ae-6.ニューラルネットワークの基礎と学習メカニズム

（AI 演習）（全１５回）

https://www.kkaneko.jp/ai/ae/index.html

金子邦彦

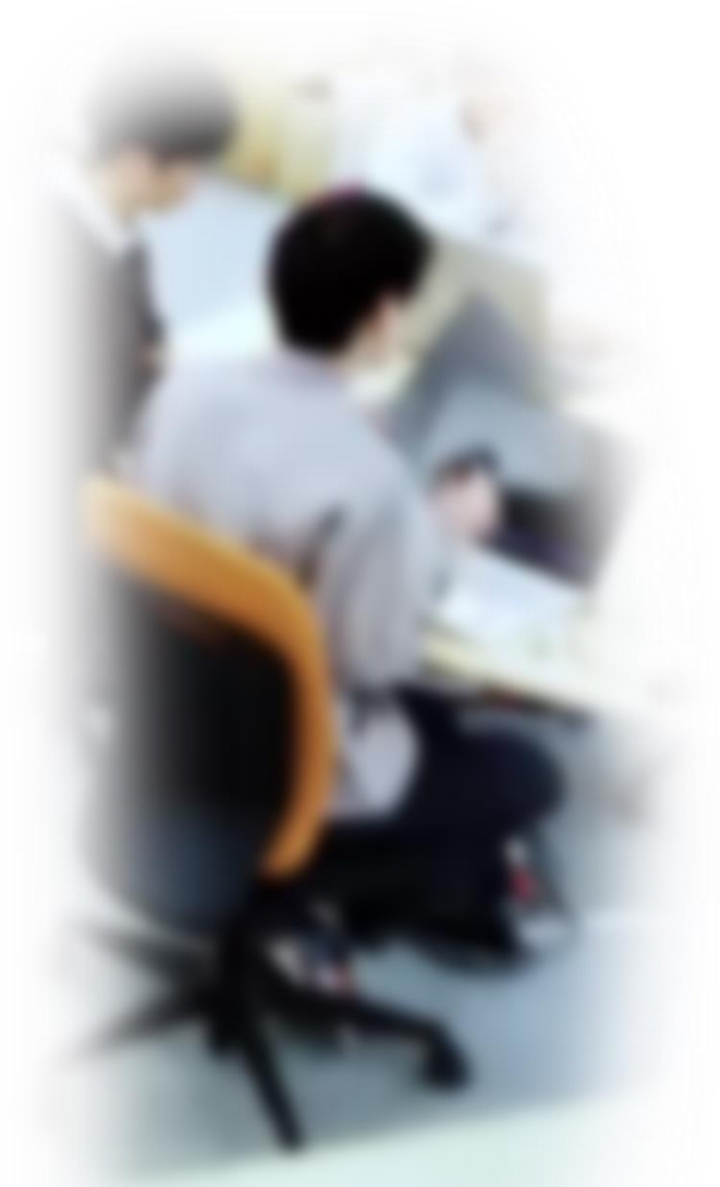
CC BY NC SA

人工知能（AI）の学習のための指針

1. **実践重視**： AIツールを実際に使用し、機能に慣れる

2. **エラーを恐れない**： 実行においては、エラーの発生の可能性がある．エラーを恐れず，むしろ学習の一部として捉えるポジティブさが大切．

3. **段階的学習**：基礎から応用へと段階的に学習を進め，AIの可能性を前向きに捉える



アウトライン

1. イントロダクション（機械学習）
2. 学習と最適化
3. 過学習
4. ニューラルネットワークでの過学習概要
5. ニューラルネットワークの基本構造とフォワードプロパゲーション
6. ニューラルネットワークを用いた分類
7. バックプロパゲーション
8. ディープラーニングの特徴と応用

trinket

- **Trinket** は**オンライン**の **Python、HTML** 等の**学習サイト**
- ブラウザで動作
- 有料の機能と無料の機能がある
- **自分が作成した Python プログラムを公開し、他の人に実行してもらうことが可能**（そのとき、書き替えて実行も可能）

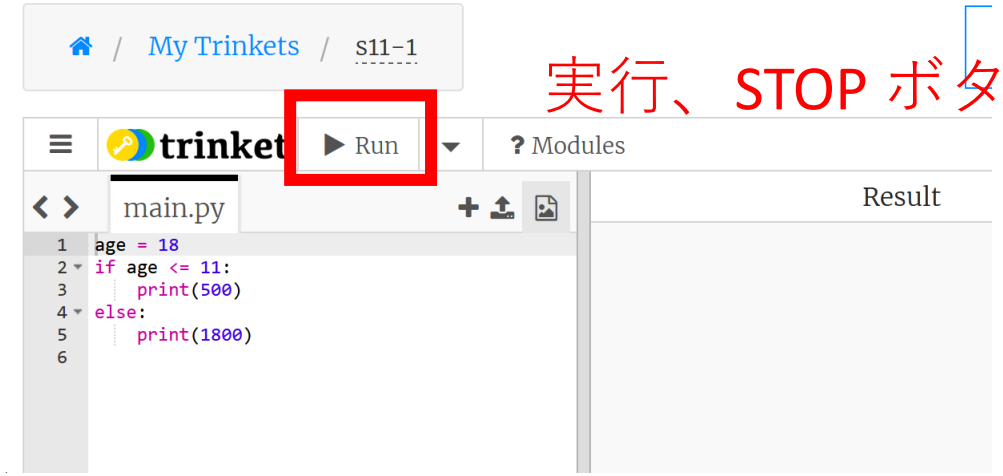
- **Python の標準機能**を登載、その他、次の外部ライブラリがインストール済み

