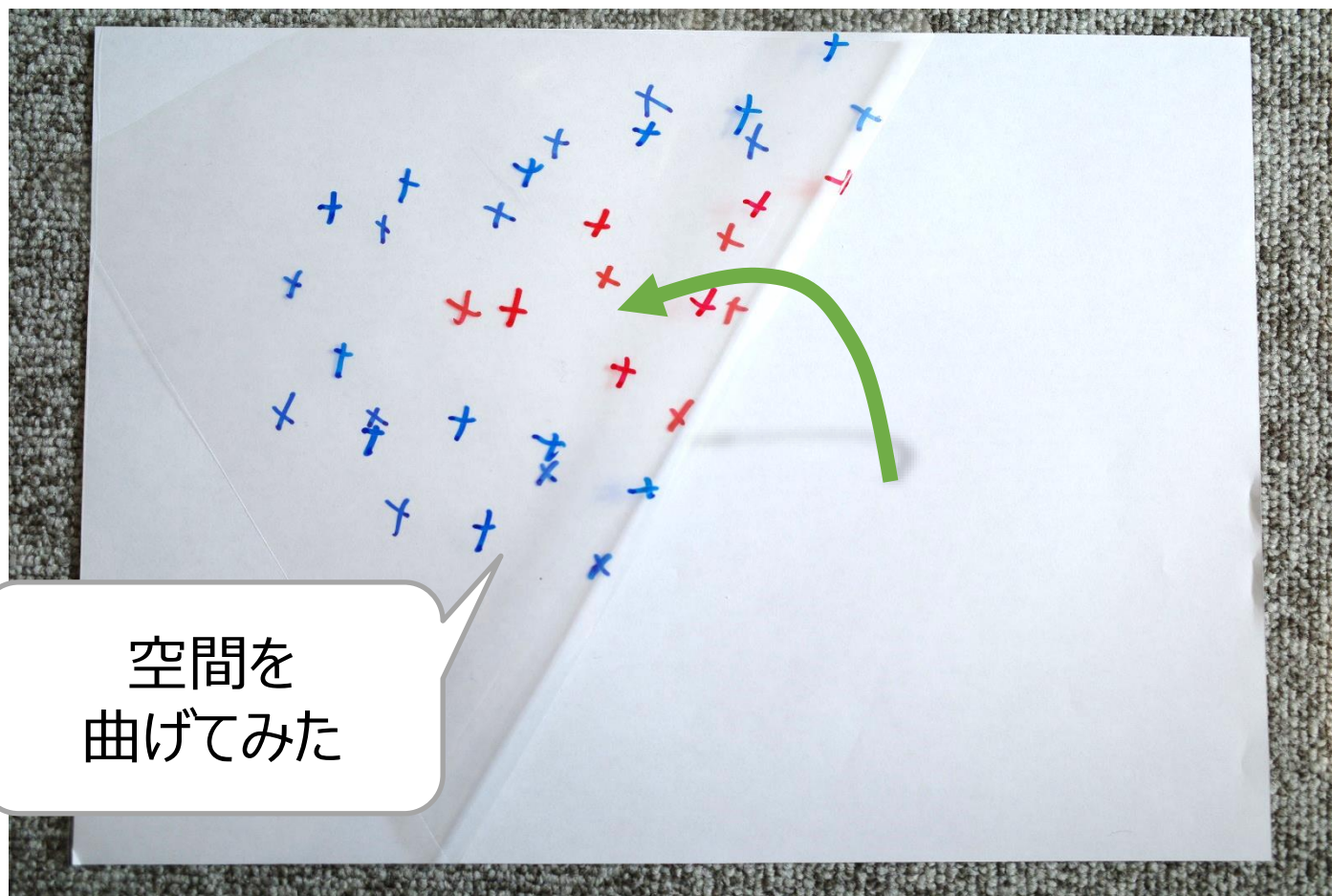
深層ニューラルネットワークの構造： 特徴抽出を繰り返すところがポイント！



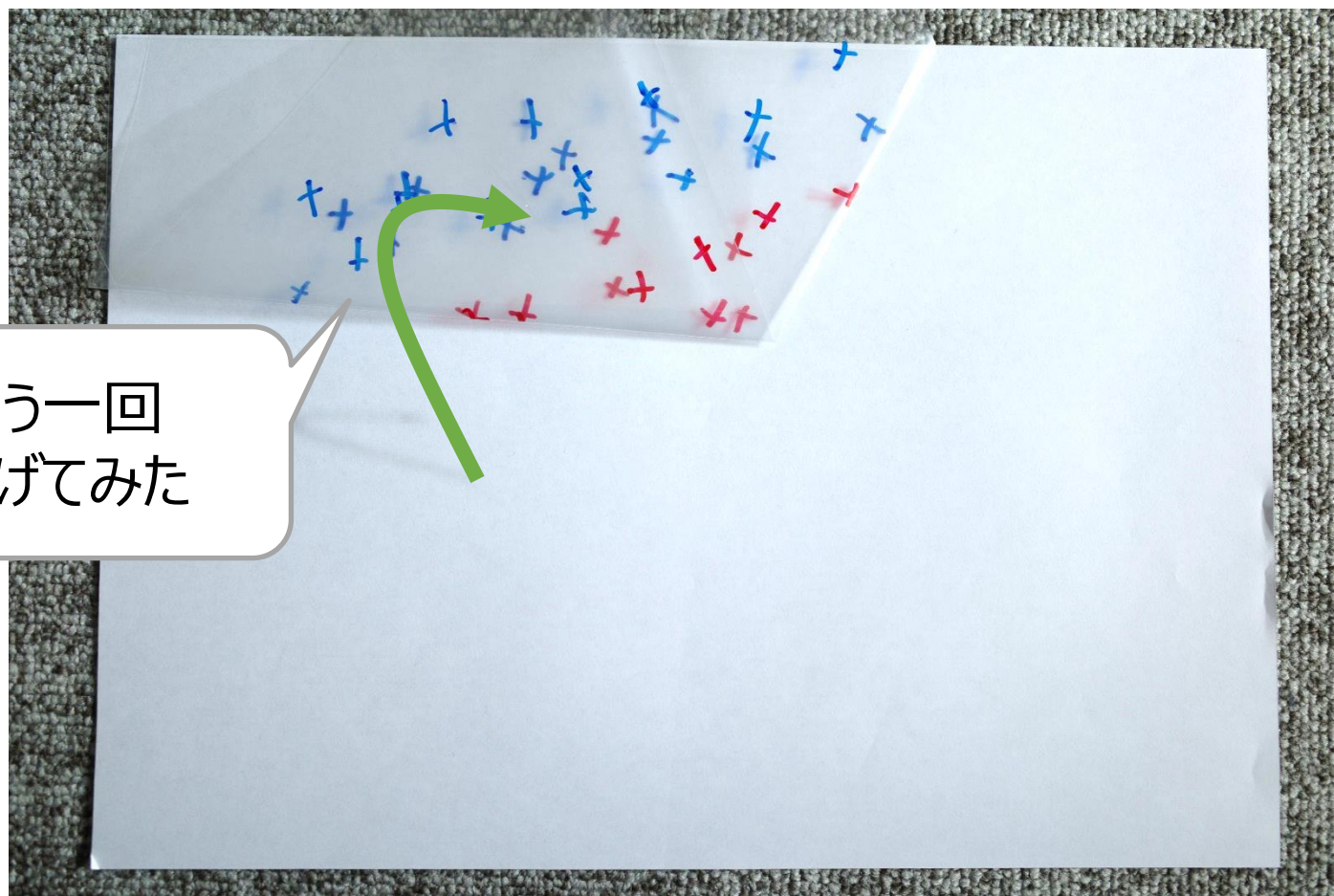
特徴抽出を繰り返すことの価値を 直感的に理解する



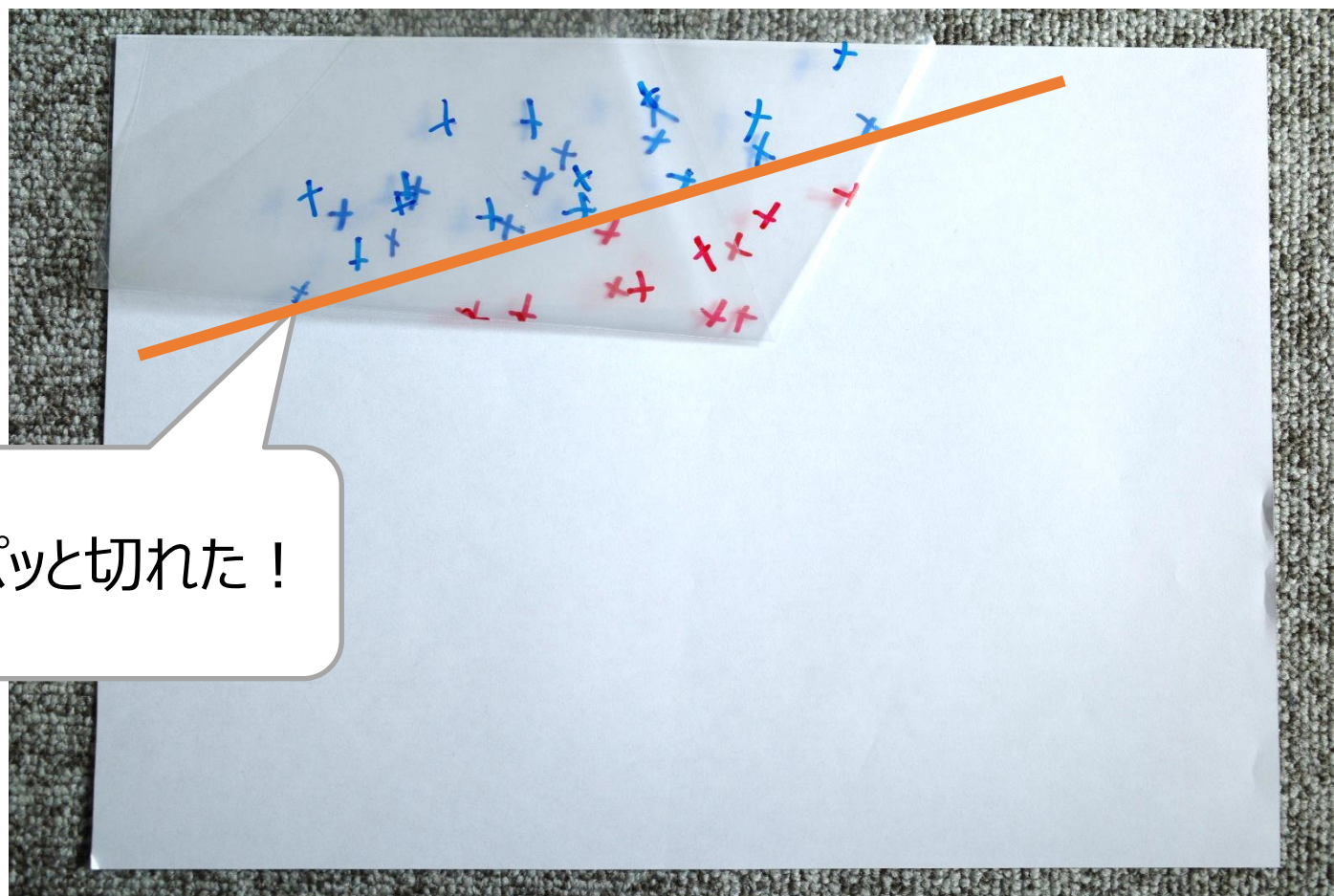
特徴抽出を繰り返すことの価値を 直感的に理解する



特徴抽出を繰り返すことの価値を 直感的に理解する



特徴抽出を繰り返すことの価値を 直感的に理解する

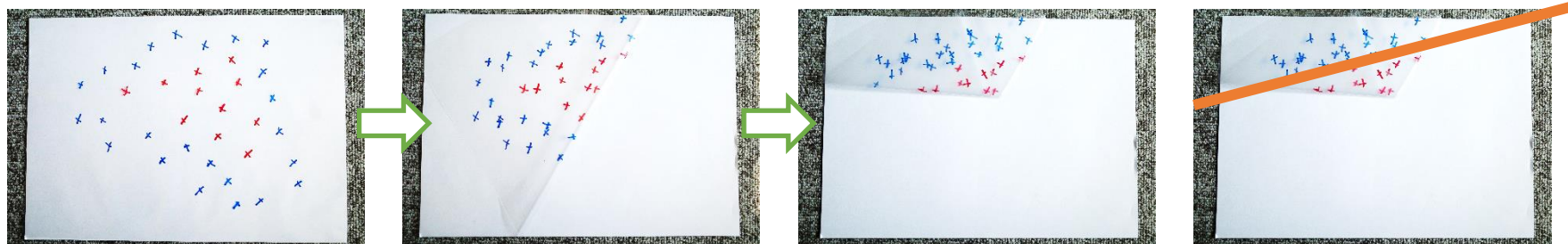
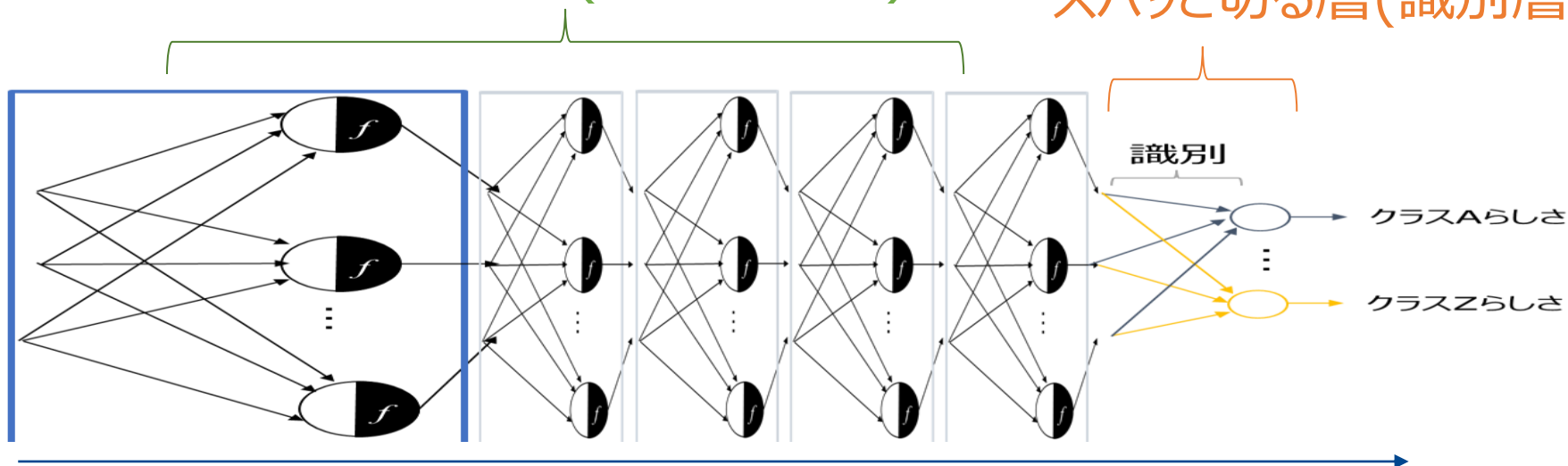


スパッと切れた！

深層ニューラルネットワークとの対応

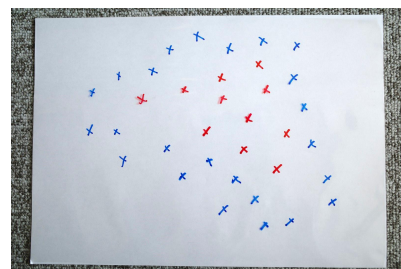
空間を曲げる層(特徴抽出層)

スパットと切る層(識別層)

