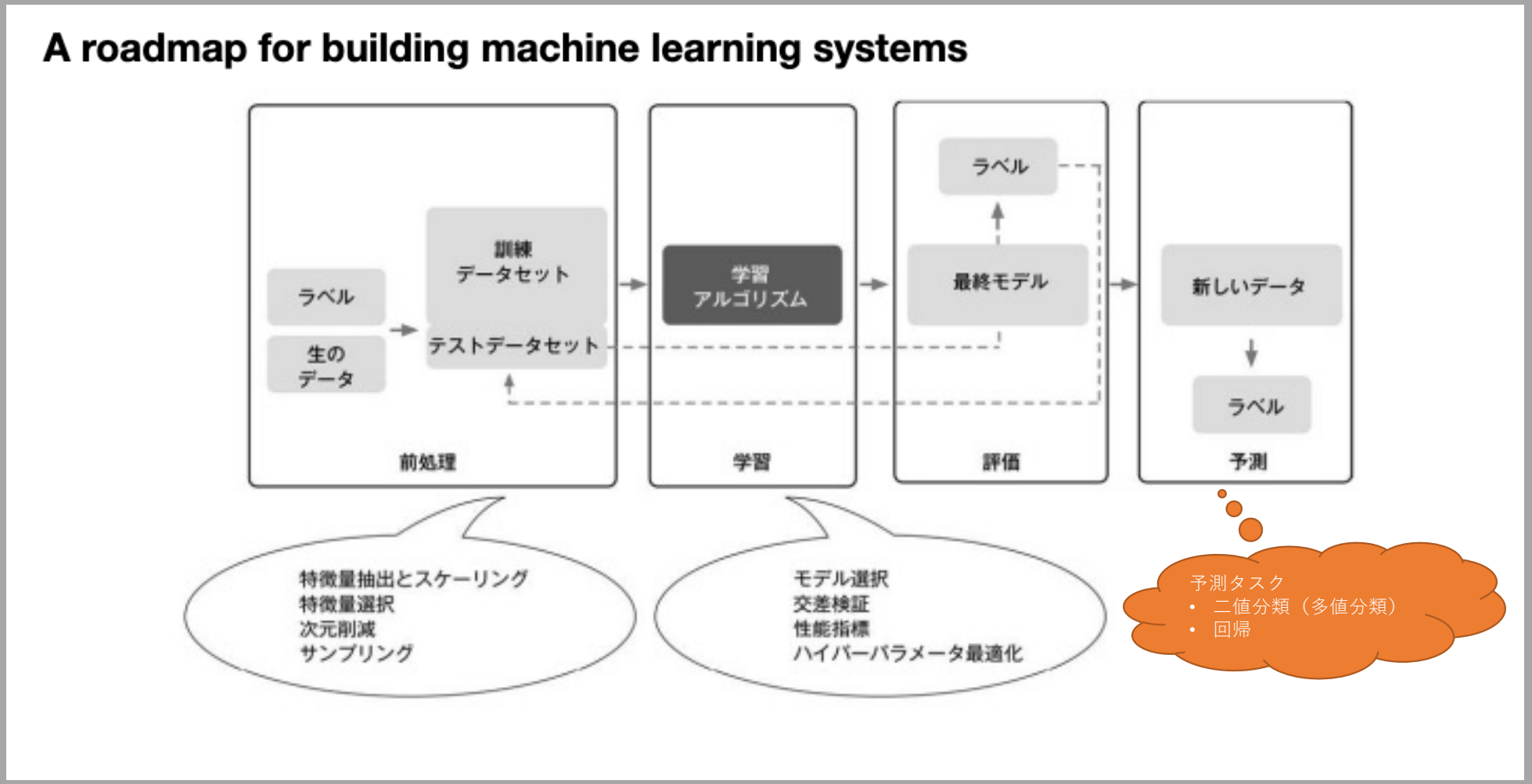
Githubよりテキストのサンプルコードをダウンロードしてください

https://github.com/rasbt/python-machine-learning-book

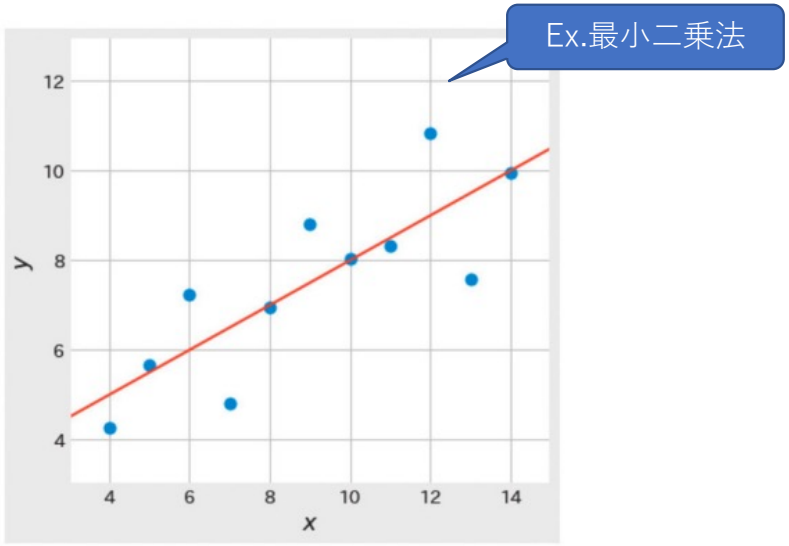
第2章 分類問題
ー機械学習アルゴリズムのトレーニングー

- 分類問題
- パーセプトロンとADALINE(Adaptive Linear Neuron)
  - **機械学習のアルゴリズムに対する直感を養う（3rd で理解するに改定された）**
    - **講義で**
  - Pandas, NumPy, matplotlib を使ってデータの読み込み、処理、可視化を行う
    - 自分で
  - 線形分類のアルゴリズムをPythonで実装する
    - 自分で

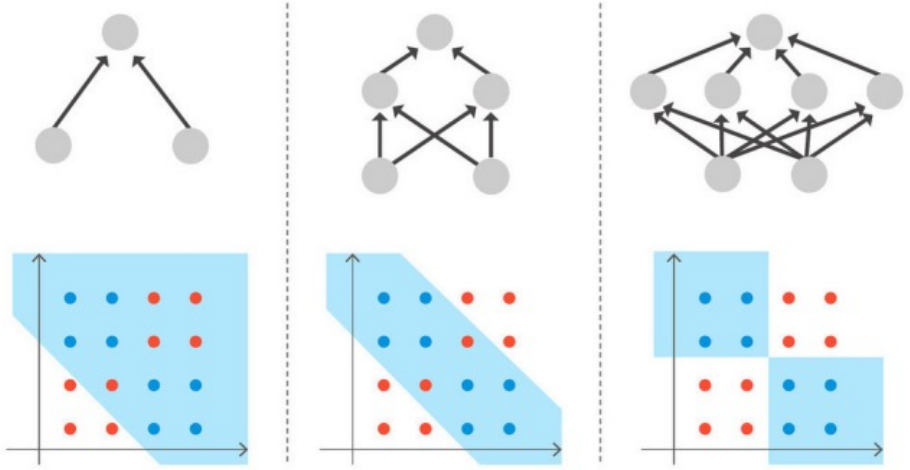
1章最後の図について、


A roadmap for building machine learning systems
ラベル
生の
データ
訓練
データセット
テストデータセット
前処理
学習
アルゴリズム
学習
ラベル
最終モデル
評価
新しいデータ
ラベル
予測
特徴量抽出とスケーリング
特徴量選択
次元削減
サンプリング
モデル選択
交差検証
性能指標
ハイパーパラメータ最適化
予測タスク
• 二値分類（多値分類）
• 回帰

回帰と分類の直感的違い



青の観測点を
オレンジの直線で近似



青とオレンジのデータを境界で分離

最小二乗法

最小二乗法（Least Squares Method）は、統計学や機械学習の分野でよく用いられる回帰分析の手法の一つで、観測値と予測値の差を二乗した値の総和を最小化することで、回帰直線や曲線を求める手法です。

最小二乗法は、以下のような回帰モデルに対して適用されます。
$y = \beta 0 + \beta 1x1 + \beta 2x2 + ... + \beta pxp + \varepsilon$

この式において、yは目的変数、x1, x2, ..., xpは説明変数、β0, β1, β2, ..., βpは回帰係数、εは誤差項を表します。最小二乗法では、誤差項の二乗和を最小化するように回帰係数を求めます。

最小二乗法によって求められた回帰直線や曲線は、観測値と予測値の差を最小化するため、データポイントに最も近い直線や曲線になります。最小二乗法は、単回帰分析（1つの説明変数を用いた回帰分析）や重回帰分析（複数の説明変数を用いた回帰分析）など、様々な回帰モデルに対して適用されます。

最小二乗法は、回帰分析の中でも基本的な手法の一つであり、簡単に実装できることから、広く使われています。ただし、誤差項が正規分布に従う場合に最も効果的であるとされており、異常値や外れ値がある場合には正確な予測を行うことができない可能性があるため、注意が必要です。

分類問題

例えば、

- 商品レビューについて
  - そのレビューは**肯定**(的)か、**否定**(的)か？
- eメールについて
  - そのメールは**スパム**か、**否**か？
- ○○について
  - その○○は△△か、□□か？

これらは、**二値分類問題**　として定式化して解くことができる。

分類問題

図 4.5: 認識結果一例

例えば、

- 表情画像について
  - その画像は**笑顔**か、**怒り**か、**悲しみ**か、**嫌悪**か、、、？
- MNISTのデータについて
  - そのデータは**0**か、**1**か、**2**か、、、、、、、？
- ○○について
  - その○○は△△か、××か、、、、、、、？