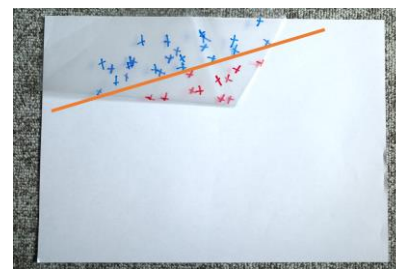


繰り返すことの価値とは

- 特徴抽出を繰り返したおかげで、単純な識別（まっすぐなハサミで一回カットしただけ）でも複雑な境界を作れる！



いきなり切り分けるには名人芸が必要



「切られる側」(データ)の分布を適切に準備しておけば、容易に切れる

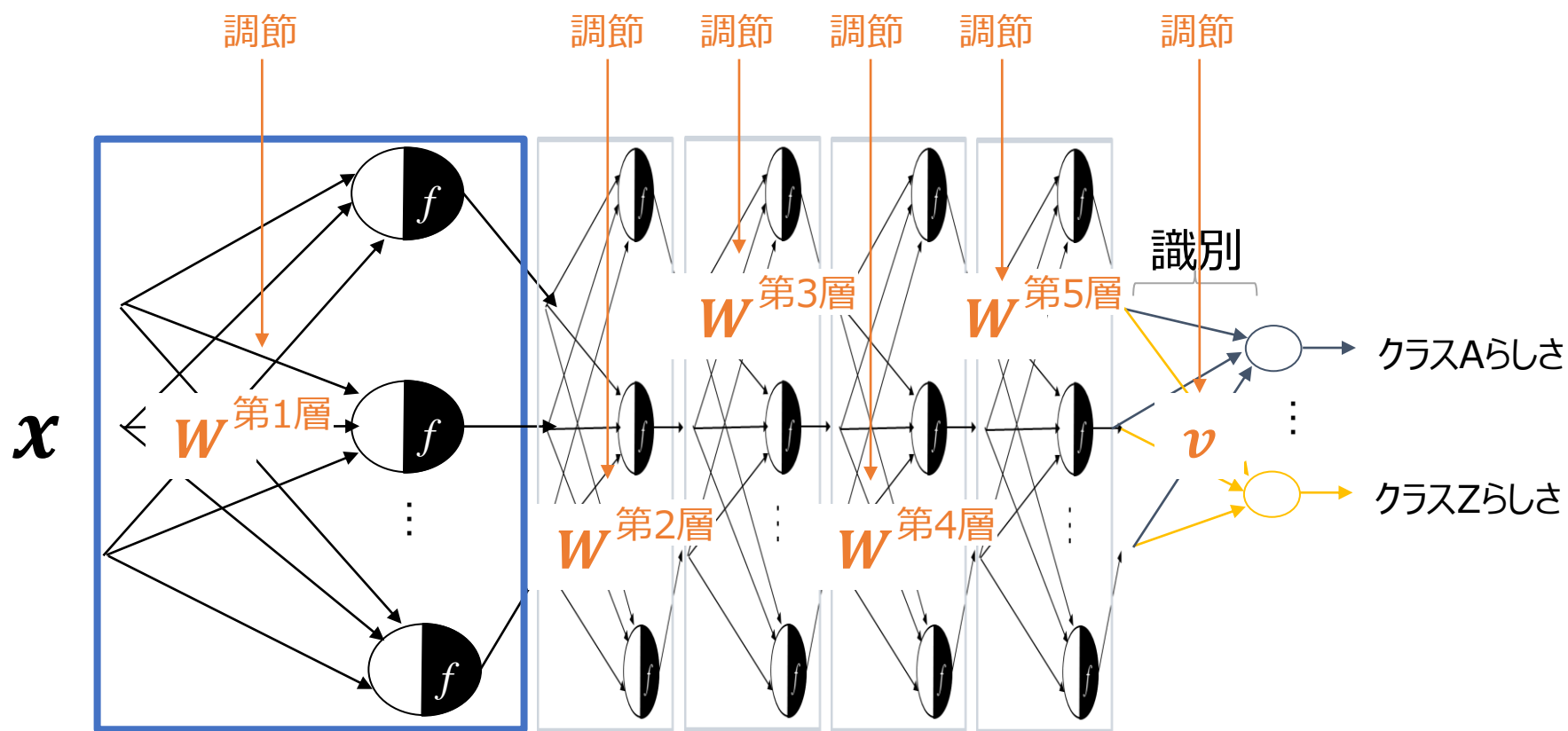
- 頑張って複雑な識別境界を構成するのではなく、簡単に識別できるようにデータ側を変換しておくという、**逆転の発想**

ニューラルネットワークの学習の概要

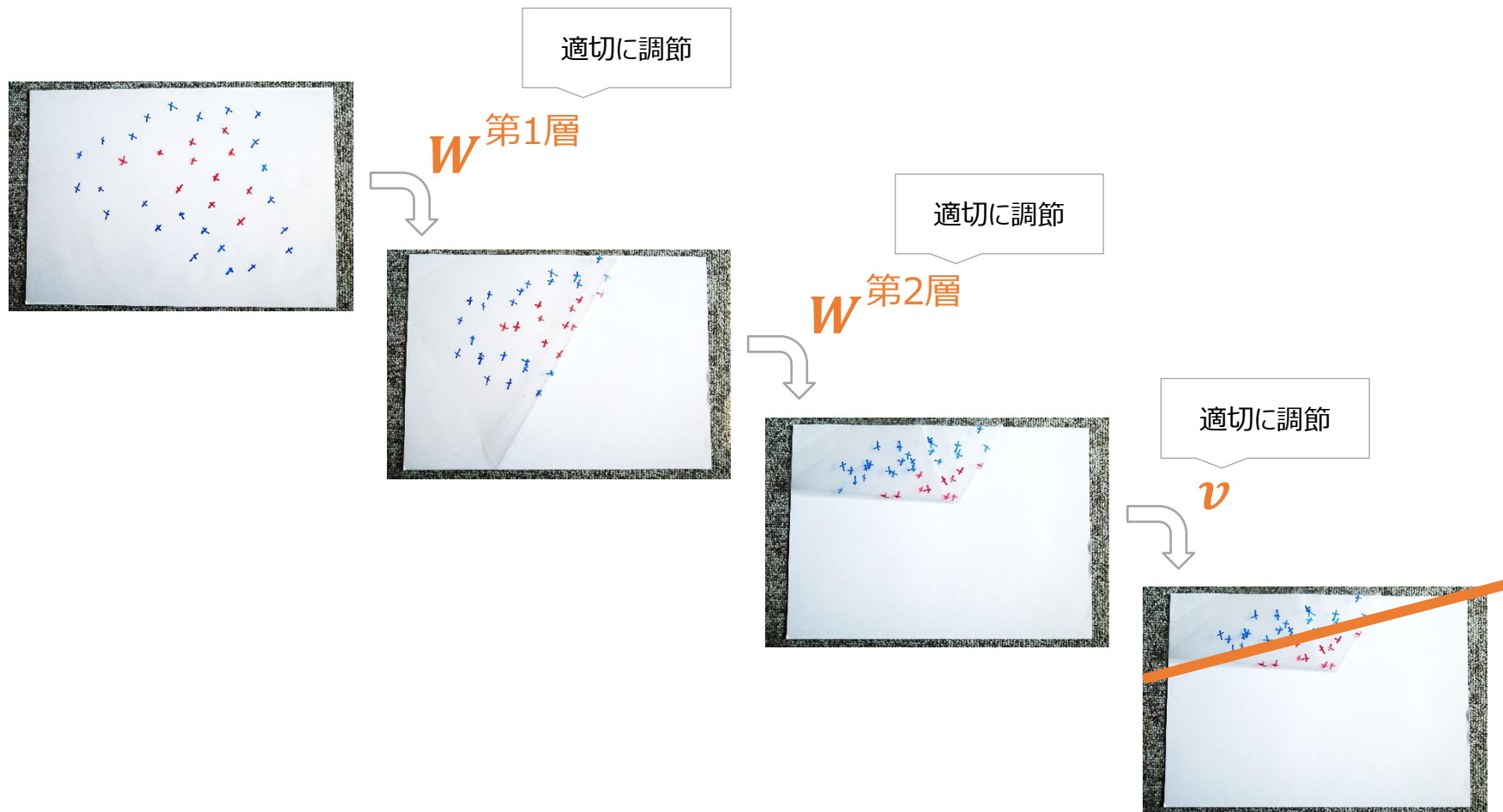
深層ニューラルネットワークによる
パターン認識⑤

ニューラルネットワークには「学習」が必要！

- 各データ x が正しいクラスに識別されるように, W や v を調節



適切な折り曲げ方（各層での特徴抽出）と 切り方（識別関数）を調節



参考：もうちょっとちゃんと言うと…：
 学習 = $\{W^{\text{第1層}}, \dots, W^{\text{第L層}}, v\}$ を調節して誤差 J を小さく

誤差

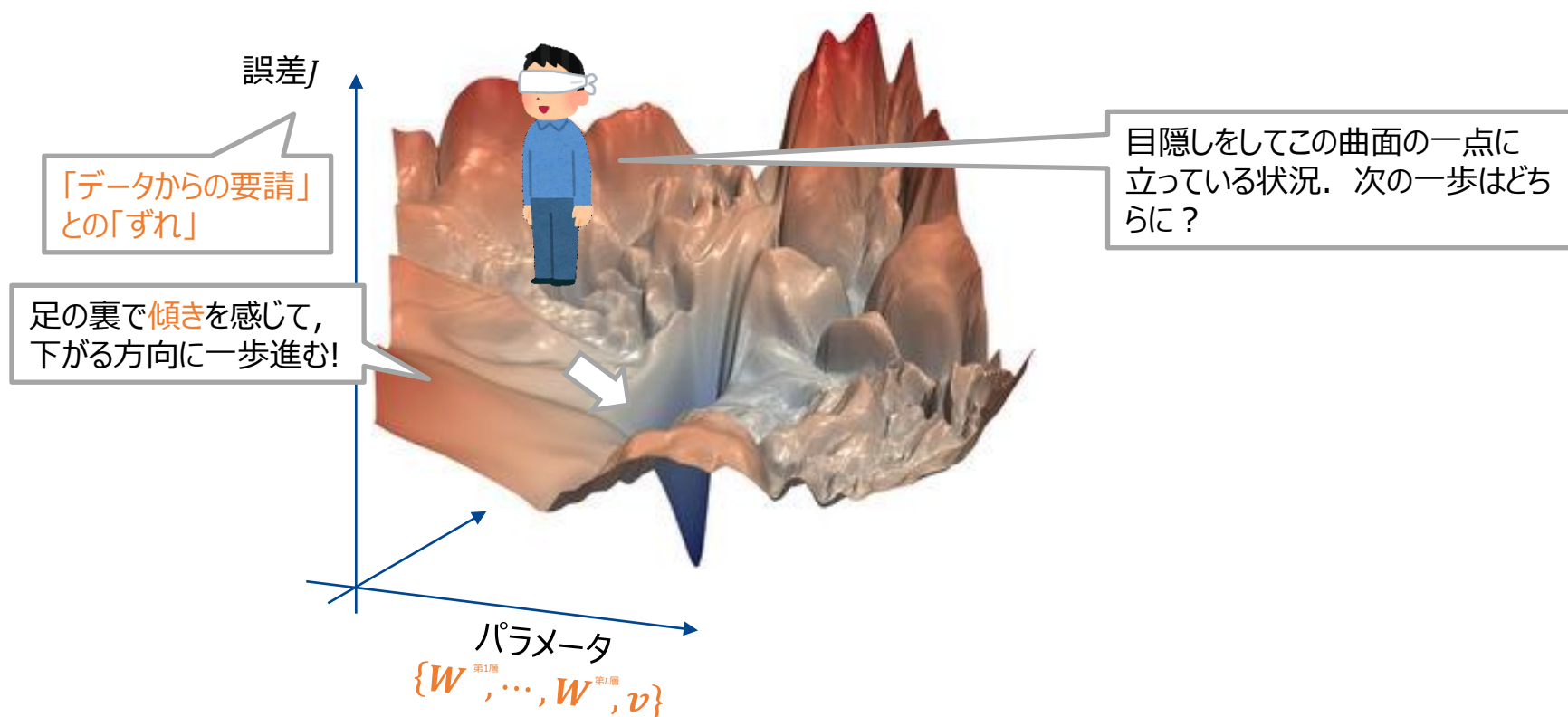
$$J = \left\| T - f(v f(W^{\text{第L層}} \dots f(W^{\text{第2層}} f(W^{\text{第1層}} x))) \right\|$$

x に対する正解(教師)

ネットワークの出力

※実際には v も W と同様に扱えるので、学習時に
 区別する必要はないが、これまでの説明との一貫性
 のため、あえて別のものと表記している

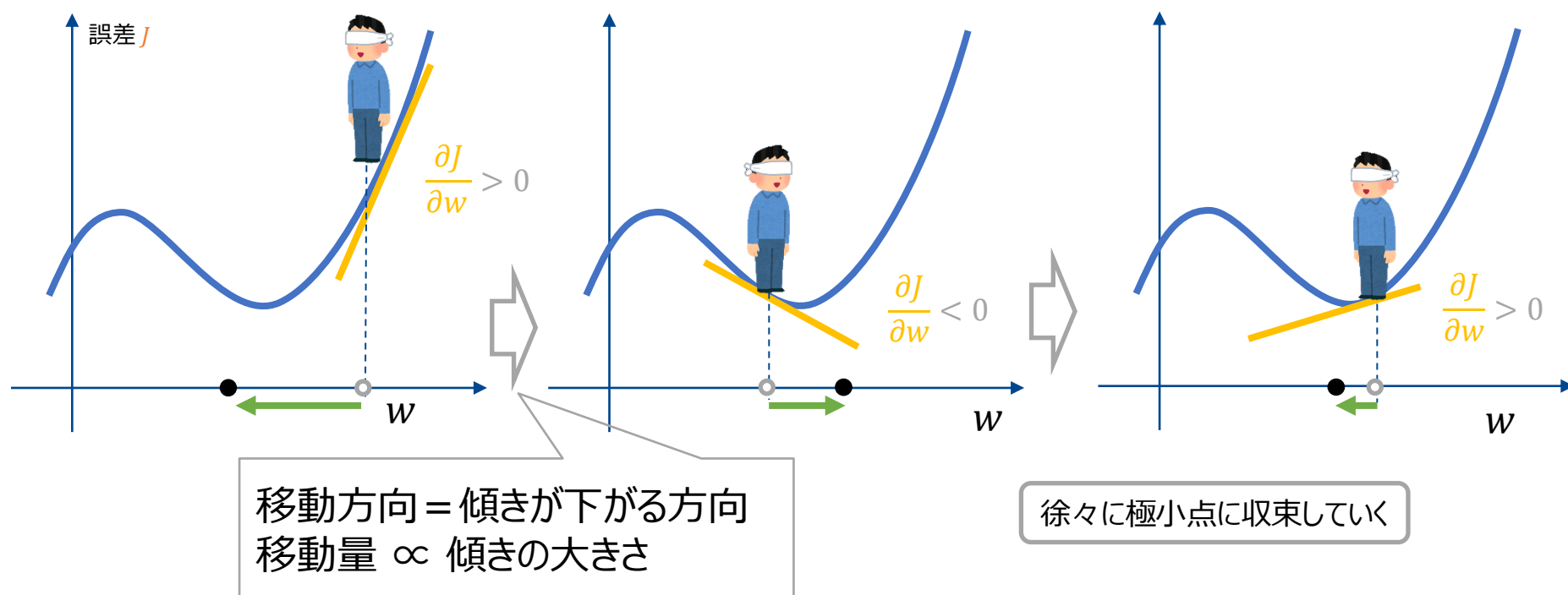
適切なパラメータを探す「地図のない」旅



<https://github.com/tomgoldstein/loss-landscape>
[Li+, "Visualizing the Loss Landscape of Neural Nets," NIPS2018]

この旅の手掛かりは「微分」＝「傾き」

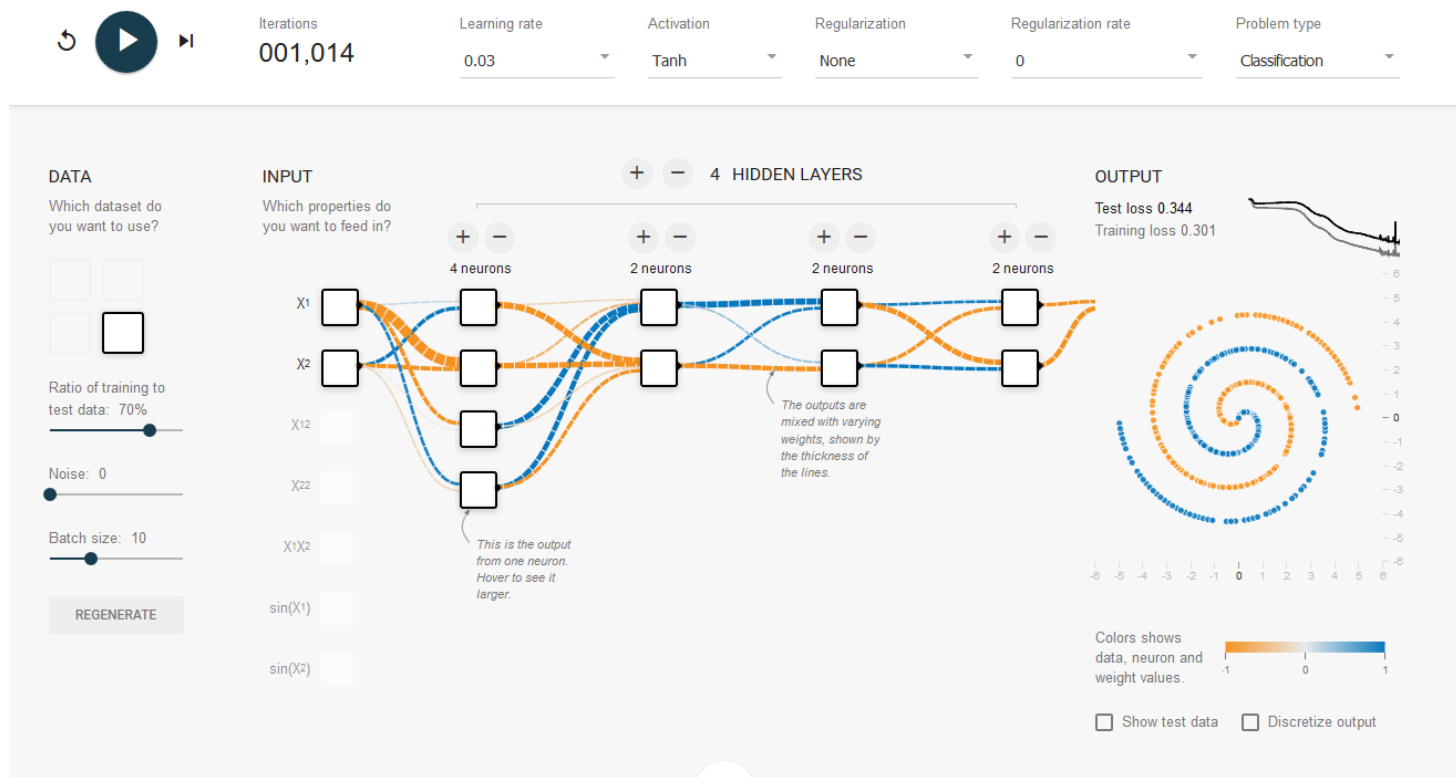
- $\{W^{\text{第1層}}, \dots, W^{\text{第L層}}, v\}$ の1要素 w だけを取り出すと



- 誤差 $J = 0$ になるところが谷底 (= すべて正しく認識)
 - ただし存在するかどうかは「？」

<http://playground.tensorflow.org>

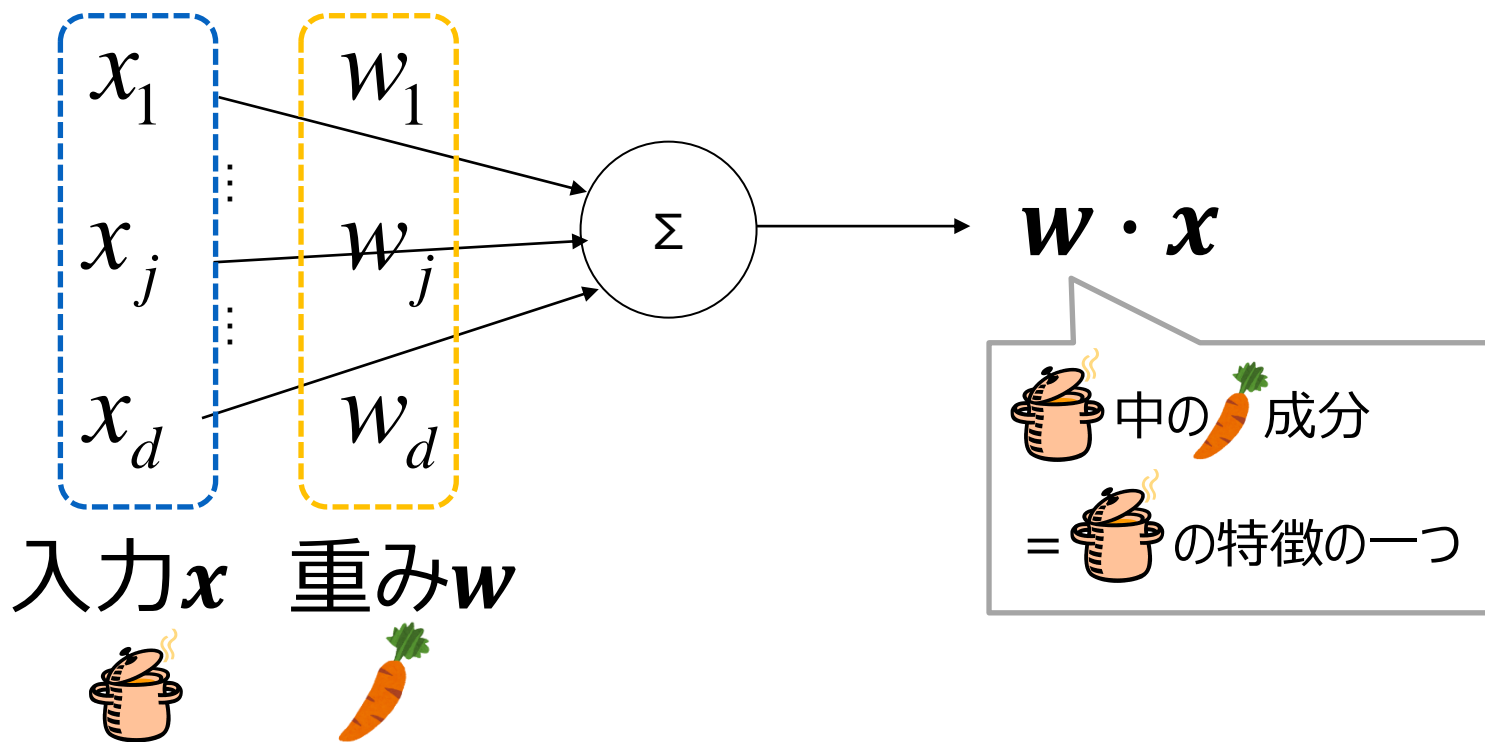
Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.



【付録】 表現学習

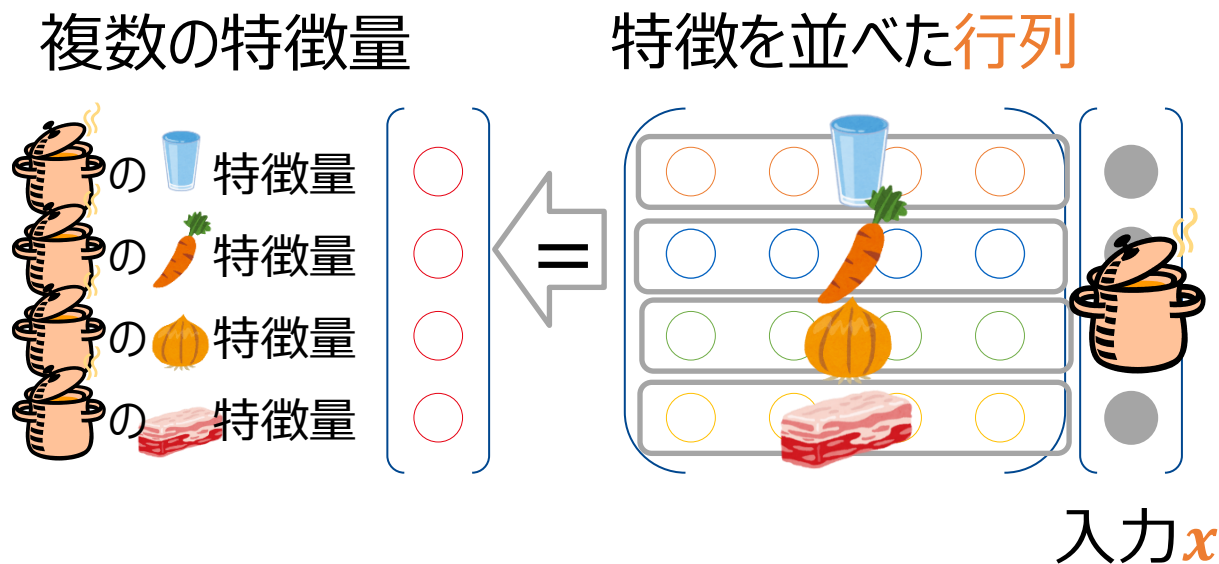
深層ニューラルネットワークによる
パターン認識⑥

重み w が x の特徴の一つを定める
その重み w が、学習で自動的に決定されるので…



表現学習 = 認識に有効な「特徴」を自動的に見つけてくれるということ！

- 認識にとってベストな 🥕 や 🍖 のようなものを自動決定！



- 以前「googleが深層学習で猫を自動的に見つけた」と話題に
- 認識対象のことを何も知らなくても、高精度な識別器を構成できる！



<https://googleblog.blogspot.jp/2012/06/>

表現学習は 「パターン認識」を民主化した

- 例えば「文字を認識するためには，こんな特徴がよい！」といった専門的な知見がなくても，文字認識が可能に
- Kaggleコンペティション「くずし字認識」2019の上位2名は，日本人ではない



■ Prize Winners

#	△	Team	Members
1	—	tascj	
2	—	Konstantin Lopuhin	
3	—	Kenji	

tascj

at
Hangzhou, Zhejiang, China

Konstantin Lopuhin

ML Engineer at Zyte
Tbilisi, Tbilisi, Georgia

<https://www.kaggle.com/c/kuzushiji-recognition/leaderboard>

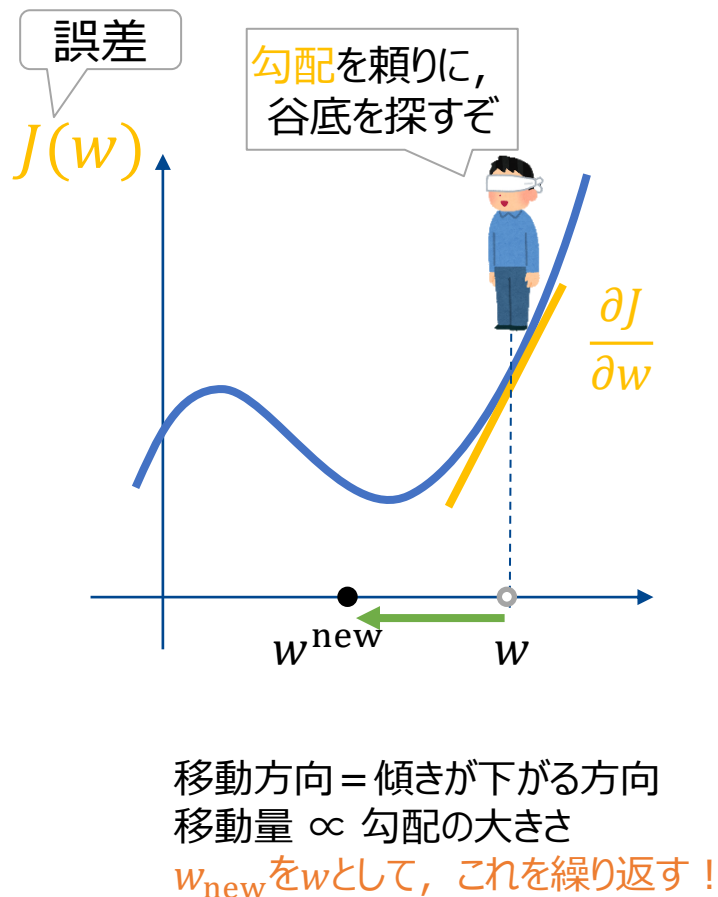
- 特徴抽出以外の部分の設計には専門知識が必要だが，それでも昔に比べて格段と取り組みやすく
 - 本当に専門的知見が不要になったかは，今後の歴史が証明するだろう...

【付録】

ニューラルネットワークの学習の詳細に入る
前の2つの「入れ知恵」：
「勾配降下法」と「合成関数の微分」

深層ニューラルネットワークによる
パターン認識⑦

入れ知恵①勾配降下法： 微分（勾配）を用いた「関数の極値探索法」



繰り返す

$$w^{\text{new}} = w - \eta \cdot \frac{\partial J}{\partial w}$$

定数

シンプルな
記法 $\nabla_w J$

繰り返す

$$w^{\text{new}} = w - \eta \cdot \nabla_w J$$



入れ知恵②合成関数の微分

- $y = f(g(x))$ で, x による y の変化 (要するに微分 = 勾配) が知りたいときは...

$$\frac{dy}{dx} = \frac{df(g(x))}{dx} = \frac{df(g)}{dg} \frac{dg}{dx}$$

- シンプルな例:

- $y = f(w \cdot x)$ なら (すなわち $g = w \cdot x$) $\frac{dy}{dx} = f'(w \cdot x) w$



ターゲット x に
たどり着くまで
分解していく

- もっと合成が深い $y = f(g(h(x)))$ なら

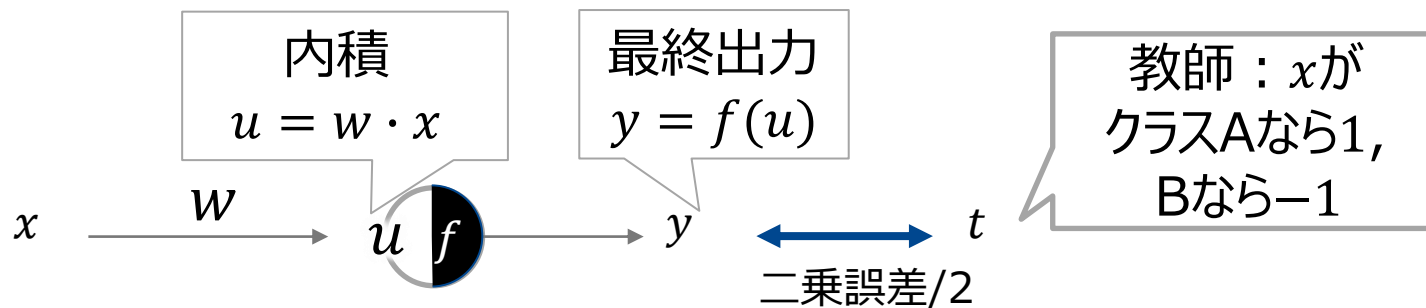
$$\frac{dy}{dx} = \frac{df(g(h(x)))}{dx} = \frac{df(g)}{dg} \frac{dg(h)}{dx} = \frac{df(g)}{dg} \frac{dg(h)}{dh} \frac{dh}{dx}$$