これらは、**多値分類問題**

（多値分類問題は二値分類の方法を拡張して解決することが多い）

分類問題

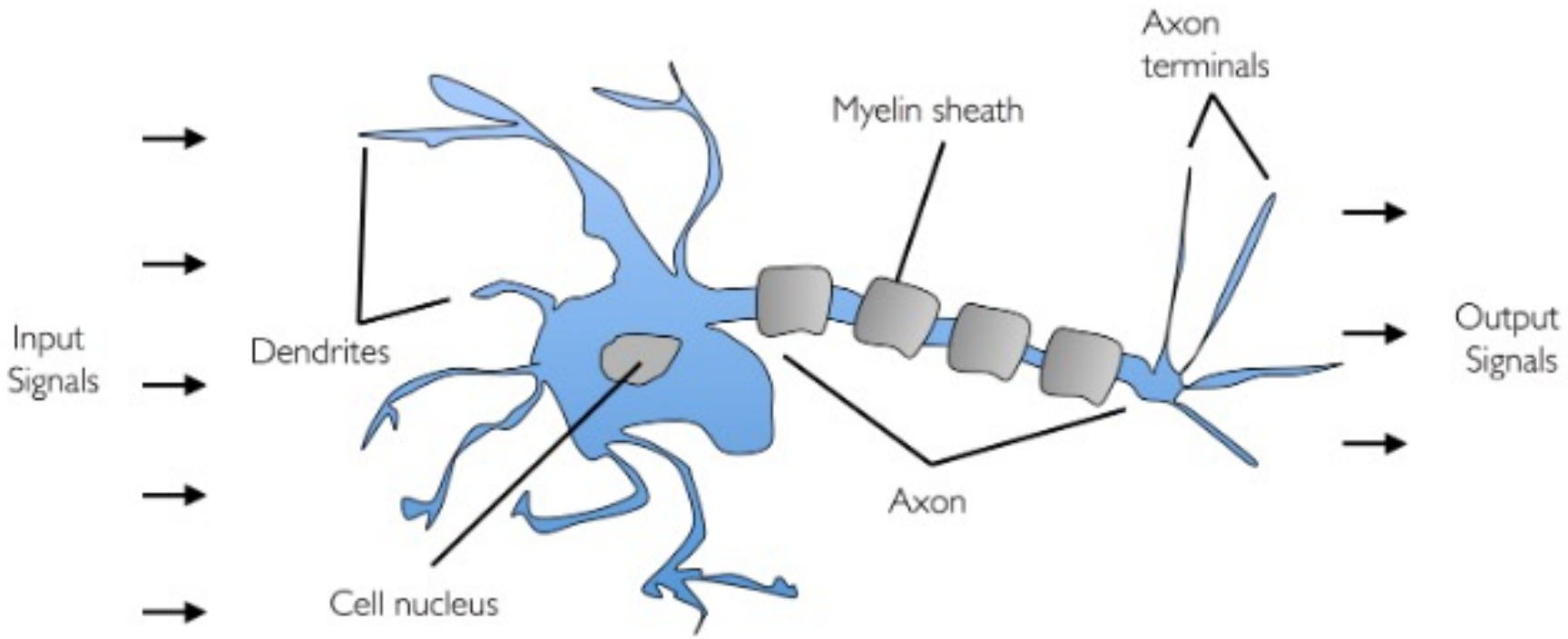
として定式化される問題は

無限

に存在する

2.1 人工ニューロン
―機械学習の前史―

計算可能なモデルとして記述したい



要素還元主義は
知能または機械学習を再現できるか？

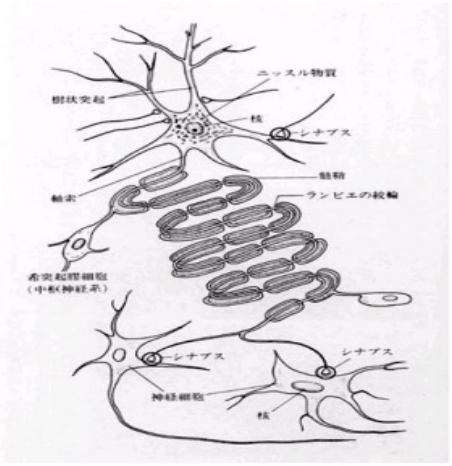
要素還元主義とは、ある複雑な現象やシステムを、最も基本的な要素や部品に分解し、その要素や部品の性質や相互作用を分析することで、全体の理解を深めようとする考え方です。

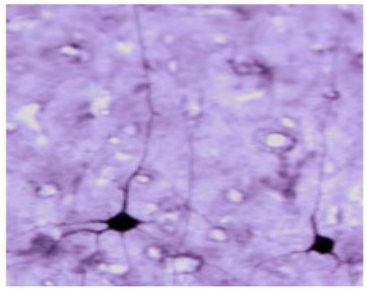
この考え方は、自然科学や工学分野において広く使われており、例えば化学では物質を原子や分子に分解して解析することで、物質の性質や反応を理解しようとします。また、生物学では細胞や遺伝子などの要素に分解して、生物の構造や機能を解明しようとします。

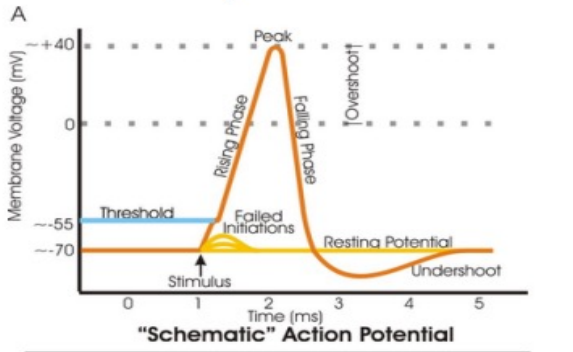
要素還元主義は、一方で全体を理解するための視点を欠いているという批判もあります。例えば、社会科学や人文科学の分野では、人間や社会現象を要素に分解することが困難であるため、要素還元主義的なアプローチが適用しづらい場合があります。

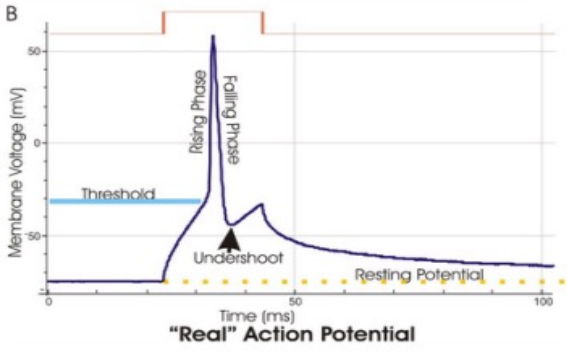
2.1 人工ニューロン
―機械学習の前史―

神経細胞(ニューロン)









・神経細胞の電気的性質
膜電位（細胞内部と外部の電位差）:-40mV～-90mV
刺激により細胞内電位が数mV正の方向へ移動することをきっかけに，あるレベルで自己再生的に細胞内電位が上昇し，再び元の状態に戻る(振幅100mV,時間幅1msの神経インパルス).
・約140億の神経細胞のネットワーク・・・超並列
・シナプスの可塑性・・・あるパターンのインパルス入力後，シナプスの伝達効率が変化する．.

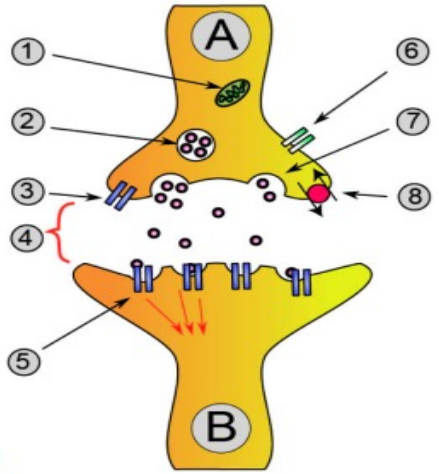
Wikipedia活動電位より

2.1 人工ニューロン
―機械学習の前史―

神経細胞の可塑性

1)シナプス伝達の可塑性
神経細胞と神経細胞の接続部であるシナプスの間隙で神経伝達する神経伝達物質の量を変化

2)シナプス結合の可塑性
神経細胞の形態の変化
損傷を受けたシナプスに変わって新しいシナプスが作られ数、位置が変化



シナプス伝達
細胞Aのインパルスが(2)の受容体を刺激し、神経伝達物質が放出される。放出された神経伝達物質は細胞Bの受容体と結合し、Bのイオンチャンネルが開く。Bの細胞内外電位差が変化し、電位信号が伝わる。

シナプス前細胞とシナプス後細胞がともに高頻度で連続発火すると伝達効率が増加。
シナプスの伝達効率が変化するシナプス可塑性　＝　記憶や学習

McCulloch & Pitts モデル

形式ニューロン

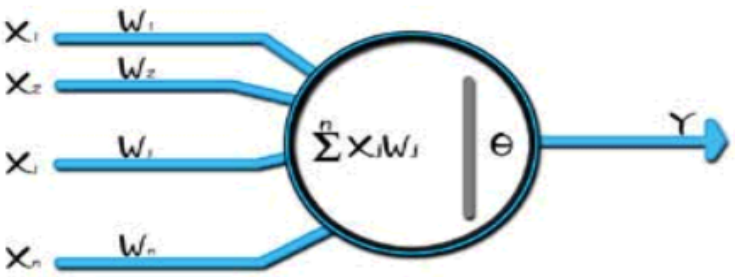
MCPニューロンは二値出力を行う単純な論理ゲートとして神経細胞を表現した！

McCulloch & Pitts model ・・・神経論理回路の提案

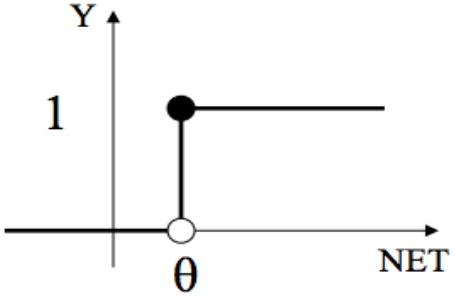
(論文：A logical calculus of the ideas immanent in nervous activity(1943))

・「入力(x)」が入り、重要さの度合いによって「重み(w)」倍される.

・それらの総和が「しきい値(θ)」をこえたとき、「出力(y)」を発生する.



活性化関数



活動電位のモデル化

$$Y\left(\sum_{i=1}^{N} w_i x_i - \theta\right)$$

$$Y(x) = \begin{cases} 0 (x \leq 0) \\ 1 (x > 0) \end{cases}$$

McCulloch & Pitts モデル（本書では）

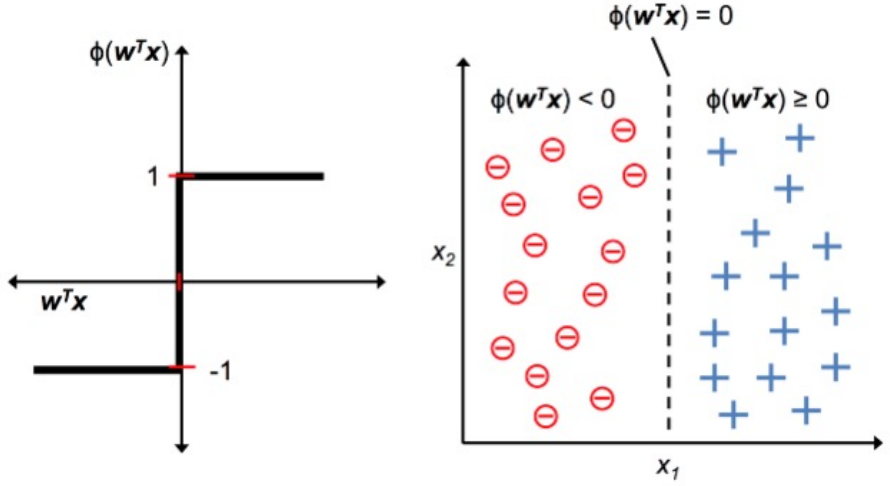
$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad (2.1.1)$$

$$\phi(z) = \begin{cases} 1 & (z \geq \theta) \\ -1 & (z < \theta) \end{cases} \quad (2.1.2)$$

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

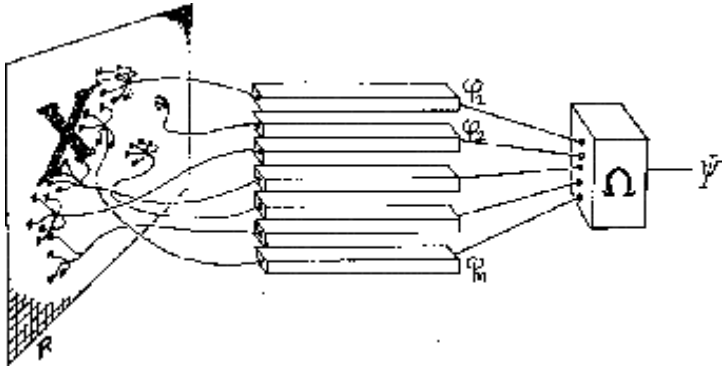
$$\text{および} \quad \phi(z) = \begin{cases} 1 & (z \geq 0) \\ -1 & (z < 0) \end{cases} \quad (2.1.3)$$

MCPニューロンは二値出力を行う単純な論理ゲートとして神経細胞を表現した！

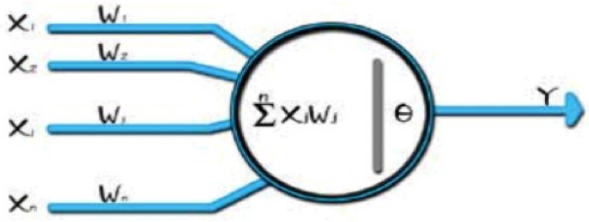


2.1.2 パーセプトロンの学習則

Rosenblatt(1962)



Step1.与えられた入力 ***x*** に対して出力 ***y*** を計算する.
Step2.
　2-1：もし出力が正しければStep1に進む.
　2-2：もし出力が間違えていて，その値が0ならば,
　　各重みにそれに対応する入力を加える.
　2-3：もし出力が間違えていて，その値が1ならば,
　　各重みにそれに対応する入力を引く.
Step3.Step1に進む.

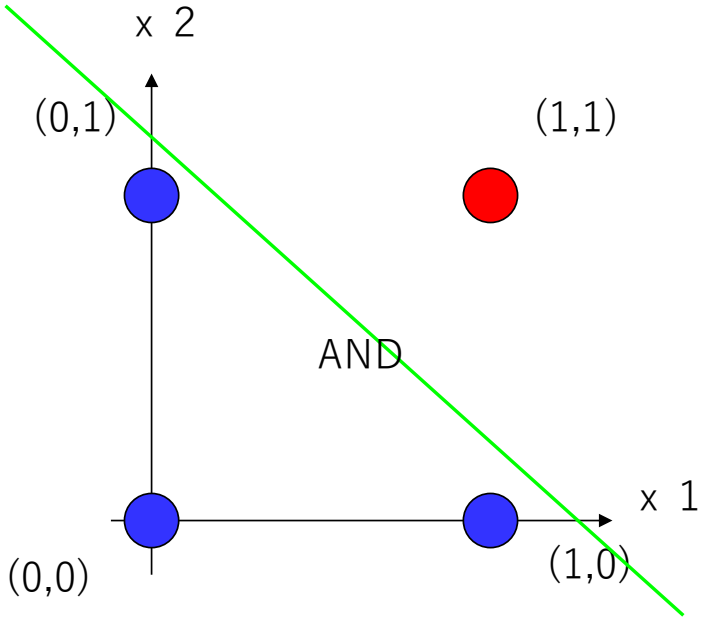


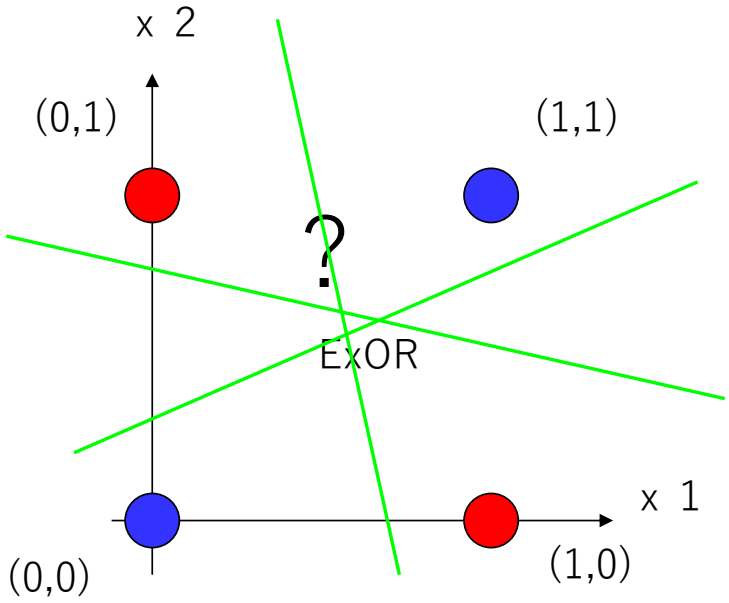
Rosenblatt のパーセプトロン学習は、教師あり学習で分類問題を解く予測モデルである。

線形分離可能性

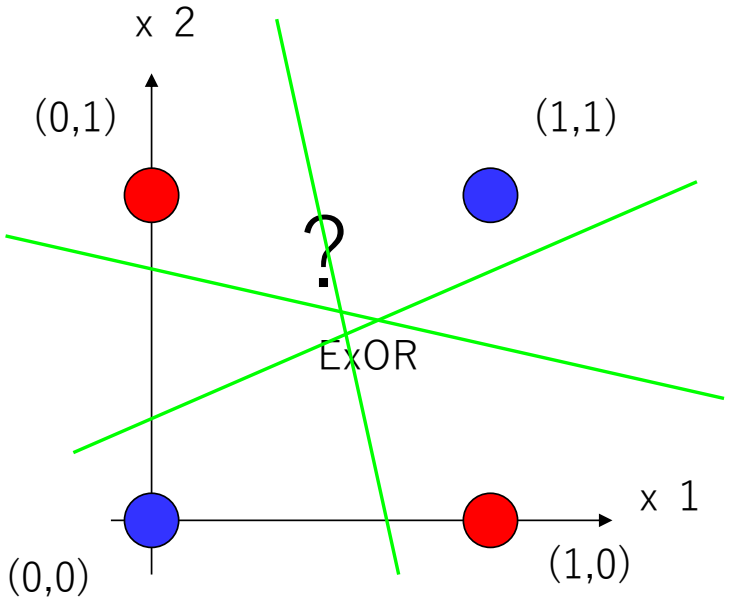
$$\theta = x_1 \cdot w_1 + x_2 \cdot w_2$$