【付録】

ニューラルネットワークの学習の詳細： 誤差逆伝播法（バックプロパゲーション）

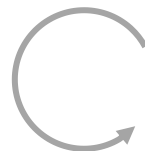
深層ニューラルネットワークによる
パターン認識⑧

最初はニューロン1個の超シンプルなネットワーク



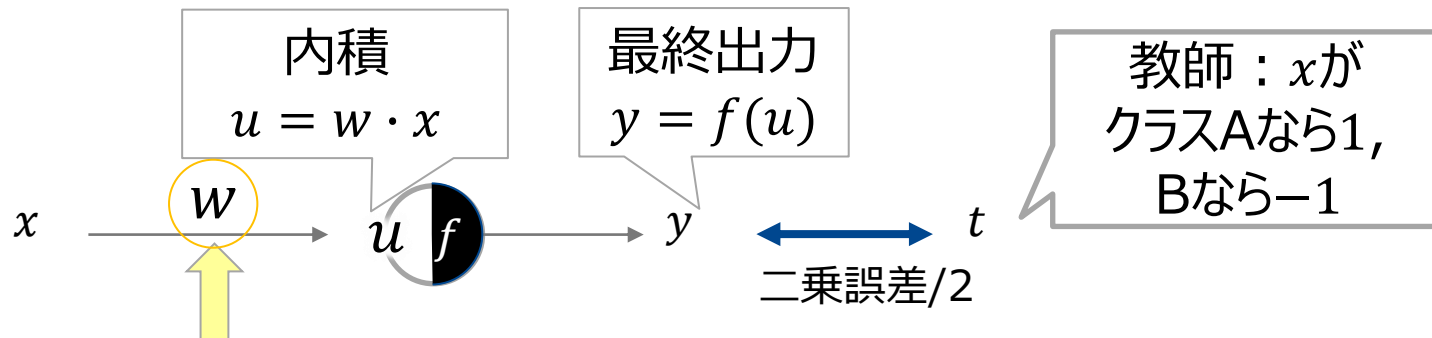
- $J(w) = (y - t)^2 / 2$
- あとは、適当な w からスタートして勾配降下を繰り返せばOK

繰り返す



$$w^{\text{new}} = w - \eta \cdot \nabla_w J$$

重み更新のキモである 勾配 $\nabla_w J = \frac{\partial J(w)}{\partial w}$ はどうなる？



- $J(w) = (y - t)^2 / 2$ だから

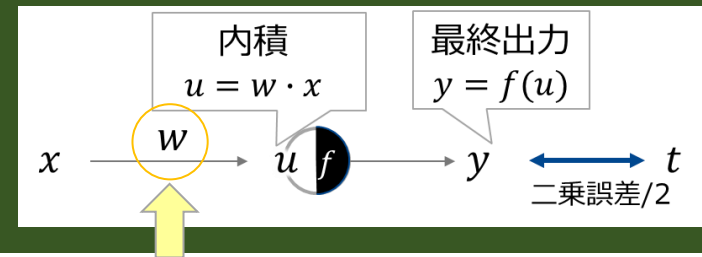
$$\begin{aligned}\nabla_w J &= \frac{\partial J(w)}{\partial w} = \frac{\partial J(w)}{\partial y} \frac{\partial y}{\partial w} = (y - t) \frac{df(u)}{dw} \\ &= (y - t) \frac{df(u)}{du} \frac{du}{dw} = (y - t) \cdot f'(u) \cdot x\end{aligned}$$

何が起きた？



落ち着いて
考えれば
大丈夫

$\nabla_w J$ は？ 落ち着こうバージョン



$$\begin{aligned}
 \nabla_w J &= \frac{\partial J(w)}{\partial w} \\
 &= \frac{\partial J(w)}{\partial y} \frac{\partial y}{\partial w} \\
 &= (y - t) \frac{\partial f(u)}{\partial w} \\
 &= (y - t) \frac{df(u)}{du} \frac{\partial u}{\partial w} \\
 &= (y - t) f'(u) x
 \end{aligned}$$

合成関数の微分

合成関数の微分

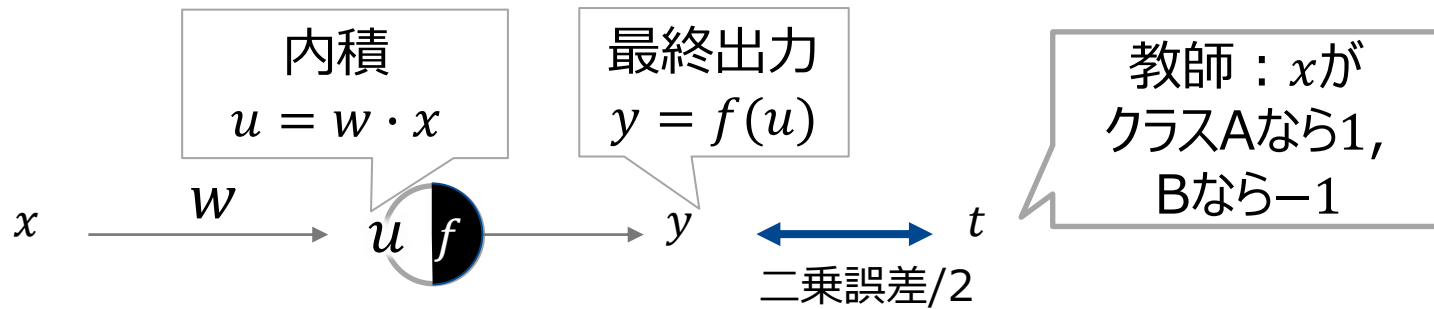
$\frac{\partial(y - t)^2/2}{\partial y}$

$\frac{\partial w \cdot x}{\partial w}$

右端がどんどん
分解されてるだけ

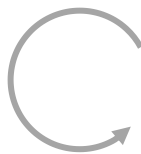


というわけで、ニューロン1個の 超シンプルなネットワークの学習法



- $J(w) = (y - t)^2 / 2$
- あとは、適当な w からスタートして以下を繰り返せばOK

繰り返す



$$w^{\text{new}} = w - \eta \cdot (y - t) \cdot f'(u) \cdot x$$

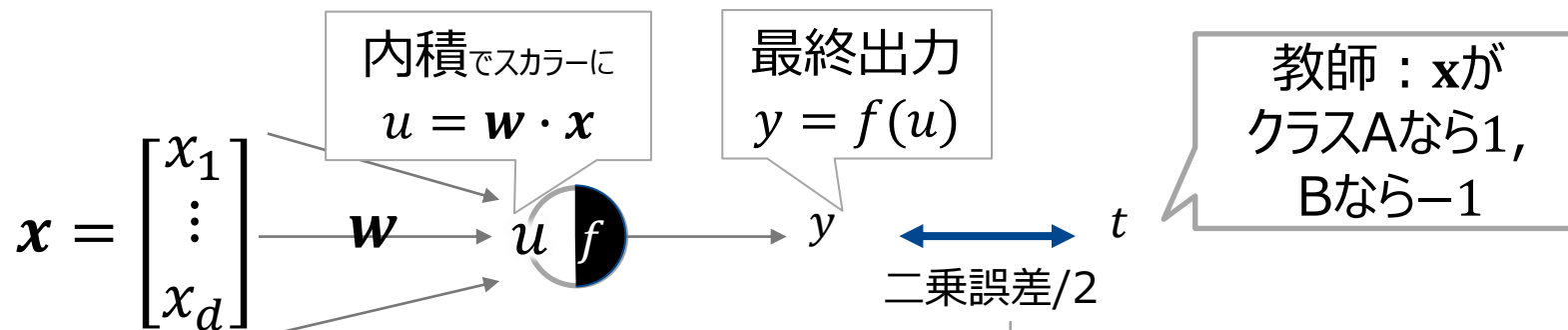
できた！



注： y や u の計算には w が使われているので、右辺第二項も w と関係してます

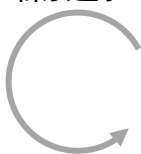
注：微分項 $f'(u)$ が残ってるのは気持ち悪いですが、 f の具体形を与えれば消えます

実はベクトル入力でも 勾配の式はほとんど同じ



- $J(\mathbf{w}) = (y - t)^2 / 2$
- あとは、適当な \mathbf{w} からスタートして以下を繰り返せばOK

繰り返す

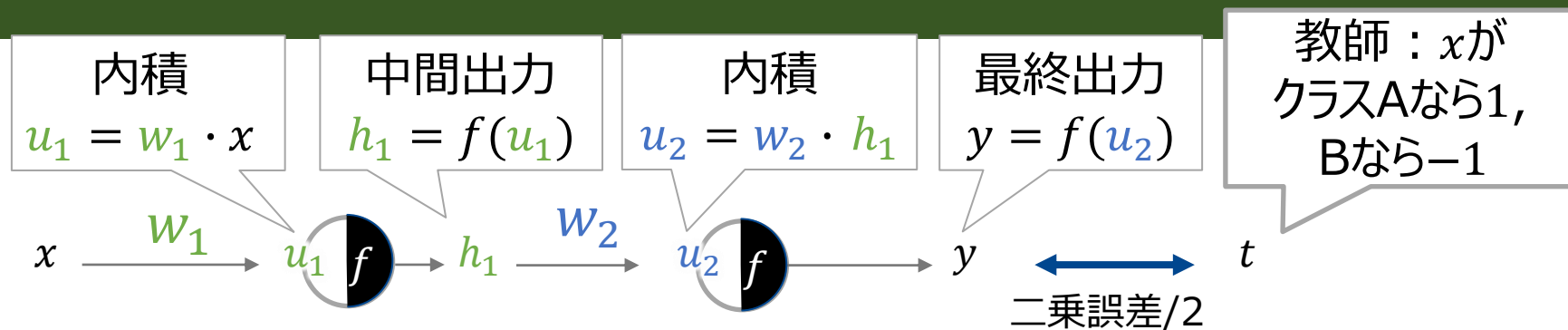


$$\mathbf{w}^{\text{new}} = \mathbf{w} - \eta \cdot (y - t) \cdot f'(u) \cdot \mathbf{x}$$

\mathbf{w} と \mathbf{x} が
太字になっただけ

というわけで、その辺は
あまり気にせず行きましょう

では2層（各層ニューロン1個）のネットワーク

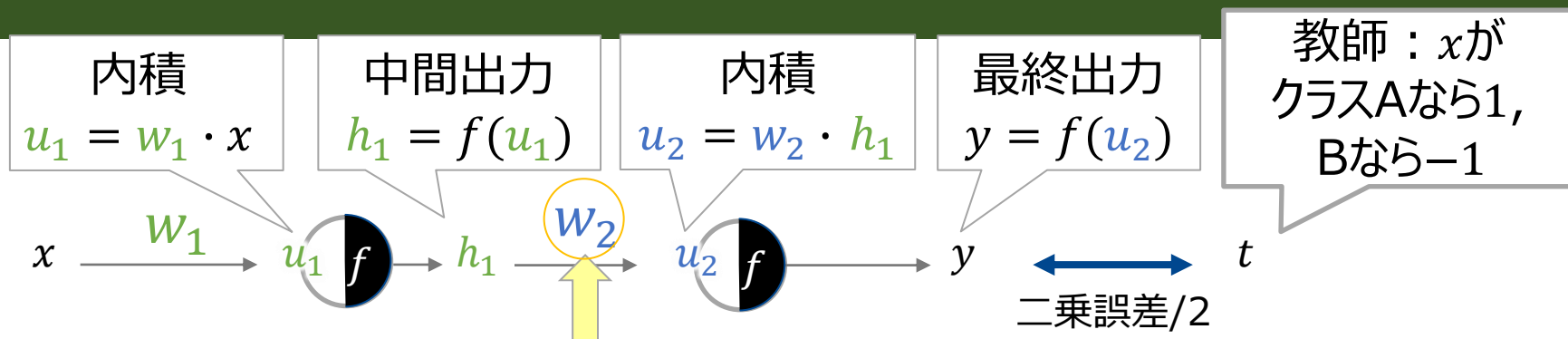


- $J(w_1, w_2) = (y - t)^2 / 2$
- あとは、適当な w_1, w_2 からスタートして以下を繰り返せばOK

繰り返す

$$\begin{aligned} w_1^{\text{new}} &= w_1 - \eta \cdot \nabla_{w_1} J \\ w_2^{\text{new}} &= w_2 - \eta \cdot \nabla_{w_2} J \end{aligned}$$

$\nabla_{w_2} J$ (=誤差 J を減らす方向を示す w_2 の勾配) はどうなる？



• $J(w_1, w_2) = (y - t)^2 / 2$ だから

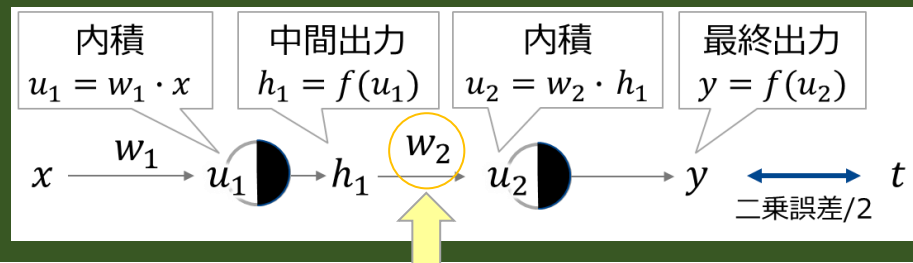
$$\begin{aligned} \nabla_{w_2} J &= \frac{\partial J(w_1, w_2)}{\partial w_2} = \frac{\partial J(w_1, w_2)}{\partial y} \frac{\partial y}{\partial w_2} = (y - t) \frac{\partial f(u_2)}{\partial w_2} \\ &= (y - t) \frac{df(u_2)}{du_2} \frac{\partial u_2}{\partial w_2} = (y - t) \cdot f'(u_2) \cdot h_1 \end{aligned}$$

また微分だらけ



落ち着いて
考えれば
大丈夫

$\nabla_{w_2} J$ は？ 落ち着こうバージョン



$$\begin{aligned}
 \nabla_{w_2} J &= \frac{\partial J(w_1, w_2)}{\partial w_2} \\
 &= \frac{\partial J(w_1, w_2)}{\partial y} \frac{\partial y}{\partial w_2} \\
 &= (y - t) \frac{\partial f(u_2)}{\partial w_2} \\
 &= (y - t) \frac{df(u_2)}{du_2} \frac{\partial u_2}{\partial w_2} \\
 &= (y - t) f'(u_2) h_1
 \end{aligned}$$