$$x_2 = -(w_1 / w_2)\, x_1 + (\theta / w_2)$$



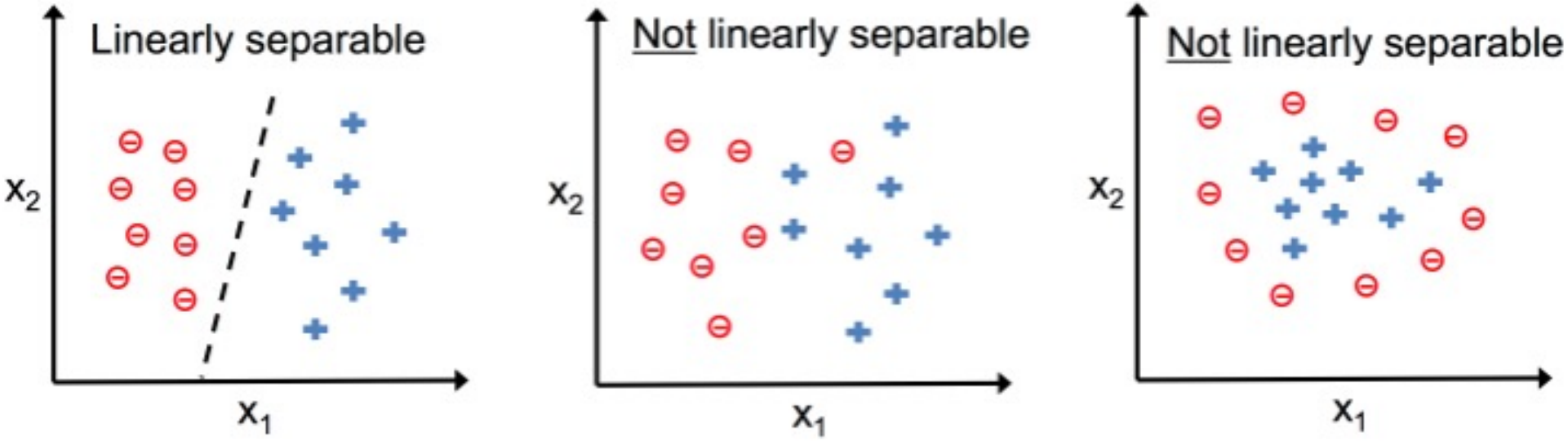


線形分離可能性

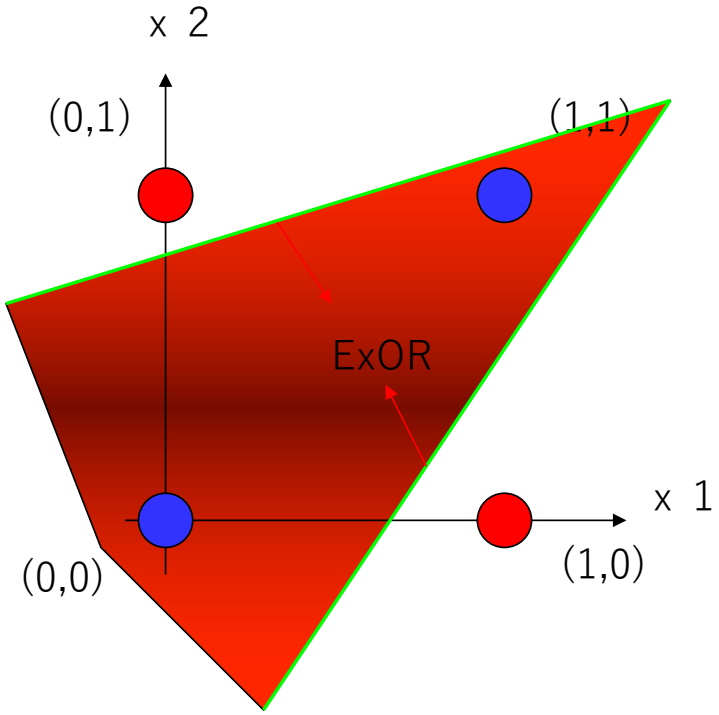


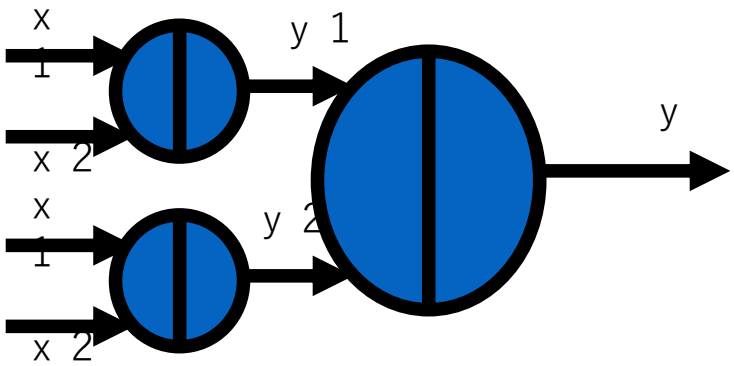
n	$2^{2\uparrow n}$	線形分離可能関数の数
1	4	4
2	16	14
3	256	104
4	65536	1882
5	$4.3\times10\uparrow9$	94572

テキストの線型分離可能性の記述



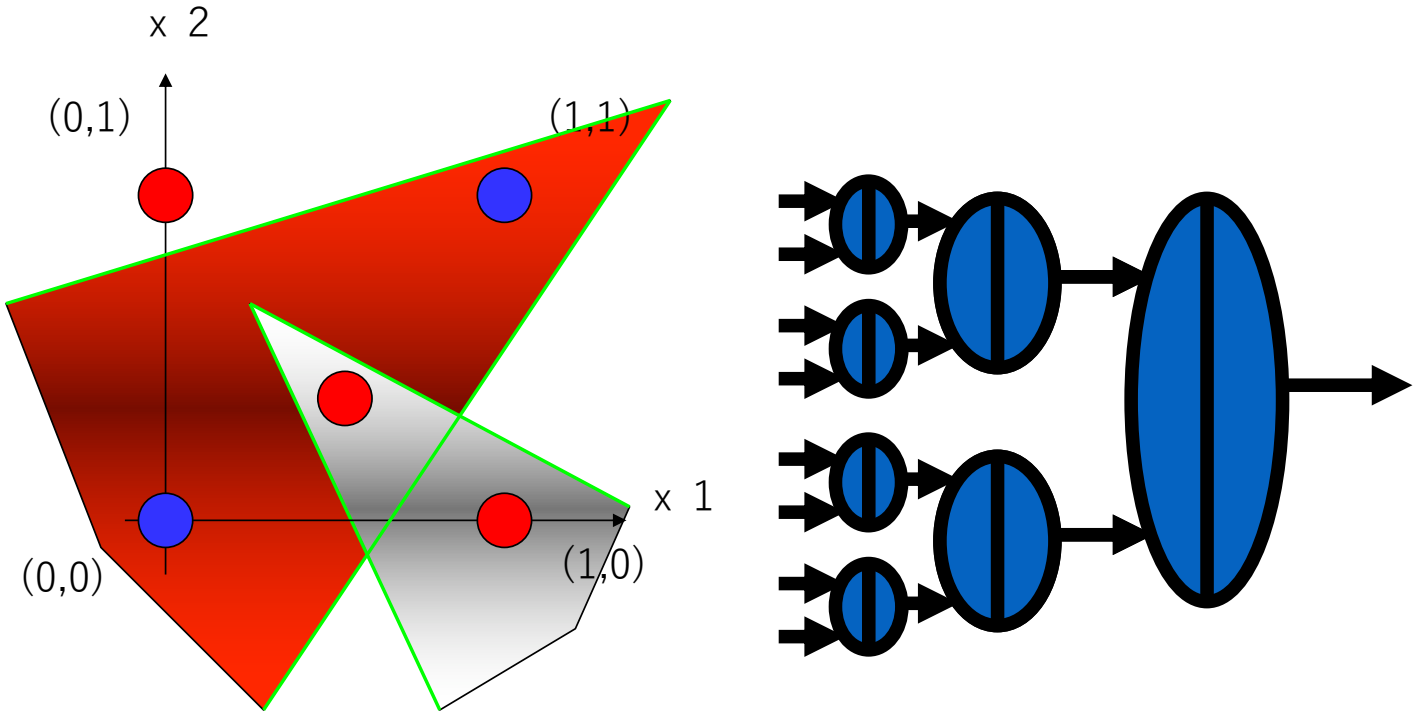
線形分離可能性の克服





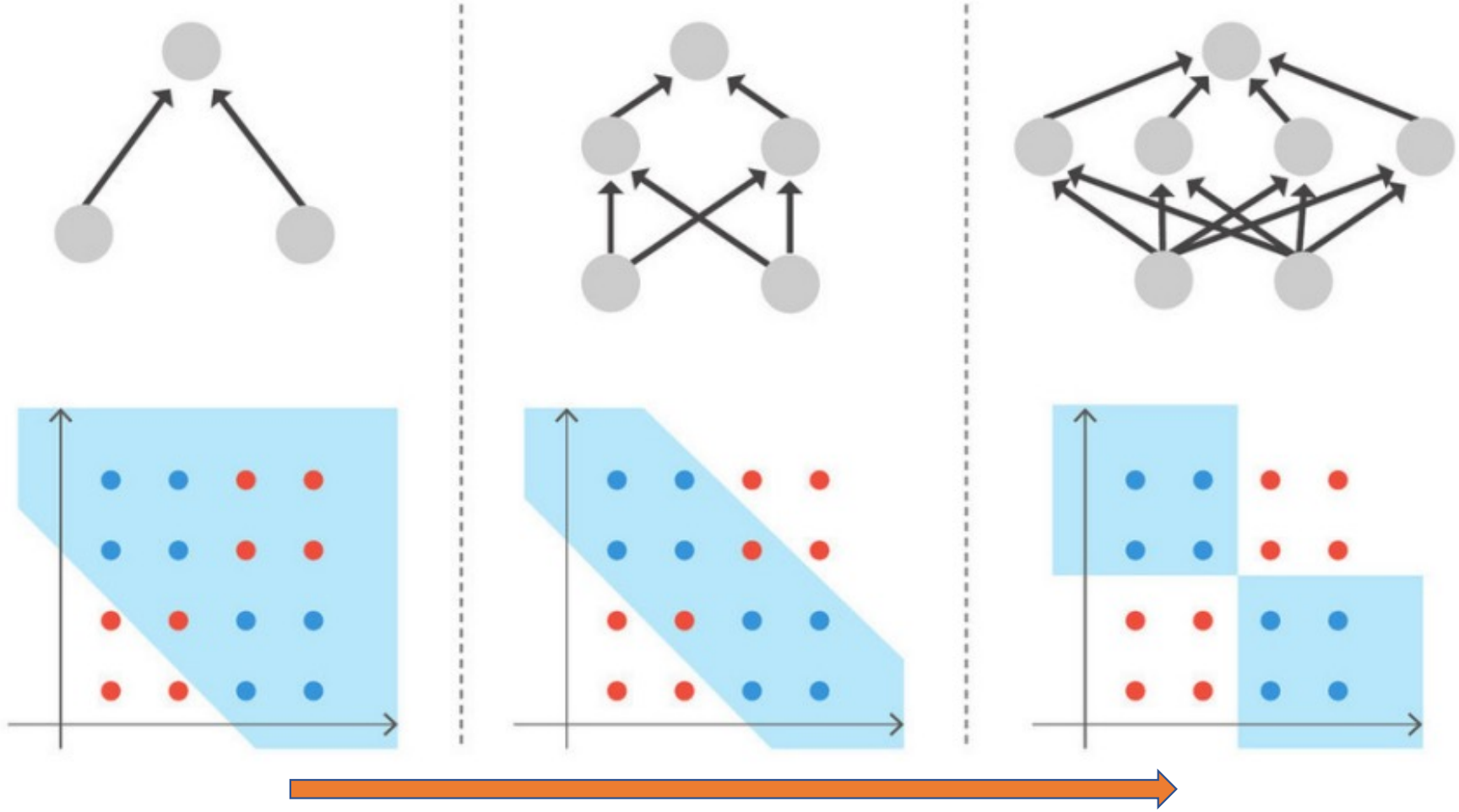
凸領域の分離が可能になる

線形分離可能性の克服

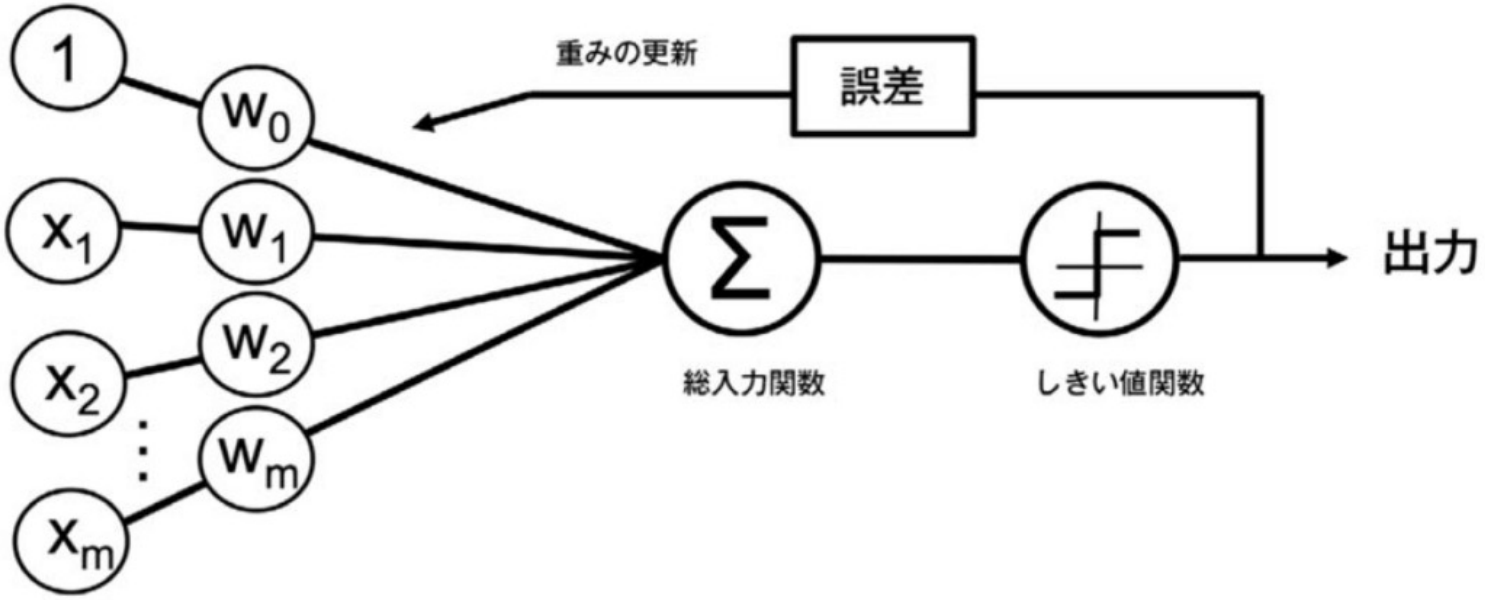


凹領域の分離が可能になる.
3層によって複雑な分離超平面の構成が出来る.

3階層にすること、隠れ層のニューロン数を増やすことで複雑な分離を可能にする。



復習：パーセプトロンの実装



誤差がなくなるまでW_iを調整する　ー>　モデルの学習

実装

```
import numpy as np
class Perceptron(object):
    """パーセプトロンの分類器

    パラメータ
    ------------
    eta : float
        学習率（0.0 より大きく 1.0 以下の値）
    n_iter : int
        トレーニングデータのトレーニング回数
    random_state : int
       重みを初期化するための乱数シード

    属性
    -----------
    w_ : 1 次元配列
        適合後の重み
    errors_ : リスト
        各エポックでの誤分類（更新）の数

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

実装

```
    def fit(self, X, y):
        """ トレーニングデータに適合させる

        パラメータ
        ----------
        X : { 配列のようなデータ構造 }, shape = [n_samples, n_features]
            トレーニングデータ
            n_samples はサンプルの個数、n_features は特徴量の個数
        y : 配列のようなデータ構造 , shape = [n_samples]
            目的変数

        戻り値
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):  # トレーニング回数分トレーニングデータを反復
            errors = 0
            for xi, target in zip(X, y):  # 各サンプルで重みを更新
                # 重み w_1, ... , w_m の更新
                # Δw_j = η( y^(i) - ŷ^(i) ) x_j^(i) (j = 1, .. , m)
                update = self.eta * (target - self.predict(xi))
```

実装

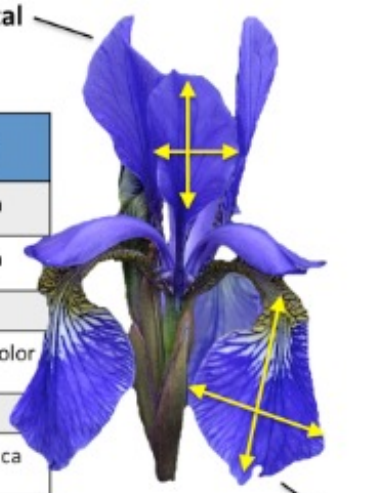
```
                self.w_[1:] += update * xi
                # 重み w_0 の更新：Δw_0 = η(y^(i) - ŷ^(i))
                self.w_[0] += update
                # 重みの更新が 0 でない場合は誤分類としてカウント
                errors += int(update != 0.0)
            # 反復回数ごとの誤差を格納
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """総入力を計算"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """1ステップ後のクラスラベルを返す"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