合成関数の微分

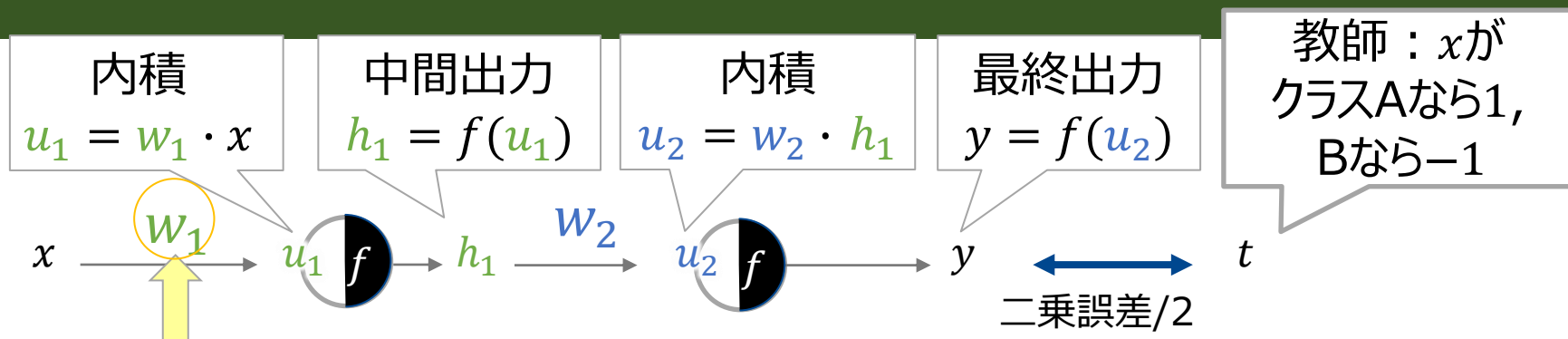
合成関数の微分

右端がどんどん
分解されてるだけ



h_1 より入力側には
 w_2 はないので、
ここでおしまい

$\nabla_{w_1} J$ (=誤差 J を減らす方向を示す w_1 の勾配) はどうなる？



• $J(w_1, w_2) = (y - t)^2 / 2$ だから

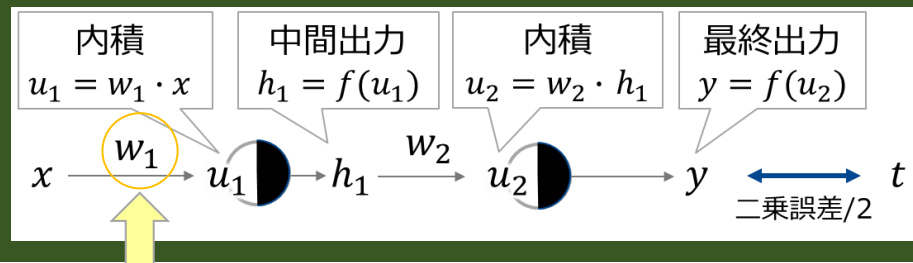
$$\begin{aligned}
 \nabla_{w_1} J &= \frac{\partial J(w_1, w_2)}{\partial w_1} = \frac{\partial J(w_1, w_2)}{\partial y} \frac{\partial y}{\partial w_1} = (y - t) \frac{df(u_2)}{dw_1} \\
 &= (y - t) \frac{df(u_2)}{du_2} \frac{du_2}{dw_1} = (y - t) f'(u_2) \frac{du_2}{dh_1} \frac{dh_1}{dw_1} \\
 &= (y - t) f'(u_2) w_2 \frac{df(u_1)}{du_1} \frac{du_1}{dw_1} \\
 &= (y - t) \cdot f'(u_2) \cdot w_2 \cdot f'(u_1) \cdot x
 \end{aligned}$$

もっとすごいのが来た！



落ち着いて
考えれば
大丈夫

$\nabla_{w_1} J$ は？ 落ち着こうバージョン

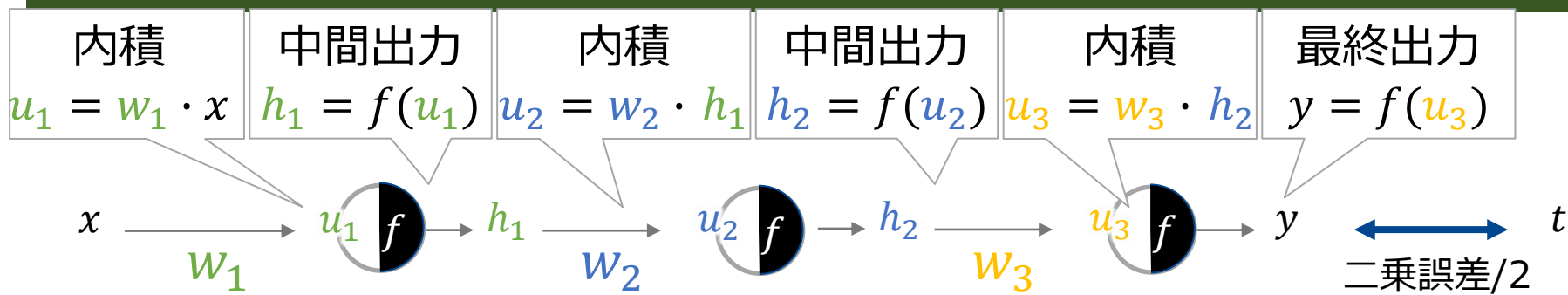


$$\begin{aligned}
 \nabla_{w_2} J &= \frac{\partial J(w_1, w_2)}{\partial w_1} \\
 &= \frac{\partial J(w_1, w_2)}{\partial y} \frac{\partial y}{\partial w_1} \quad \text{合成関数の微分} \\
 &= (y - t) \frac{\partial f(u_2)}{\partial w_1} \\
 &= (y - t) \frac{df(u_2)}{du_2} \frac{\partial u_2}{\partial w_1} \quad \text{合成関数の微分} \\
 &= (y - t) f'(u_2) \frac{\partial u_2}{\partial h_1} \frac{\partial h_1}{\partial w_1} \quad \text{合成関数の微分} \\
 &= (y - t) f'(u_2) w_2 \frac{df(u_1)}{du_1} \frac{\partial u_1}{\partial w_1} \quad \text{合成関数の微分} \\
 &= (y - t) f'(u_2) w_2 f'(u_1) x
 \end{aligned}$$

やはり右端がどんどん
分解されてるだけ

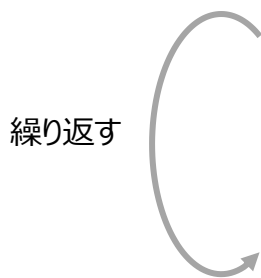


ここまで来たら3層のネットワークも！



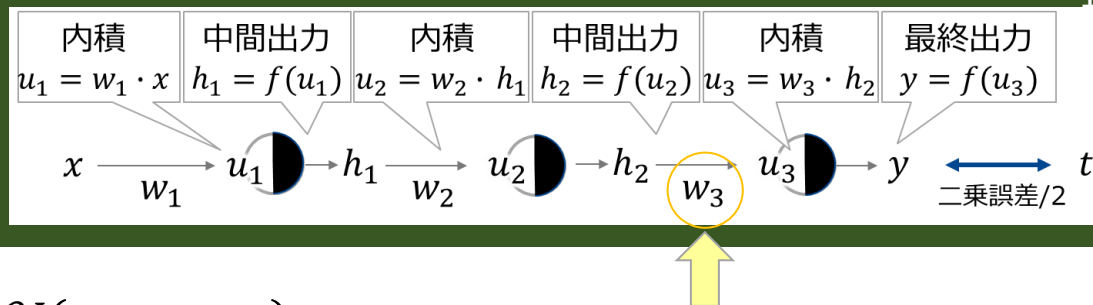
- $J(w_1, w_2, w_3) = (y - t)^2 / 2$

- あとは、適当な w_1, w_2, w_3 からスタートして以下を繰り返せばOK



$$\begin{aligned} w_1^{\text{new}} &= w_1 - \eta \cdot \nabla_{w_1} J \\ w_2^{\text{new}} &= w_2 - \eta \cdot \nabla_{w_2} J \\ w_3^{\text{new}} &= w_3 - \eta \cdot \nabla_{w_3} J \end{aligned}$$

$\nabla_{w_3} J$ は？ 落ち着こうバージョン



$$\begin{aligned}
 \nabla_{w_3} J &= \frac{\partial J(w_1, w_2, w_3)}{\partial w_3} \\
 &= \frac{\partial J(w_1, w_2, w_3)}{\partial y} \frac{\partial y}{\partial w_3} \\
 &= (y - t) \frac{df(u_3)}{du_3} \frac{\partial u_3}{\partial w_3} \\
 &= (y - t) f'(u_3) h_2
 \end{aligned}$$

合成関数の微分

合成関数の微分

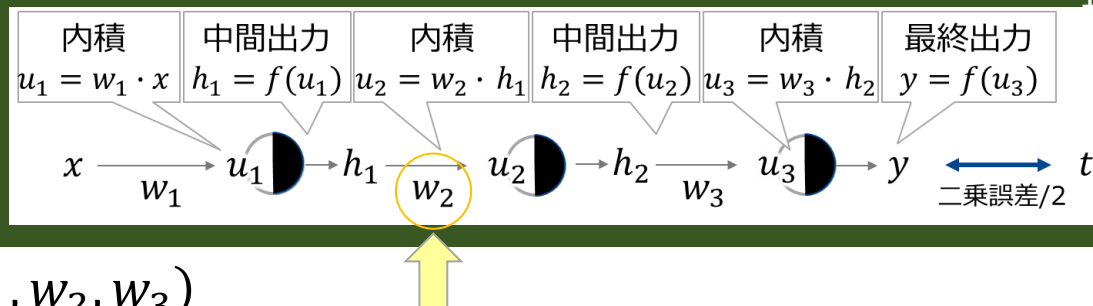
何層でも
右端がどんどん
分解されてるだけ



さらに言えば,
2層の場合の
 $\nabla_{w_2} J$ と同じ形式

h_2 より入力側には
 w_3 はないので,
ここでおしまい

$\nabla_{w_2} J$ は？ 落ち着こうバージョン



$$\nabla_{w_2} J = \frac{\partial J(w_1, w_2, w_3)}{\partial w_2}$$

やはり右端がどんどん
分解されてるだけ



さらに言えば、
2層の場合の
 $\nabla_{w_1} J$ と同じ形式で、
右端だけ違う

$$= \frac{\partial J(w_1, w_2, w_3)}{\partial y} \frac{\partial y}{\partial w_2}$$

合成関数の微分

$$= (y - t) \frac{df(u_3)}{du_3} \frac{\partial u_3}{\partial w_2}$$

合成関数の微分

$$= (y - t) f'(u_3) \frac{\partial u_3}{\partial h_2} \frac{\partial h_2}{\partial w_2}$$

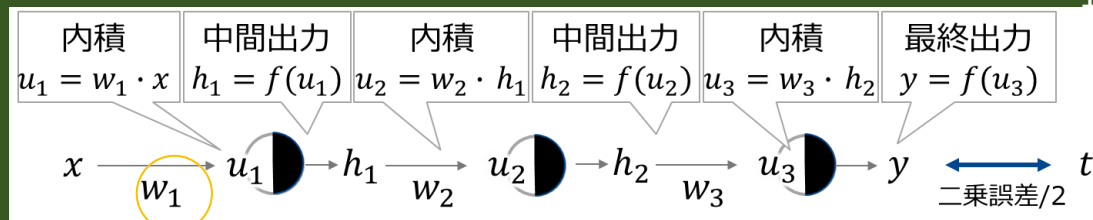
合成関数の微分

$$= (y - t) f'(u_3) w_3 \frac{df(u_2)}{du_2} \frac{\partial u_2}{\partial w_2}$$

合成関数の微分

h_1 より入力側には
 w_2 はないので、
ここでおしまい

$\nabla_{w_1} J$ は？ 落ち着こうバージョン



$$\nabla_{w_1} J = \frac{\partial J(w_1, w_2, w_3)}{\partial w_1}$$

合成関数の微分

$$= \frac{\partial J(w_1, w_2, w_3)}{\partial y} \frac{\partial y}{\partial w_1}$$

$$= (y - t) \frac{df(u_3)}{du_3} \frac{\partial u_3}{\partial w_1}$$

合成関数の微分

$$= (y - t) f'(u_3) \frac{\partial u_3}{\partial h_2} \frac{\partial h_2}{\partial w_1}$$

合成関数の微分

$$= (y - t) f'(u_3) w_3 \frac{df(u_2)}{du_2} \frac{\partial u_2}{\partial w_1}$$

合成関数の微分

$$= (y - t) f'(u_3) w_3 f'(u_2) \frac{du_z}{dh_1} \frac{\partial h_1}{\partial w_1}$$

合成関数の微分

$$= (y - t) f'(u_3) w_3 f'(u_2) w_2 \frac{df(u_1)}{du_1} \frac{\partial u_1}{\partial w_1}$$

合成関数の微分

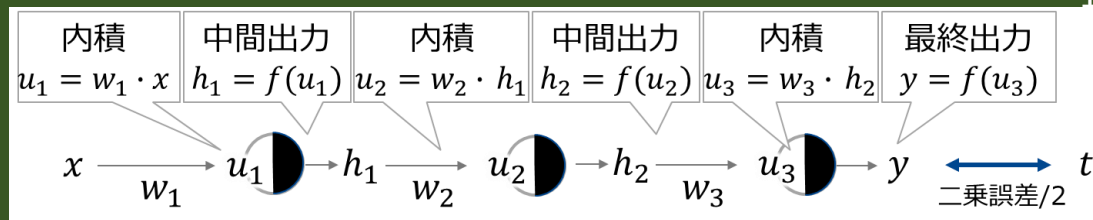
$$= (y - t) f'(u_3) w_3 f'(u_2) w_2 f'(u_1) x$$

要は、何層だろうが、
同じようなことを
繰り返せばいいわけだ



しかし、
よく書いたね、これ...

各勾配を並べてみると...(1/2)



$$\bullet \nabla_{w_3} J = (y - t) \cdot f'(u_3) \cdot h_2$$

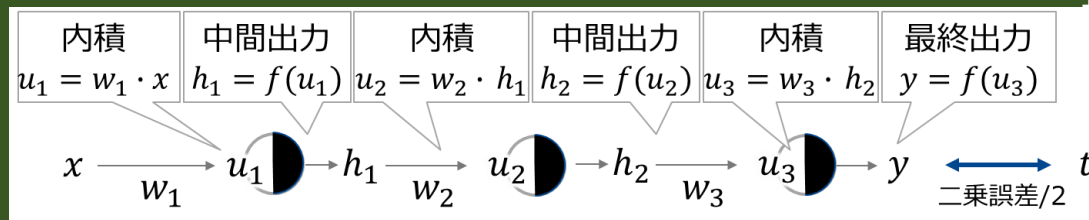
$$\bullet \nabla_{w_2} J = (y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot h_1$$

$$\bullet \nabla_{w_1} J = (y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot w_2 \cdot f'(u_1) \cdot x$$

ん～!?