1.3 基本用語と表記法

Iris dataset

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

2値分類モデルを使って多値分類問題を解く？

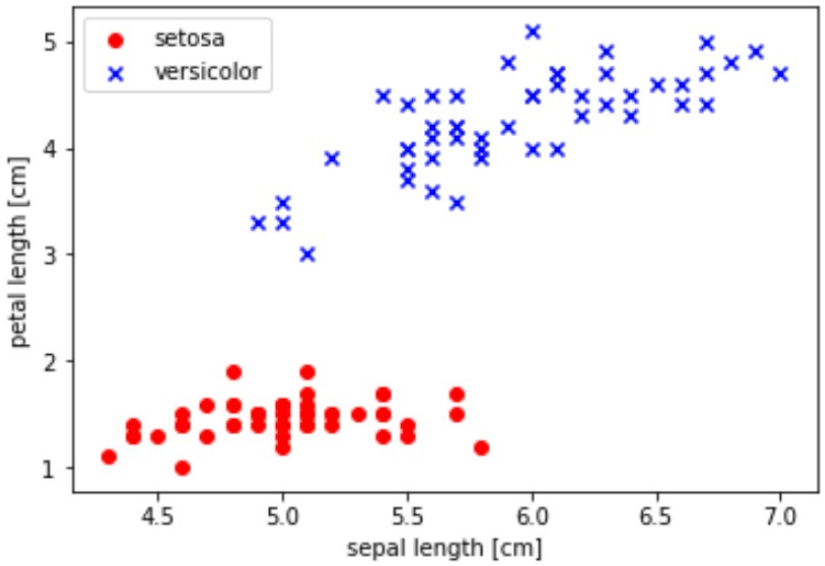
本日のミニ課題

二値分類器が与えられたとして、多値分類の問題を解く擬似コードを示してください。

Iris dataset の読み込み

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # 1-100 行目の目的変数の抽出
>>> y = df.iloc[0:100, 4].values
>>> # Iris-setosa を -1, Iris-virginica を 1 に変換
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> # 1-100 行目の 1、3 列目の抽出
>>> X = df.iloc[0:100, [0, 2]].values
>>> # 品種 setosa のプロット（赤の○）
>>> plt.scatter(X[:50,0], X[:50,1], color='red', marker='o', label='setosa')
>>> # 品種 versicolor のプロット（青の ×）
>>> plt.scatter(X[50:100,0], X[50:100,1], color='blue', marker='x', label='versicolor')
>>> # 軸のラベルの設定
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> # 凡例の設定（左上に配置）
>>> plt.legend(loc='upper left')
>>> # 図の表示
>>> plt.show()
```

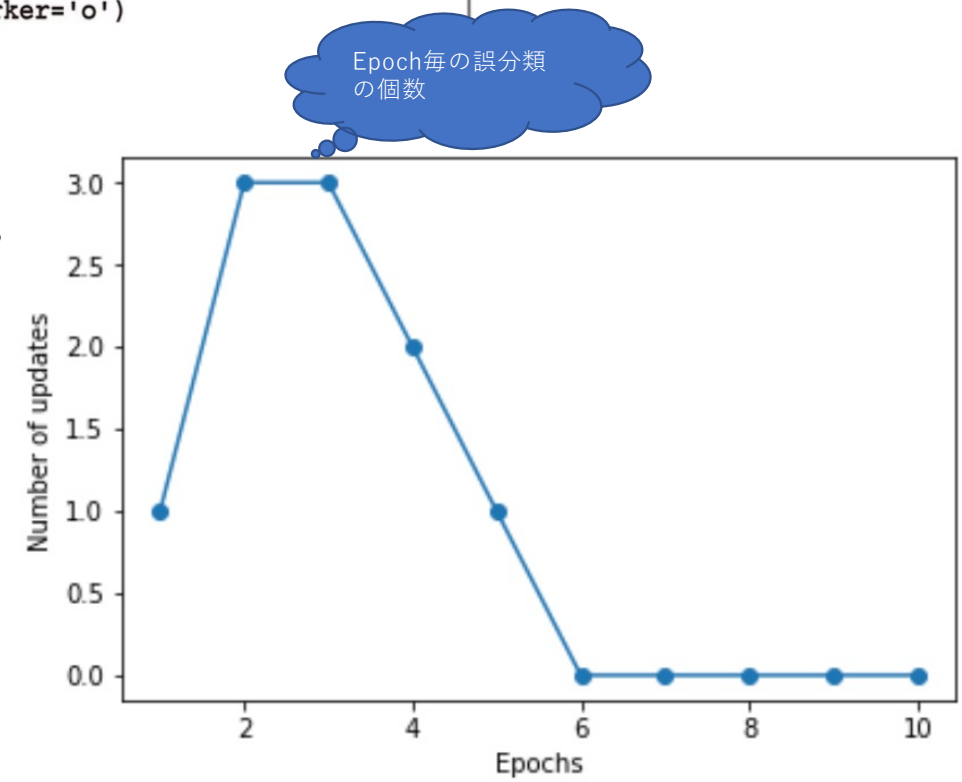
二属性なら平面、三属性なら空間、それ以上なら多次元空間に配置されたデータ群に対して、ラベルを正しく分割する超平面を獲得したい！



4つの属性から花びらの長さ＆がくの長さを選び２次元平面に配置した。

Iris dataset の学習

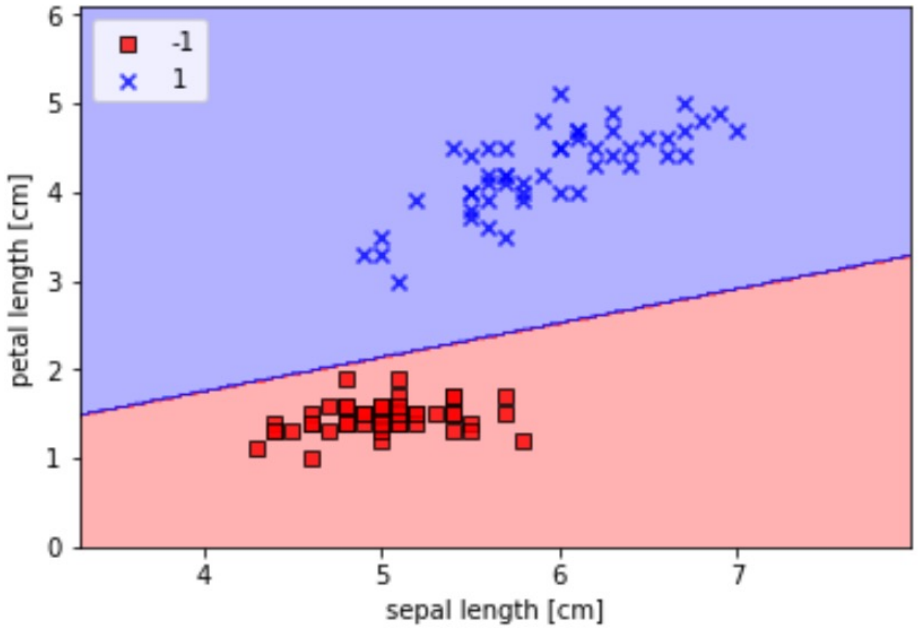
```
>>> # パーセプトロンのオブジェクトの生成（インスタンス化）
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> # トレーニングデータへのモデルの適合
>>> ppn.fit(X, y)
>>> # エポックと誤分類誤差の関係の折れ線グラフをプロット
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
>>> # 軸のラベルの設定
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of update')
>>> # 図の表示
>>> plt.show()
```



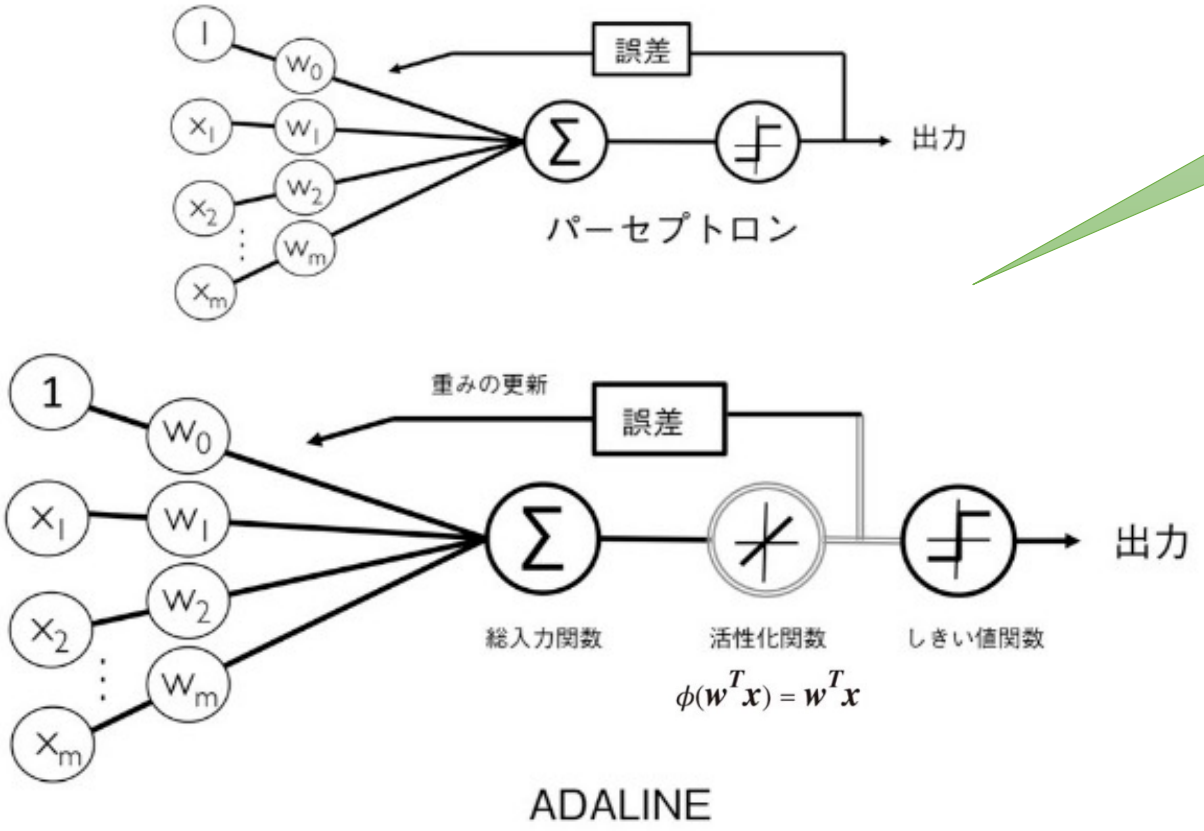
Iris dataset の学習結果可視化

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # マーカーとカラーマップの準備
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # 決定領域のプロット
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    # グリッドポイントの生成
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    # 各特徴量を 1 次元配列に変換して予測を実行
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    # 予測結果を元のグリッドポイントのデータサイズに変換
    Z = Z.reshape(xx1.shape)
    # グリッドポイントの等高線のプロット
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    # 軸の範囲の設定
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # クラスごとにサンプルをプロット
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')
```

```
>>> # 決定領域のプロット
>>> plot_decision_regions(X, y, classifier=ppn)
>>> # 軸のラベルの設定
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> # 凡例の設定 （左上に配置）
>>> plt.legend(loc='upper left')
>>> # 図の表示
>>> plt.show()
```



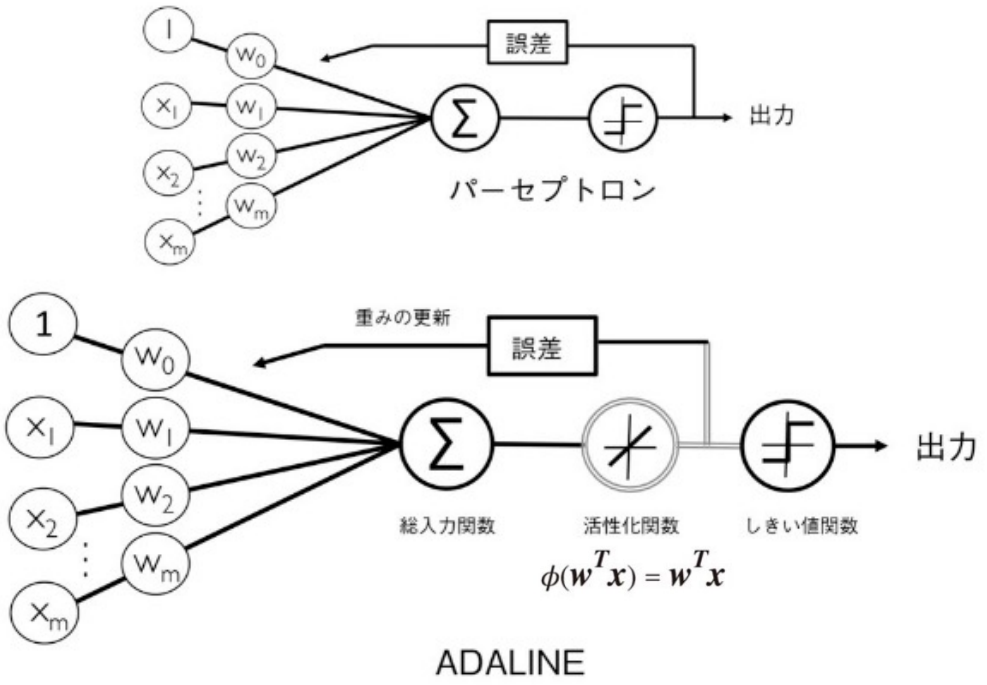
2.4 AdaLine と学習の収束



2.4 AdaLine と学習の収束

Widrow-Hoff モデル　パーセプトロンの改良
　どこが違うか？
　その違いはなぜ重要か？

活性化関数が恒等関数
コスト関数定義
その最小化



デルタ規則（Widrow-Hoff 則）

$$\phi(w^T x) = w^T x$$

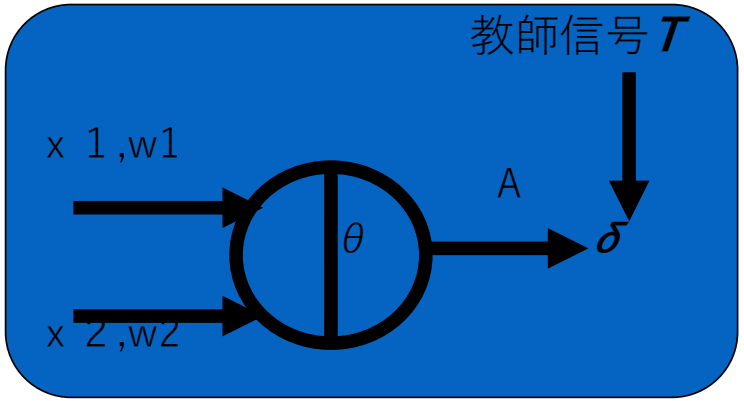
目標出力 ***T,*** 実際出力 ***A,*** 誤差 $\delta = (T-A)$

パーセプトロンアルゴリズムにおいて,

$\delta = 0$は2-1に, $\delta > 0$は2-2に, $\delta < 0$は2-3にそれぞれ対応する.