各勾配を並べて みると...(2/2)



- なんと「使いまわせる」ことが発覚

$$\bullet \nabla_{w_3} J = \frac{(y - t) \cdot f'(u_3) \cdot h_2}{}$$

使いまわせる！

$$\bullet \nabla_{w_2} J = \frac{(y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot h_1}{}$$

使いまわせる！

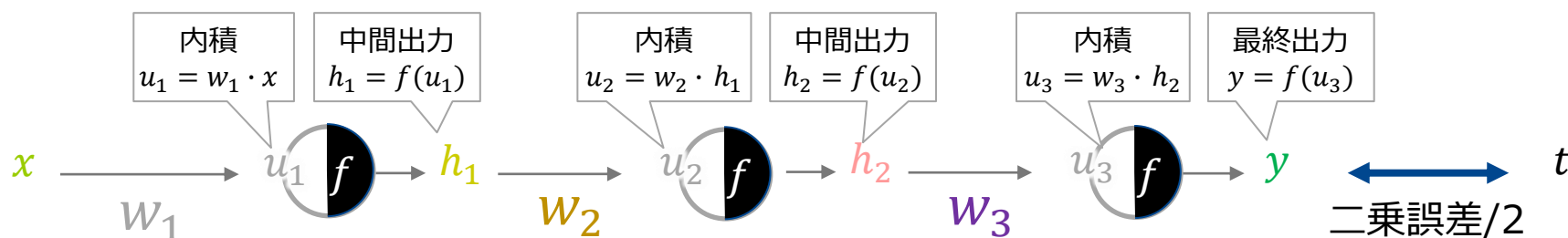
$$\bullet \nabla_{w_1} J = (y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot w_2 \cdot f'(u_1) \cdot x$$

無駄な計算が減って
効率的！



要するに、非常に効率的な計算で学習できる！

- ①ある x が与えられる
- ①前向き処理：ニューロン出力を $h_1 \rightarrow h_2 \rightarrow y$ の順に求める



- ②後向き処理：勾配を $\nabla_{w_3} J \rightarrow \nabla_{w_2} J \rightarrow \nabla_{w_1} J$ の順に求める

$$\nabla_{w_3} J = (y - t)y(1 - y)h_2$$

$$\nabla_{w_2} J = (y - t)y(1 - y)w_3 h_2(1 - h_2)h_1$$

$$\nabla_{w_1} J = (y - t)y(1 - y)w_3 h_2(1 - h_2)w_2 h_1(1 - h_1)x$$

- ③求めた勾配で w_1, w_2, w_3 をアップデート

繰り返す

$$\begin{cases} w_1^{\text{new}} = w_1 - \eta \cdot \nabla_{w_1} J \\ w_2^{\text{new}} = w_2 - \eta \cdot \nabla_{w_2} J \\ w_3^{\text{new}} = w_3 - \eta \cdot \nabla_{w_3} J \end{cases}$$

L層になっても，デルタ「 δ_l 」という記号を導入すれば もっとシンプルに書けます

- $\nabla_{w_L} J = \underline{(y - t) \cdot f'(u_L)} \cdot h_{L-1}$
 δ_L
- $\nabla_{w_{L-1}} J = \underline{\delta_L \cdot w_L \cdot f'(u_{L-1})} \cdot h_{L-2}$
 δ_{L-1}
- $\nabla_{w_{L-2}} J = \underline{\delta_{L-1} \cdot w_{L-1} \cdot f'(u_{L-2})} \cdot h_{L-3}$
 δ_{L-2}
- $\nabla_{w_l} J = \underline{\delta_{l+1} \cdot w_{l+1} \cdot f'(u_l)} \cdot h_{l-1}$
 δ_l
- $\nabla_{w_{l-1}} J = \delta_l \cdot w_l \cdot f'(u_{l-1}) \cdot h_{l-2}$

使いまわせるところを
 δ_l で表現しただけ！
でも，さらに
すっきりした！



誤差($y - t$)が，
入力に近い層の
 δ_l に伝わっていく！

これぞバックプロパゲーション（誤差逆伝播）

- ①ある x が与えられる
- ①前向き処理（順伝播）
 - ニューロン出力を $h_1 \rightarrow h_2 \rightarrow \dots \rightarrow h_L = y$ の順に求める
- ②後向き処理（逆伝播）
 - デルタを $\delta_L \rightarrow \delta_{L-1} \rightarrow \dots \rightarrow \delta_2$ の順に求めつつ
 - 勾配を $\nabla_{w_L} J \rightarrow \nabla_{w_{L-1}} J \rightarrow \dots \rightarrow \nabla_{w_1} J$ の順に求める
- ③求めた勾配で w_1, w_2, \dots, w_L をアップデート

次スライドでは
擬似コード表現

これぞバックプロパゲーション（誤差逆伝播）： 擬似コード表現

- ③ある x が与えられる
- ①前向き処理（順伝播）

```

 $h_0 \leftarrow x$ 
for  $l = 1, \dots, L$ 
   $u_l \leftarrow w_l \cdot h_{l-1}$ 
   $h_l \leftarrow f(u_l)$ 
 $y \leftarrow h_L$ 
  
```

ほとんどが
四則演算！
シンプル！



- ②後向き処理（逆伝播）

```

 $\delta_L = (y - t) \cdot f'(u_L)$ 
for  $l = L - 1, \dots, 1$ 
   $\delta_l \leftarrow \delta_{l+1} \cdot w_{l+1} \cdot f'(u_l)$ 
   $\nabla_{w_l} J \leftarrow \delta_l \cdot h_{l-1}$ 
  
```

- ③ w_1, w_2, \dots, w_L をアップデート

```

for  $l = 1, \dots, L$ 
   $w_l^{\text{new}} = w_l - \eta \cdot \nabla_{w_l} J$ 
  
```

ニューラルネットワークの学習： ここまでの話のまとめ



- ニューラルネットワークの学習は，勾配降下法
 - 微分を使って，誤差を減らすように，重み w を調整
 - 繰り返して，徐々によい w にしていく
- 「バックプロパゲーション」という考え方で，重みの調整量は効率的に求まる
 - まず，順伝播で各ニューロンの出力を求める
 - 次に，逆伝搬で各ニューロンの重みの調整量を求める
- 何層のニューラルネットワークであっても，同じ要領

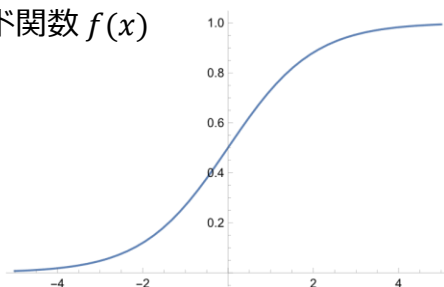
【付録】
活性化関数 $f(x)$ がシグモイド関数の
場合の誤差逆伝播法

深層ニューラルネットワークによる
パターン認識⑨

活性化関数 f によく使う (使われていた?) シグモイド関数

- (標準)シグモイド関数 $f(x) = 1/(1 + e^{-x})$

シグモイド関数 $f(x)$

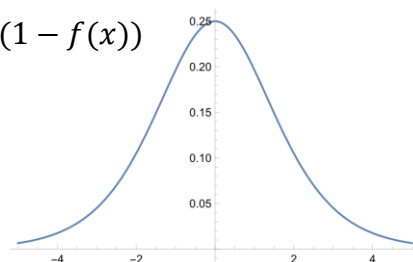


入力 x は $(-\infty, \infty)$ でも
出力は $(0, 1)$ に制約される

- 微分 : $f'(x) = e^{-x}/(1 + e^{-x})^2 = f(x) \cdot (1 - f(x))$

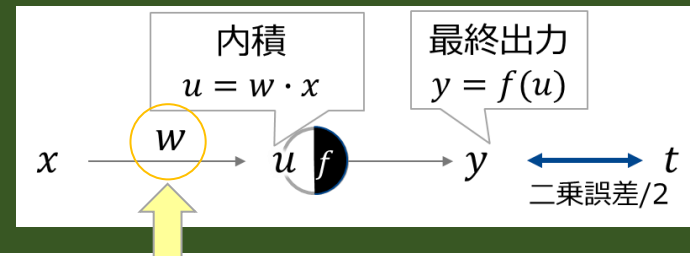
- 導出の詳細が見たい人は→ <https://qiita.com/yosshi4486/items/d111272edeba0984cef2>

$f(x)(1 - f(x))$



微分した結果が
元の関数 $f(x)$ を
用いてシンプルに
表現されるところが
面白い

1層の場合, $\nabla_w J$ はどうなる?



- $\nabla_w J = \frac{\partial J(w)}{\partial w} = (y - t) \cdot f'(u) \cdot x$

でした.

なので,

シグモイド関数の性質
 $f'(x) = f(x)(1 - f(x))$

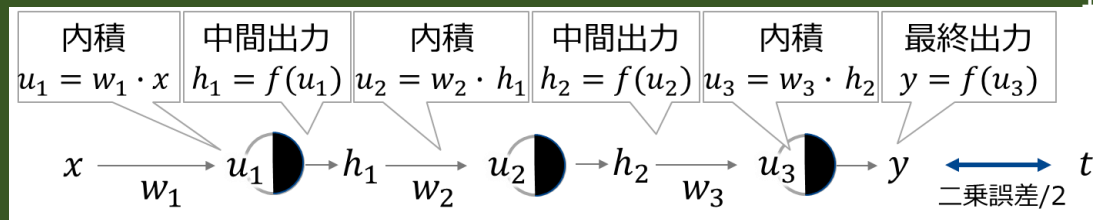
- $\nabla_w J = (y - t) \cdot f(u) \cdot (1 - f(u)) \cdot x$
 $= (y - t) \cdot y \cdot (1 - y) \cdot x$

$y = f(u)$

とっても簡単な
式になった!



3層の場合：



$$\left\{ \begin{array}{l} \bullet \nabla_{w_3} J = (y - t) \cdot f'(u_3) \cdot h_2 \\ \bullet \nabla_{w_2} J = (y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot h_1 \\ \bullet \nabla_{w_1} J = (y - t) \cdot f'(u_3) \cdot w_3 \cdot f'(u_2) \cdot w_2 \cdot f'(u_1) \cdot x \end{array} \right.$$

でした.

なので,

$$\left\{ \begin{array}{l} \bullet \nabla_{w_3} J = (y - t) \cdot y \cdot (1 - y) \cdot h_2 \\ \bullet \nabla_{w_2} J = (y - t) \cdot y \cdot (1 - y) \cdot w_3 \cdot h_2 \cdot (1 - h_2) \cdot h_1 \\ \bullet \nabla_{w_1} J = (y - t) \cdot y \cdot (1 - y) \cdot w_3 \cdot h_2 \cdot (1 - h_2) \cdot w_2 \cdot h_1 \cdot (1 - h_1) \cdot x \end{array} \right.$$

【付録】

もう少し詳細化, その1 :

データ x は一つじゃない!

数多くの学習データを用いて学習する

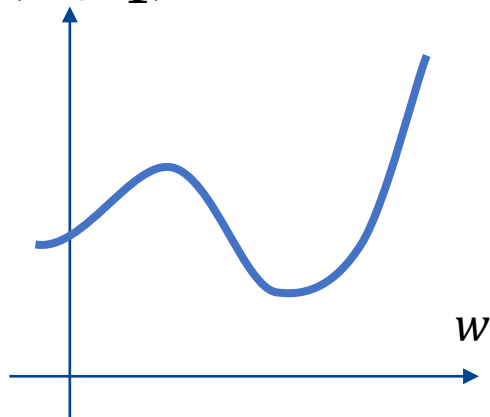
深層ニューラルネットワークによる
パターン認識⑩

さて、ニューラルネットワークの学習だと

- 各学習データ $\{x_n\}$ がそれぞれ「こういう w がいい」と要求

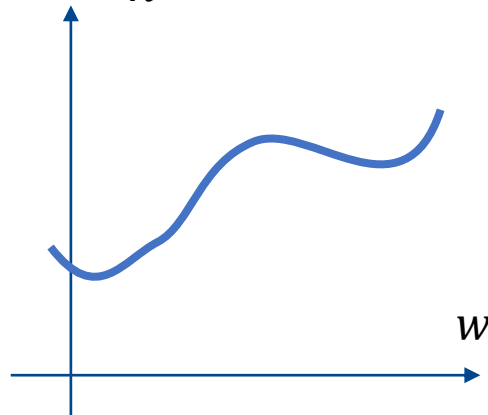
小さいほど x_1 を正しく認識

$J(w|x_1)$



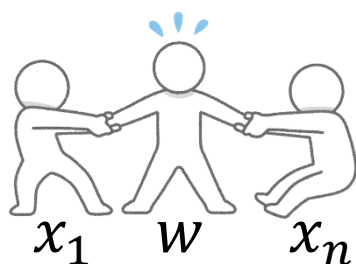
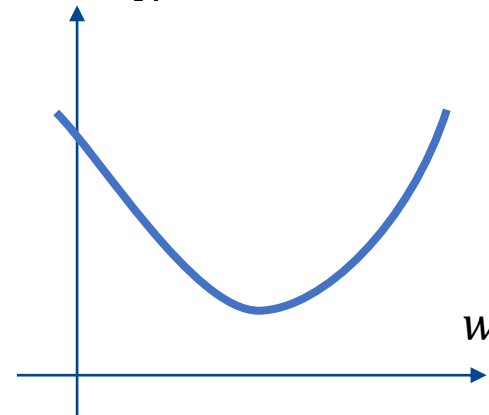
$J(w|x_n)$

...



...

$J(w|x_N)$



どうまとめるかな...



全学習データを考えた場合のバックプロパゲーション (オレンジが単一データの場合との違い)

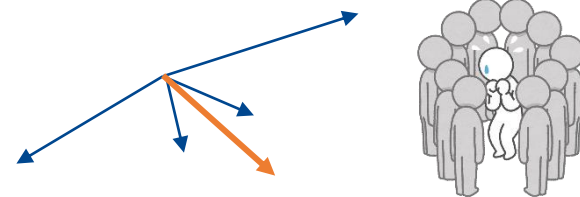
- ①ある $x \in x_1 \cdots x_n \cdots x_N$ が与えられる
- ①前向き処理 (順伝播)
 - ニューロン出力を $h_1 \rightarrow h_2 \rightarrow \cdots \rightarrow h_L = y$ の順に求める

- ②後向き処理 (逆伝播)
 - デルタを $\delta_L \rightarrow \delta_{L-1} \rightarrow \cdots \rightarrow \delta_2$ の順に求めつつ
 - 勾配を $\nabla_{w_L} J \rightarrow \nabla_{L-1} J \rightarrow \cdots \rightarrow \nabla_{w_1} J$ の順に求める

要するに各学習データが,
「こっちに w_l を変更してほしい」
と好き勝手言うので,
それらの合計を使えばOK!

- ①ー②を繰り返す (勾配も複数求まる)

- ③求めた複数の勾配の合計で w_1, w_2, \dots, w_L をアップデート



実はいろいろバリエーションあり

- 一括学習（バッチ学習）

- 全 N データの勾配を合計して重み w を更新
 - ネットワークとデータが大規模になると、勾配情報の記憶も大変に...



- ミニバッチ学習

- 各ミニバッチのデータの勾配を合計したもので重み w を更新
 - ミニバッチ = 全 N データを K 分割したもの
 - ミニバッチを固定する場合もあり、毎度ランダムに構成する場合もあり



- 逐次学習（オンライン学習）

- データ x が1つ入る度に重み w をアップデート(合計しない)
 - 学習に用いるデータをランダムに選ぶ方法もあり（確率的勾配降下法）



【付録】

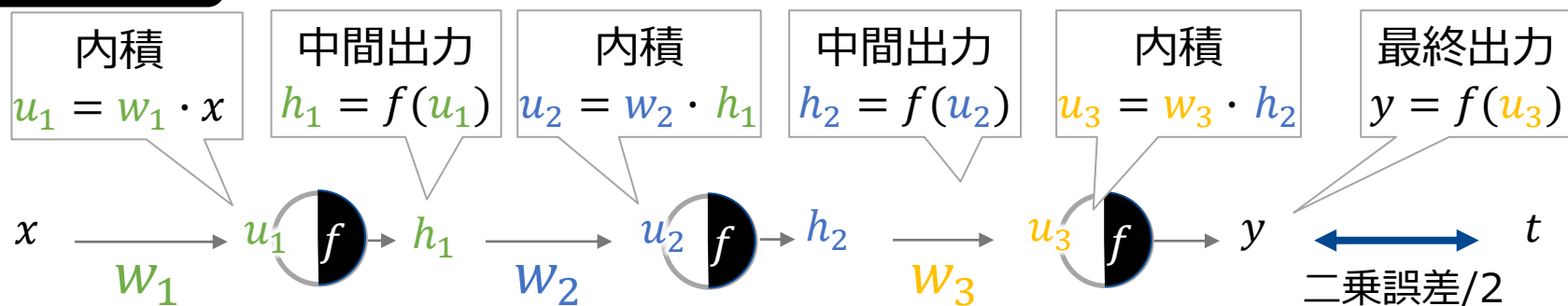
もう少し詳細化, その2 :
各層のニューロンは1つじゃない

深層ニューラルネットワークによる
パターン認識⑪

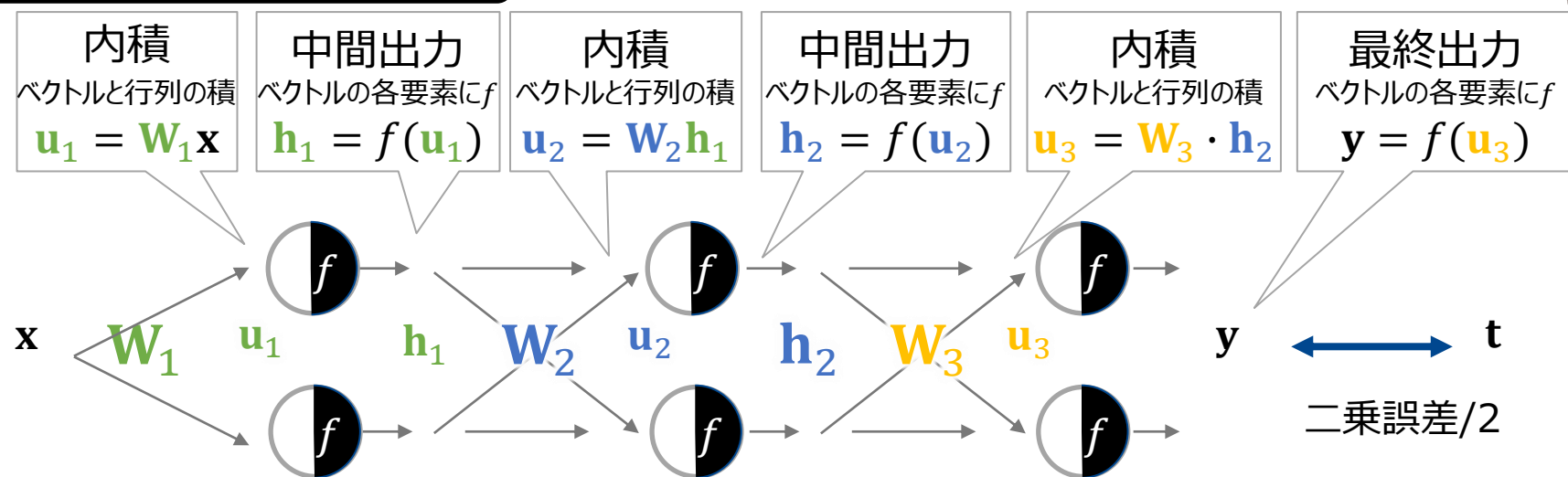
各層のニューロンが複数になった場合

ついでに入出力&教師もベクトル x, y, t に

先ほどまで



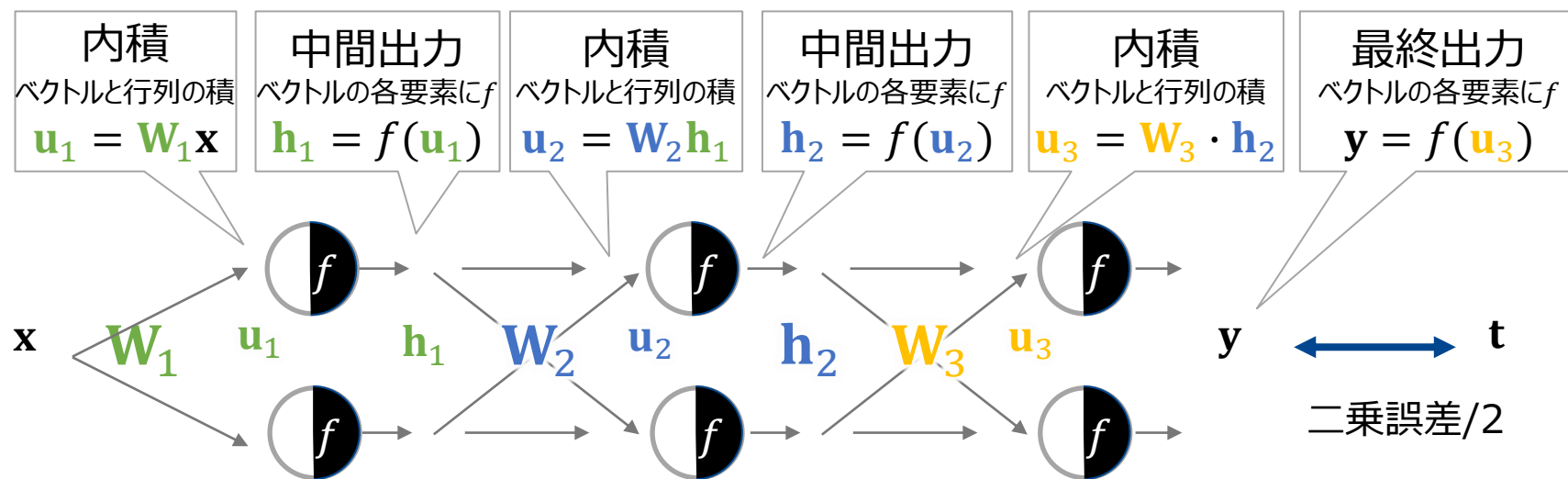
各層のニューロンを複数に



なんかあまり違わない?



式は複雑になるけど



本質的には同じ原理で
バックプロパゲーションによる学習が可能！

$$\mathbf{x} = \mathbf{h}_0 \rightarrow \mathbf{h}_1 \rightarrow \mathbf{h}_2 \rightarrow \cdots \rightarrow \mathbf{h}_L = \mathbf{y}$$

順伝播

$$\Delta_L \rightarrow \Delta_{L-1} \rightarrow \cdots \rightarrow \Delta_2$$

逆伝搬

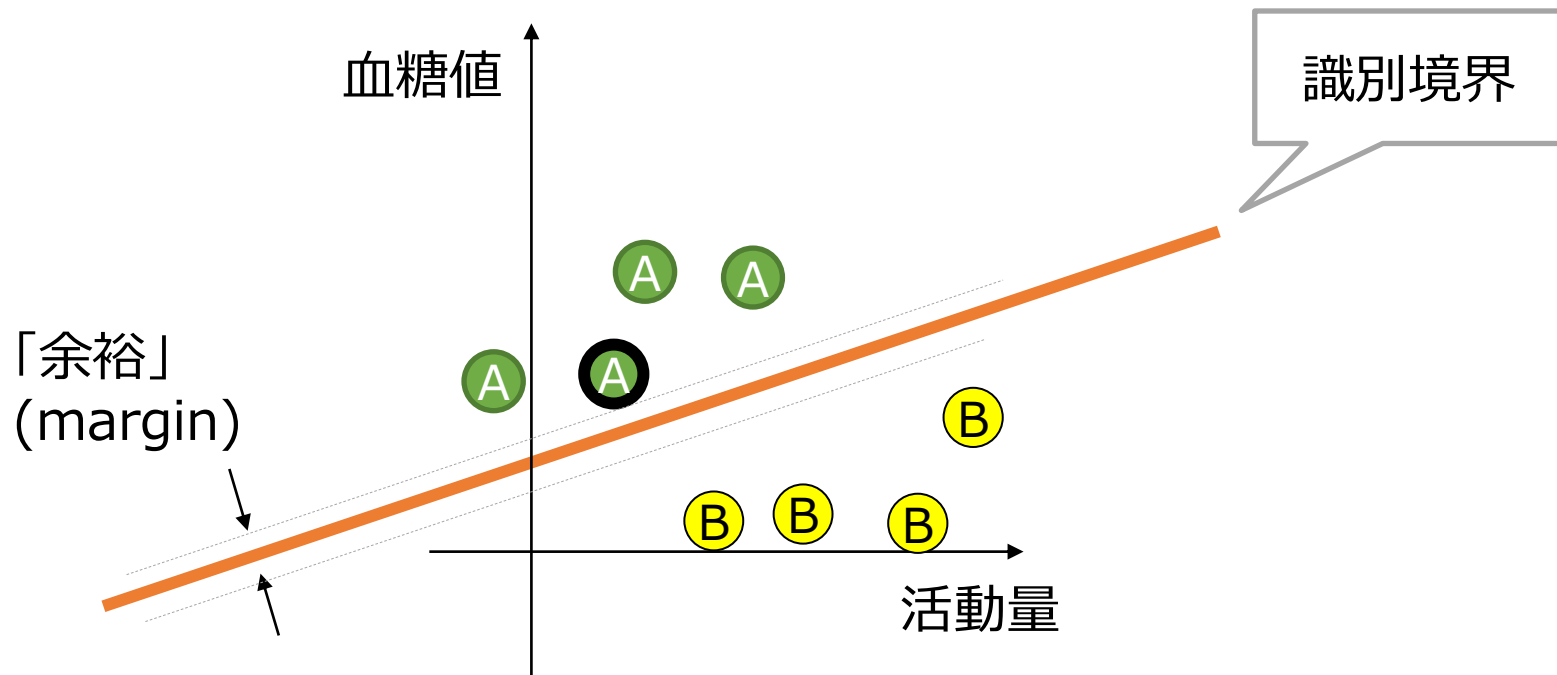
$$\nabla_{\mathbf{W}_L} J \rightarrow \nabla_{\mathbf{W}_{L-1}} J \rightarrow \cdots \rightarrow \nabla_{\mathbf{W}_1} J$$

【付録】 サポートベクトルマシン Support Vector Machines (SVM)

識別関数を決定する他の方法

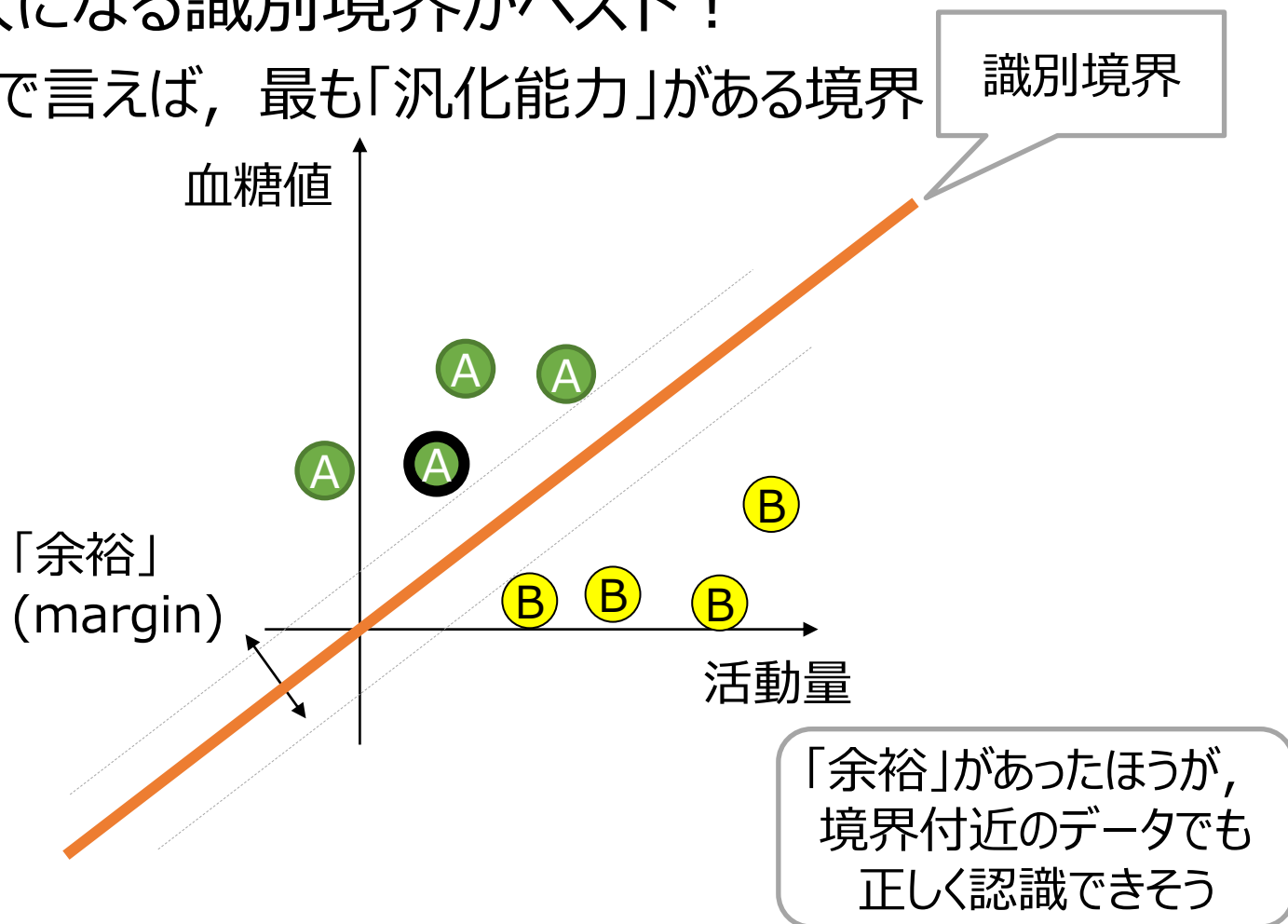
SVMの考え方 (1/2)

- 学習パターンは正しく識別できているが、ちょっとしたずれたら誤分類発生？

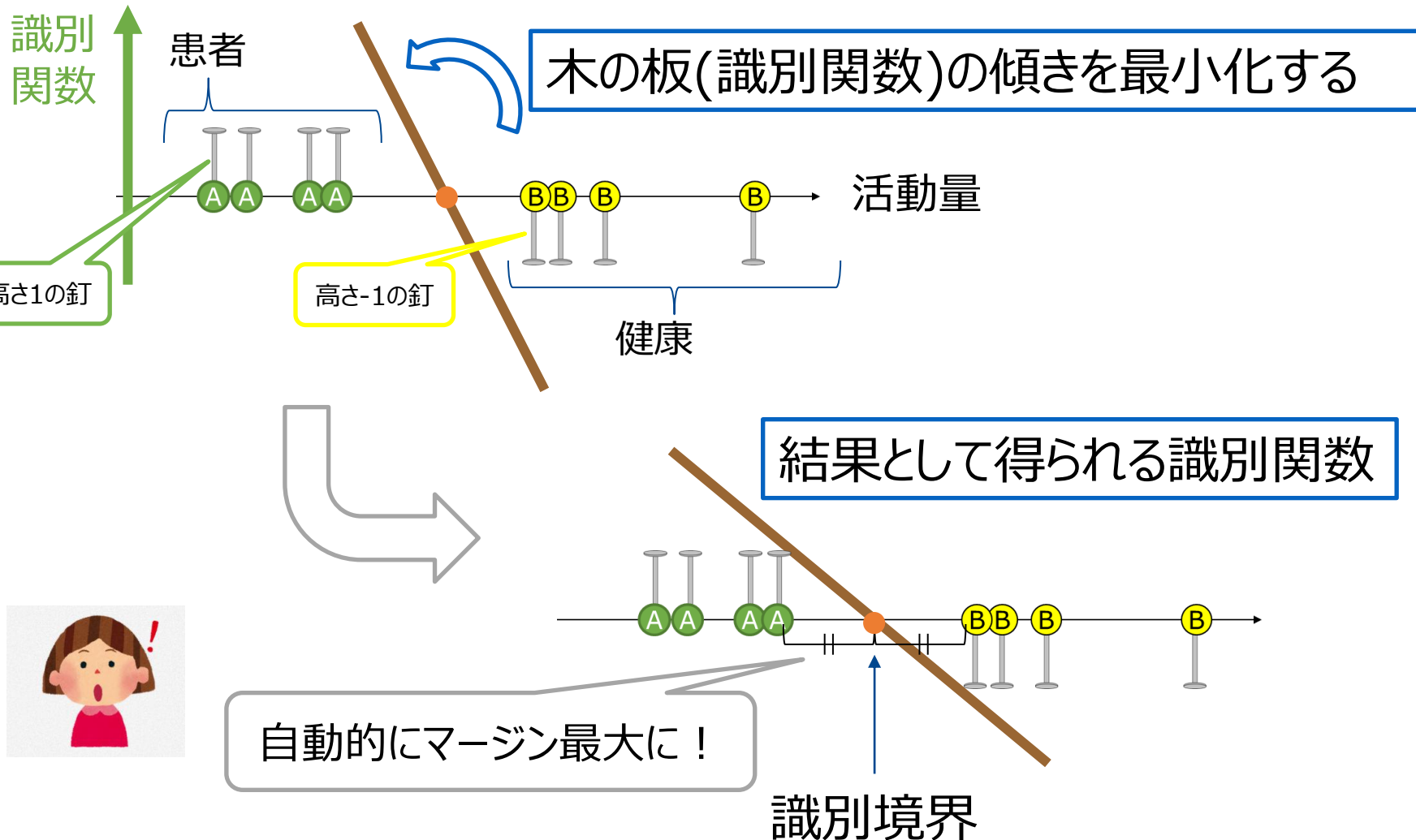


SVMの考え方 (2/2)