修正量 $\Delta_i = \eta \delta x_i$

変更式 $w_i(n+1) = w_i(n) + \Delta_i$



2.5 勾配降下法によるコスト関数の最小化

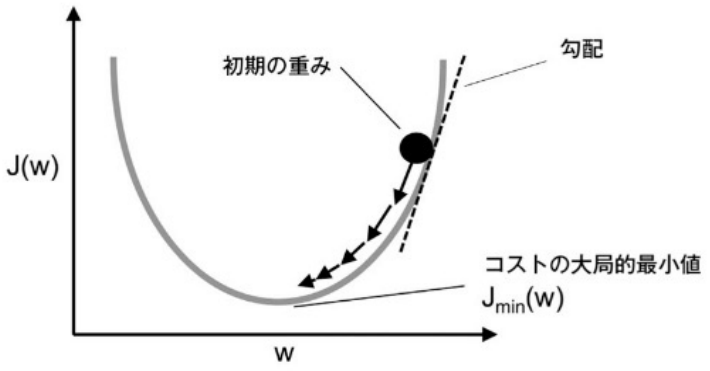
- 機械学習の学習過程とは目的関数を最適化することである
- 一般に、目的関数として最小化したいコスト関数を用いる
- **コスト関数（誤差関数）**を最小化する
  - Sum of Squared Error: SSE　（誤差平方和）

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2 \qquad (2.5.1)$$

重みの変化である $\Delta \boldsymbol{w}$ は、負の勾配に学習率 η を掛けたものとして定義される。

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \qquad (2.5.3)$$

※ 36　［監注］直後で説明されるように、勾配 $\nabla J(w)$ はコスト関数 J の重みベクトル w に関する偏導関数である。偏微分を表す記号 ∇ は「ナブラ」(nabla) と呼ばれる。



Adaline実装

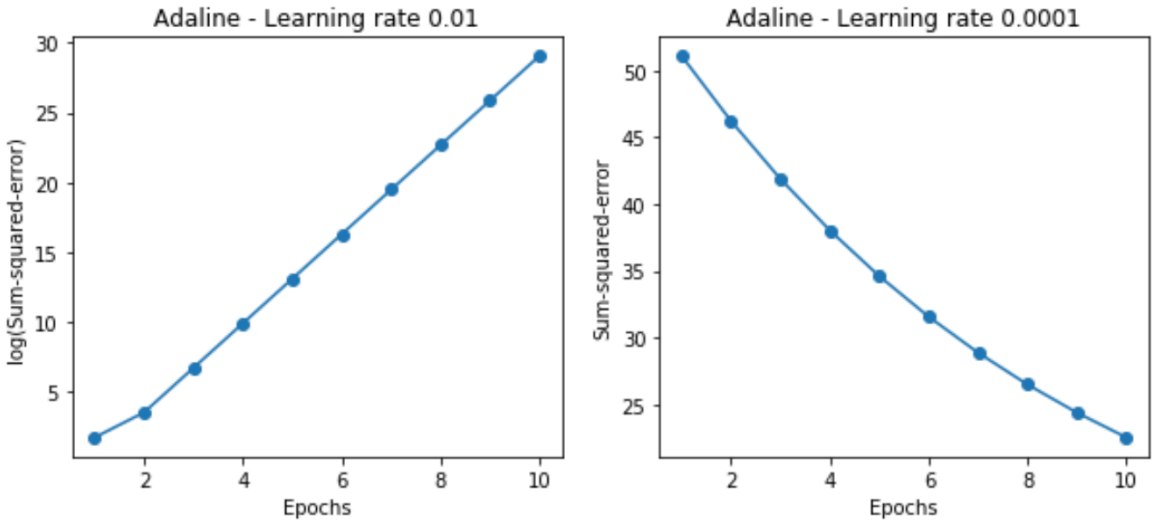
```
class AdalineGD(object):
    """ADAptive LInear NEuron 分類器

    パラメータ
    ------------
    eta : float
        学習率 (0.0 より大きく 1.0 以下の値 )
    n_iter : int
        トレーニングデータのトレーニング回数
    random_state : int
        重みを初期化するための乱数シード

    属性
    -----------
    w_ : 1 次元配列
        適合後の重み
    cost_ : リスト
        各エポックでの誤差平方和のコスト関数

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ トレーニングデータに適合させる

        パラメータ
        ----------
        X : { 配列のようなデータ構造 }, shape = [n_samples, n_features]
            トレーニングデータ
            n_sample はサンプルの個数、n_feature は特徴量の個数
        y : 配列のようなデータ構造 , shape = [n_samples]
            目的変数

        戻り値
        -------
        self : object
```