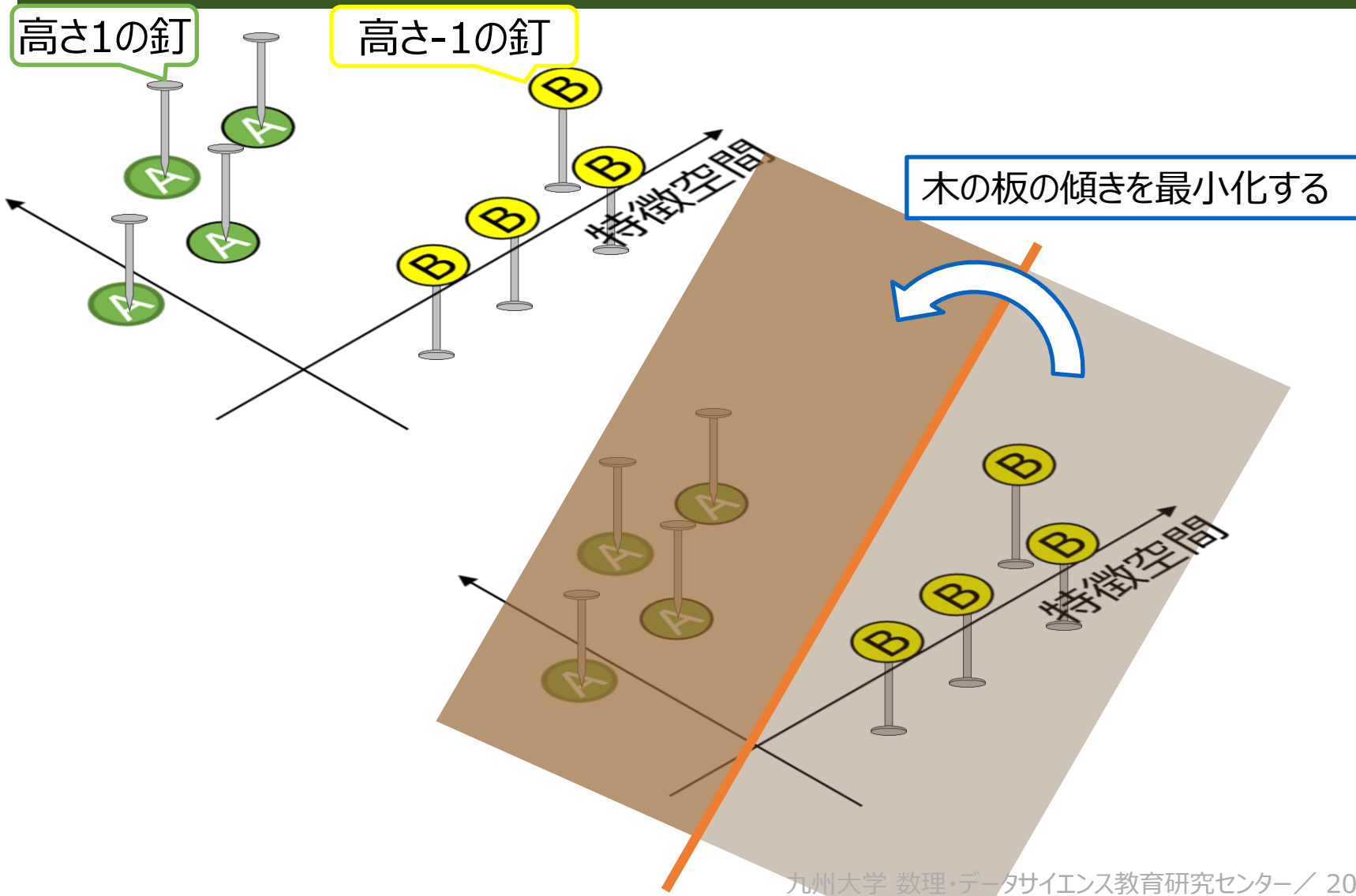
- 余裕が最大になる識別境界がベスト！
 - 難しい言葉で言えば，最も「汎化能力」がある境界



マージン最大化はどうやって？ 1次元の場合

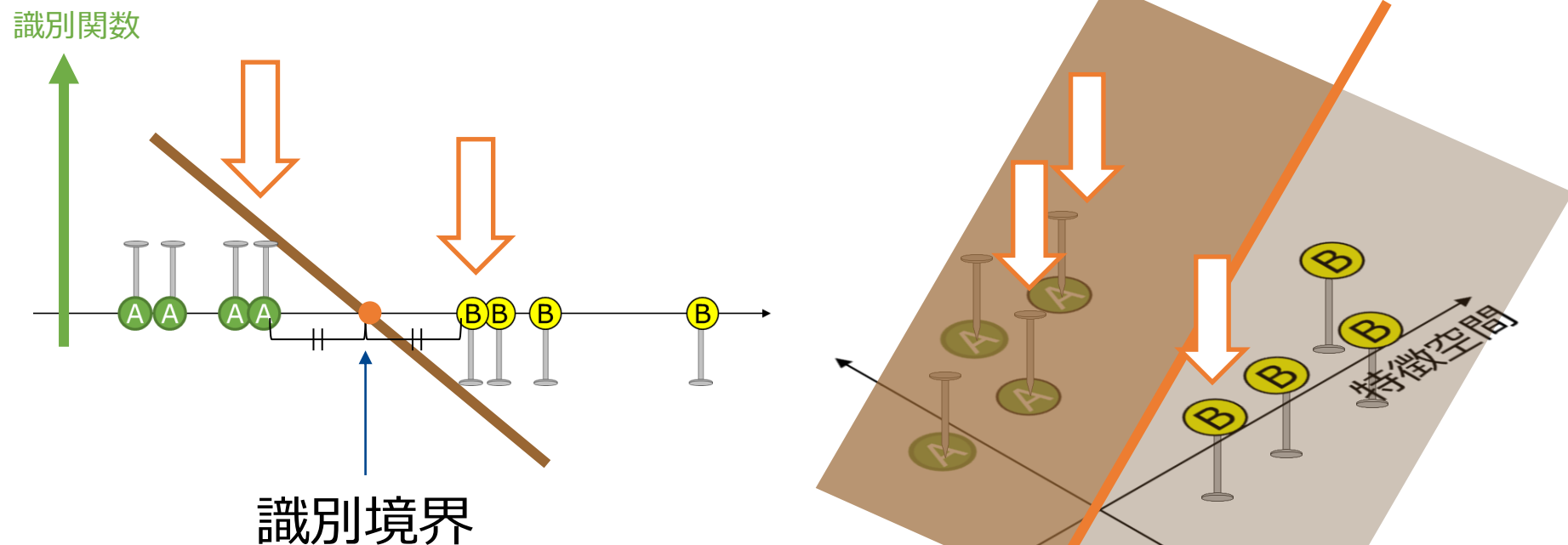


マージン最大化はどうやって？ 2次元の場合



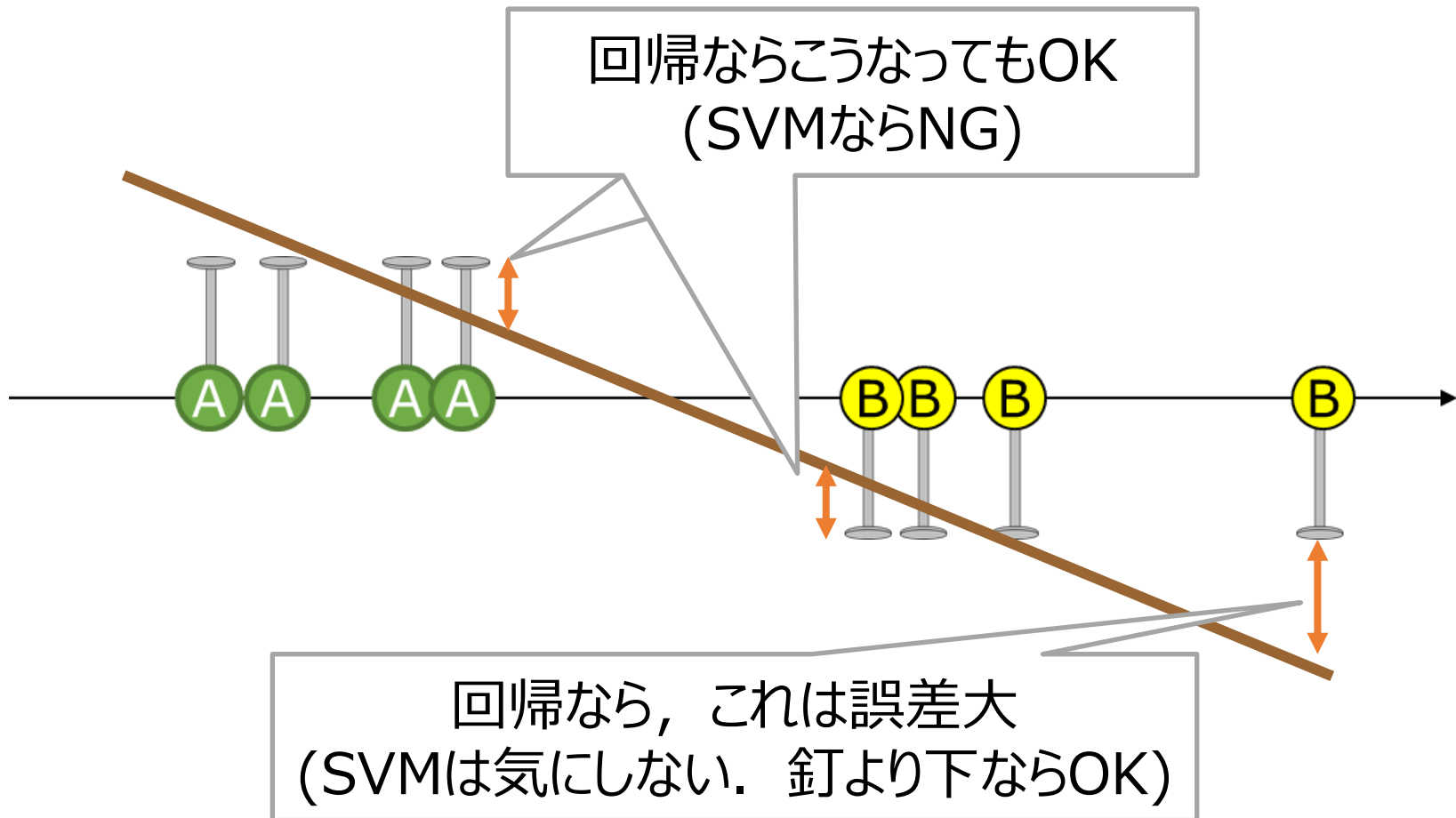
サポートベクトル？

- 識別関数を決めているデータのこと = 木の板が当たっているデータ

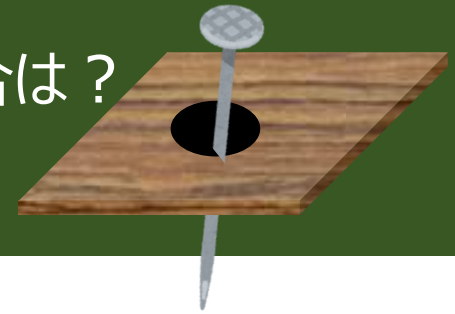


- ポイント：たくさんデータがあっても，クラス境界付近の少数のデータだけで識別関数が決まる！

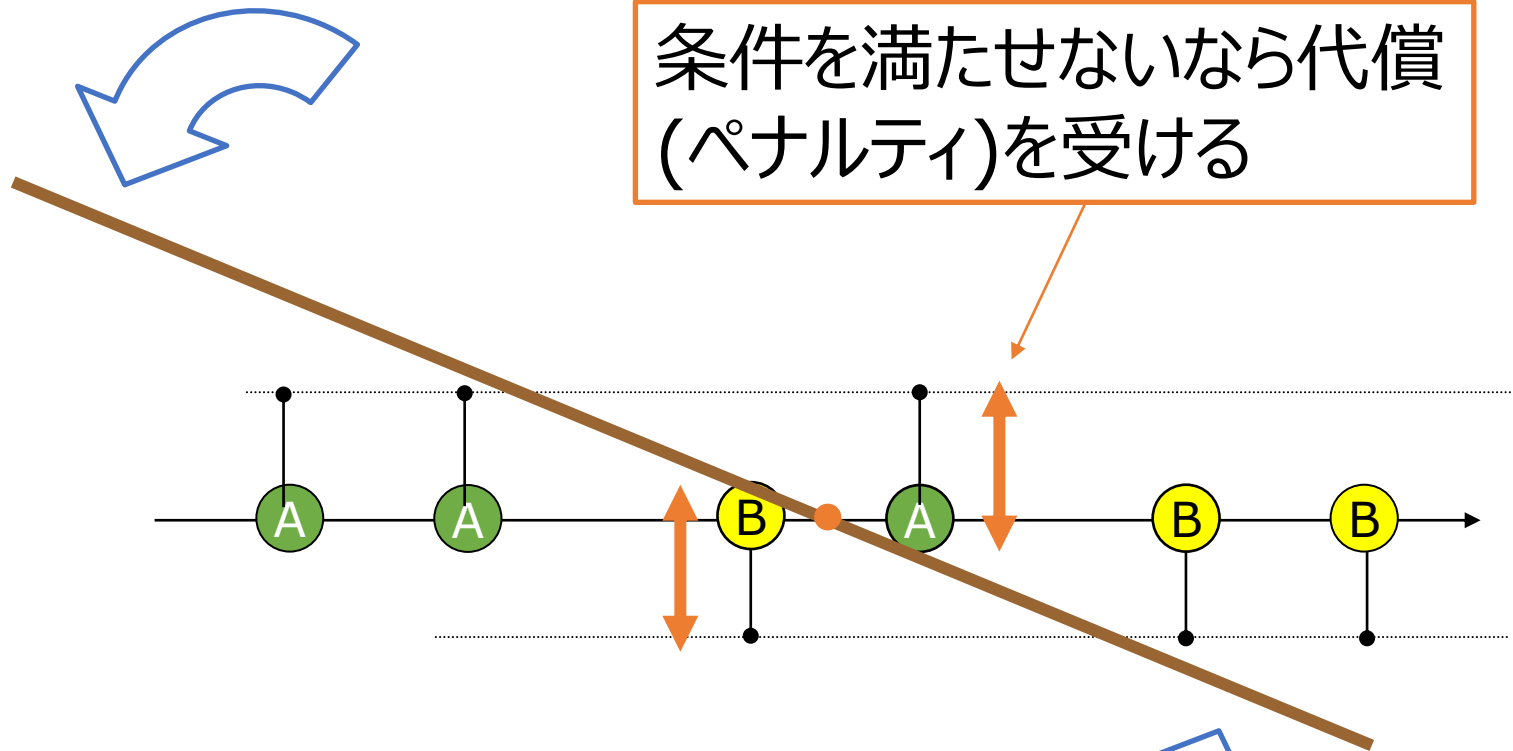
参考：回帰分析と似てるがちょっと違う



さらに参考：きれいに分けられない場合は？
→ソフトマージンSVM



条件を満たせないなら代償
(ペナルティ)を受ける



条件を極力満たしながら、
ペナルティも最小化する