Adaline実装

```
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):  # トレーニング回数分トレーニングデータを反復
        net_input = self.net_input(X)
        # activation メソッドは単なる恒等関数であるため、
        # このコードでは何の効果もないことに注意。代わりに、
        # 直接 `output = self.net_input(X)` と記述することもできた。
        # activation メソッドの目的は、より概念的なものである。
        # つまり、（後ほど説明する）ロジスティック回帰の場合は、
        # ロジスティック回帰の分類器を実装するために
        # シグモイド関数に変更することもできる
        output = self.activation(net_input)
        # 誤差 y^(i) - φ(z^(i)) の計算
        errors = (y - output)
        # w_1, ..., w_m の更新
        # Δw_j = ηΣ_i(y^(i) - φ(z^(i))) x_j^(i) (j = 1, ..., m)
        self.w_[1:] += self.eta * X.T.dot(errors)
        # w_0 の更新 Δw_0 = ηΣ_i(y^(i) - φ(z^(i)))
        self.w_[0] += self.eta * errors.sum()
        # コスト関数の計算 J(w) = 1/2 Σ_i (y^(i) - φ(z^(i)))^2
        cost = (errors**2).sum() / 2.0
        # コストの格納
        self.cost_.append(cost)
    return self

def net_input(self, X):
    """ 総入力を計算 """
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

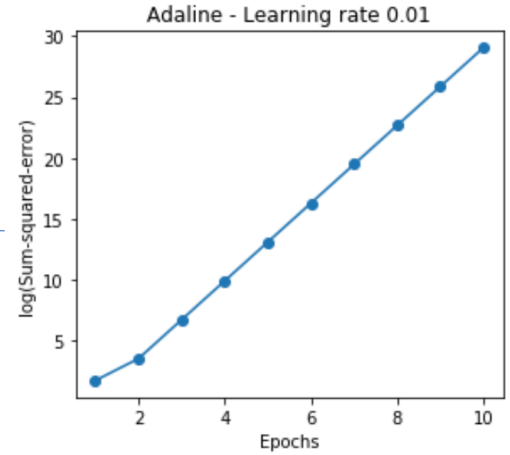
Adalineの実行：２つの学習率でコスト関数をプロットしてみる

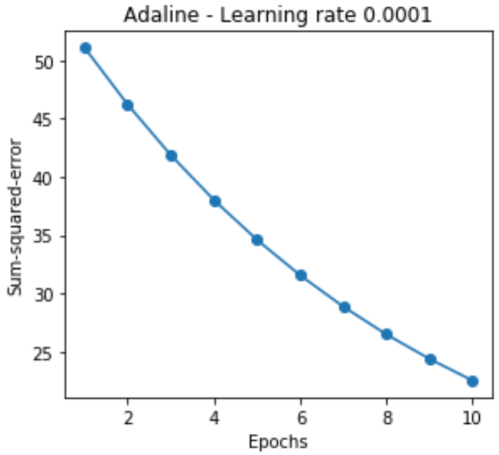
```
>>> # 描画領域を１行２列に分割
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> # 勾配降下法による ADALINE の学習（学習率 eta=0.01）
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> # エポック数とコストの関係を表す折れ線グラフのプロット（縦軸のコストは常用対数）
>>> ax[0].plot(range(1,len(ada1.cost_)+1), np.log10(ada1.cost_), marker='o')
>>> # 軸のラベルの設定
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> # タイトルの設定
>>> ax[0].set_title('Adaline - Learning rate 0.01')
>>> # 勾配降下法による ADALINE の学習（学習率 eta=0.0001）
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> # エポック数とコストの関係を表す折れ線グラフのプロット
>>> ax[1].plot(range(1,len(ada2.cost_)+1), ada2.cost_, marker
>>> # 軸のラベルの設定
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> # タイトルの設定
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> # 図の表示
>>> plt.show()
```

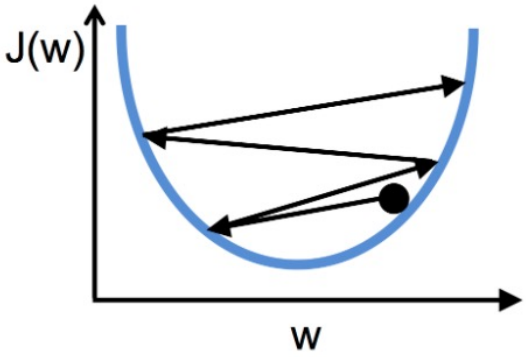


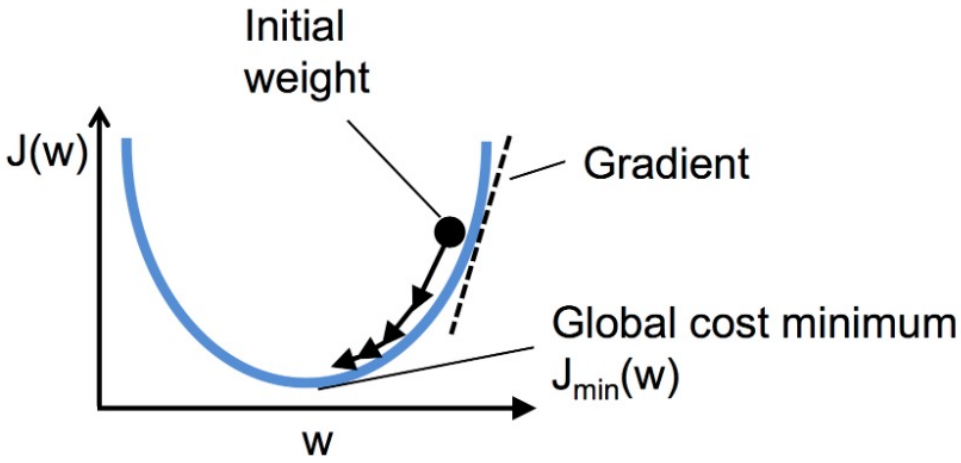
Adalineの実行：２つの学習率でコスト関数をプロットしてみる：考察

学習率大きすぎると
→ 誤差発散
学習率小さいと
→ 誤差収束も時間がかかる



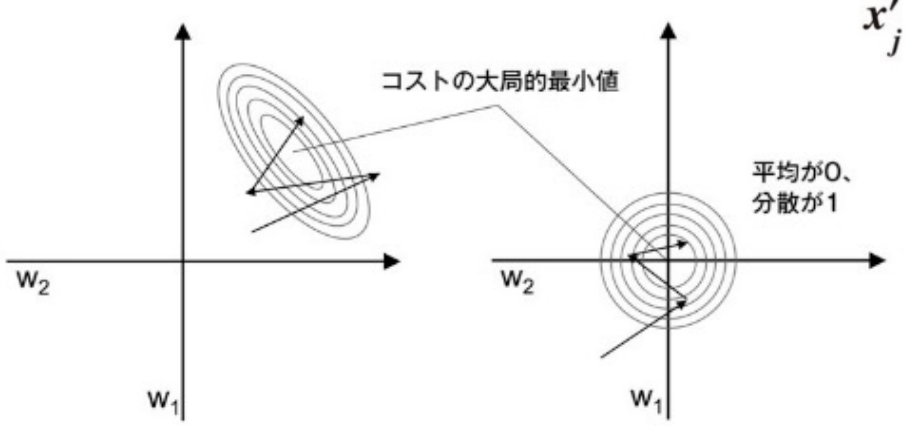






2.5.2 特徴量のスケーリングを通じて勾配降下法を改善する

- Standardization 標準化
  - データに標準正規分布の特性を与える
    （各特徴量の平均をずらして中心が0になるようにし、標準偏差を1にする）

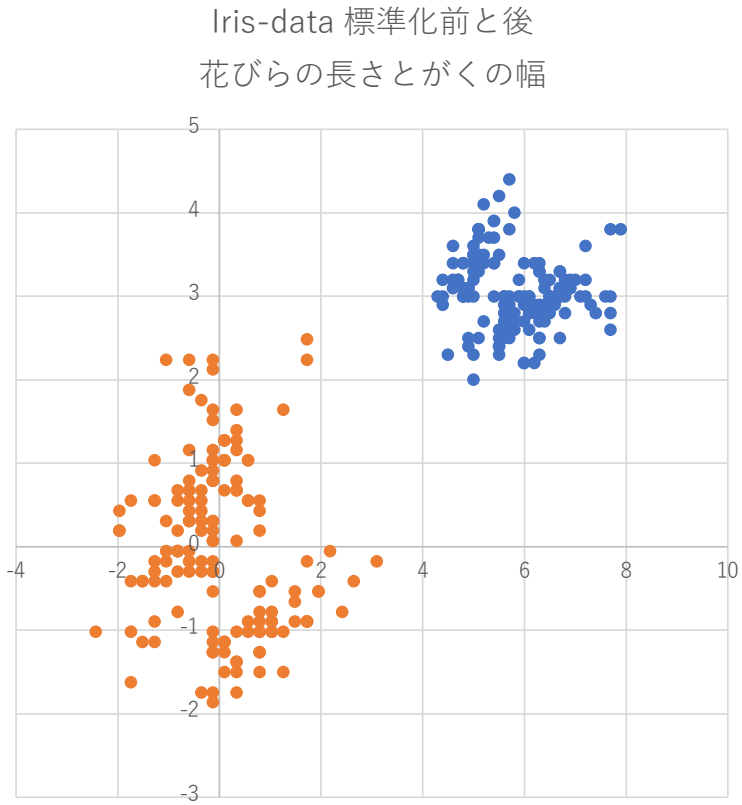


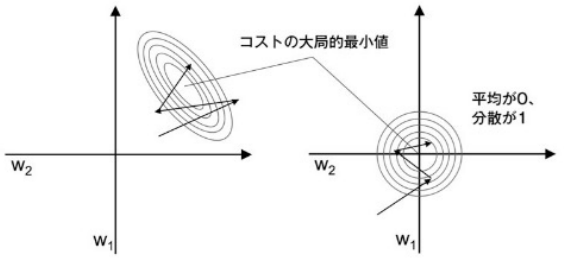
$$x'_j = \frac{x_j - \mu_j}{\sigma_j} \qquad (2.5.6)$$

属性jの平均μ_jを各属性から引いて
標準偏差で割る

属性ごとのレンジがバラバラ
標準化により各属性の影響度を均一にする

- Standardization 標準化
  - データに標準正規分布の特性を与える
    （各特徴量の平均をずらして中心が0になるようにし、標準偏差を1にする）



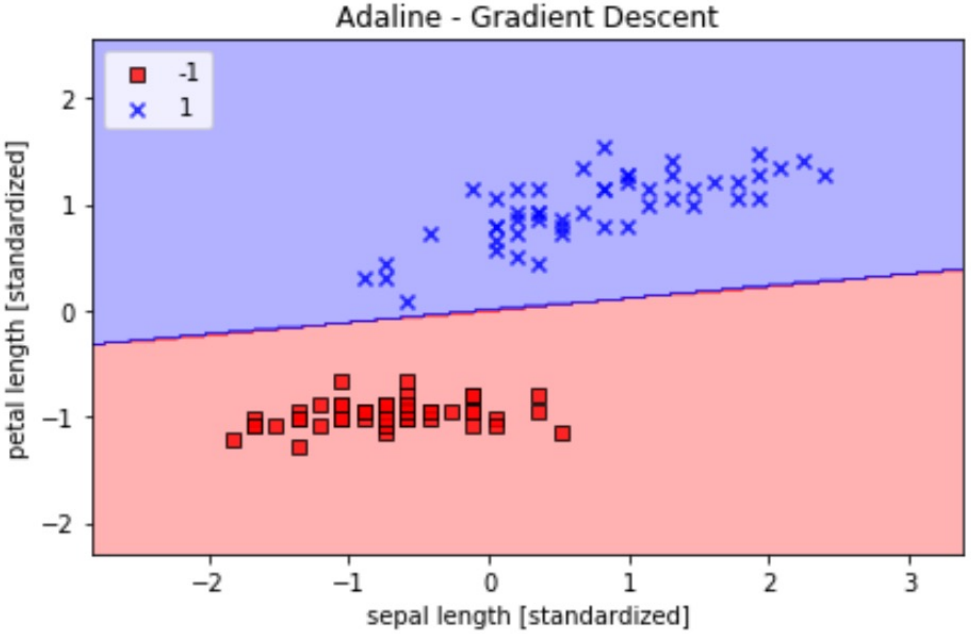


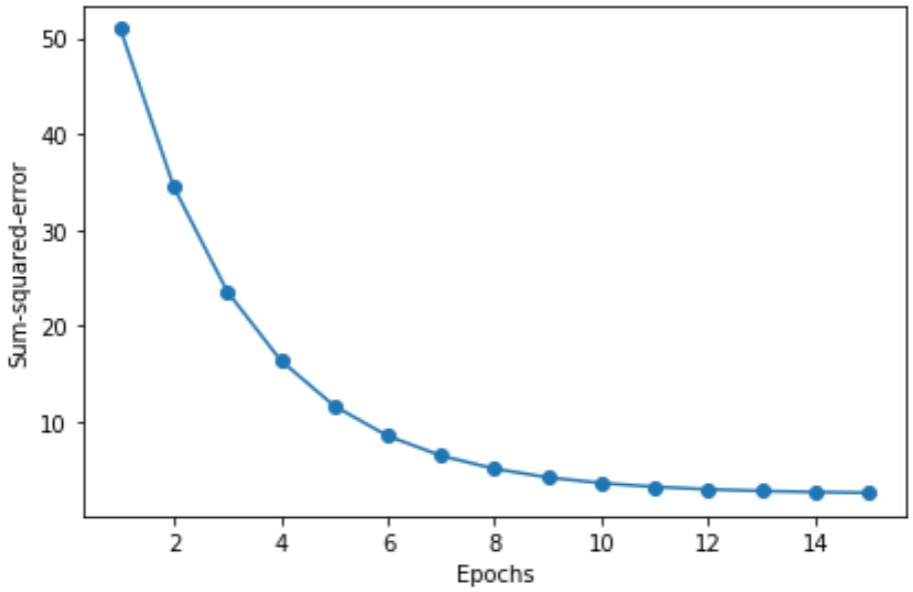
$$x'_j = \frac{x_j - \mu_j}{\sigma_j} \qquad (2.5.6)$$

属性jの平均μ_jを各属性から引いて
標準偏差で割る

属性ごとのレンジがバラバラ
標準化により各属性の影響度を均一にする

標準化データの学習結果





まとめ

- 教師あり学習
  - 分類と回帰
  - 二値分類（多値分類）
  - ラベル付きデータセット
  - 分類モデルの古典的な例
    - ニューラルネットの古典
      - 形式ニューロン
      - パーセプトロン学習
      - パーセプトロンからAdaLine
        - コスト関数
        - 勾配降下法
  - スケーリング