matplotlib.pyplot, numpy, processing, pygal

trinket でのプログラム実行

- **trinket は Python, HTML などのプログラムを書き実行できる**サイト
- https://trinket.io/python/0fd59392c8

のように、違うプログラムには違う URL が割り当てられる



- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

6-1. イントロダクション（機械学習）

機械学習の基本

機械学習は，**コンピュータ**が**データ**を使用して**学習**することにより**知的能力を向上**させる技術

- **情報の抽出**：データからパターンや関係性を自動で見つけ出す能力を持つ
- **知的なタスクの実行**：予測，分類などの知的なタスクを実行

機械学習の３つの特徴

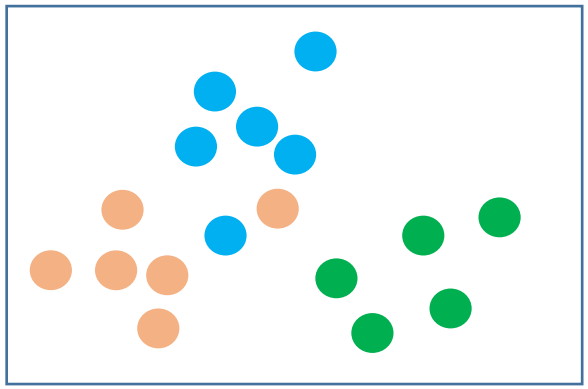
1. **情報の抽出**：データからパターンや関係性を自動で見つけ出す能力を持つ
2. **簡潔さ**：人間が設定していたルール等を，自動生成できる
3. **限界の超越**：従来の方法では困難だった課題も解決できる可能性がある

機械学習における訓練データの役割

- **訓練データ**は，**機械学習の学習に使用されるデータ**である．
- **データを用いた学習**により，コンピュータは分類や予測などの**知的な能力を獲得**する．

  例：画像分類では分類済みの大量データを使用する．

訓練データ



3種類に分類済み

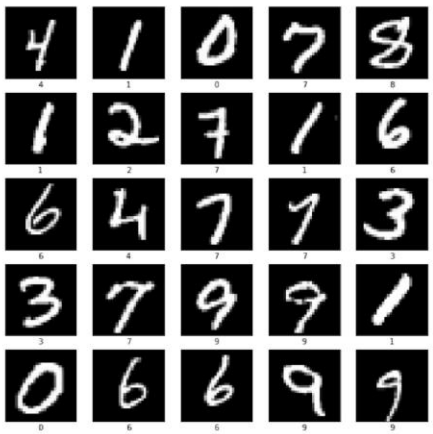
学習

学習者

機械学習の学習プロセス

Database Lab.

①データの準備：
目的に応じた訓練データを準備



画像 60000枚
（うち一部）

4 1 0 7 8
1 2 7 1 6
6 4 7 7 3
3 7 9 9 1
0 6 6 9 9

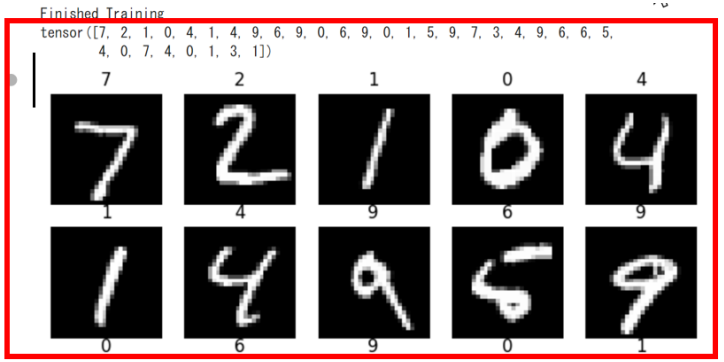
正解 60000個
（うち一部）

②学習の実行：
データを用いてパターンを学習.
モデルを構築

プログラム

データを用いて学習を行う．学習の結果、文字認識の能力を獲得.

③タスクの実行：
新しいデータを処理



機械学習まとめ

機械学習の特徴

- データを用いて知的能力を向上
- 自動でデータのパターンを抽出
- さまざまなタスクを自動実行

応用事例

画像理解、自然言語処理、予測など多数

機械学習の定義：

- **訓練データ**を用いて**学習**し、その結果として知的能力が向上
- **訓練データの追加**により、さらに**知的能力が向上**する可能性

機械学習の特徴と応用

- **能力の多様性**:

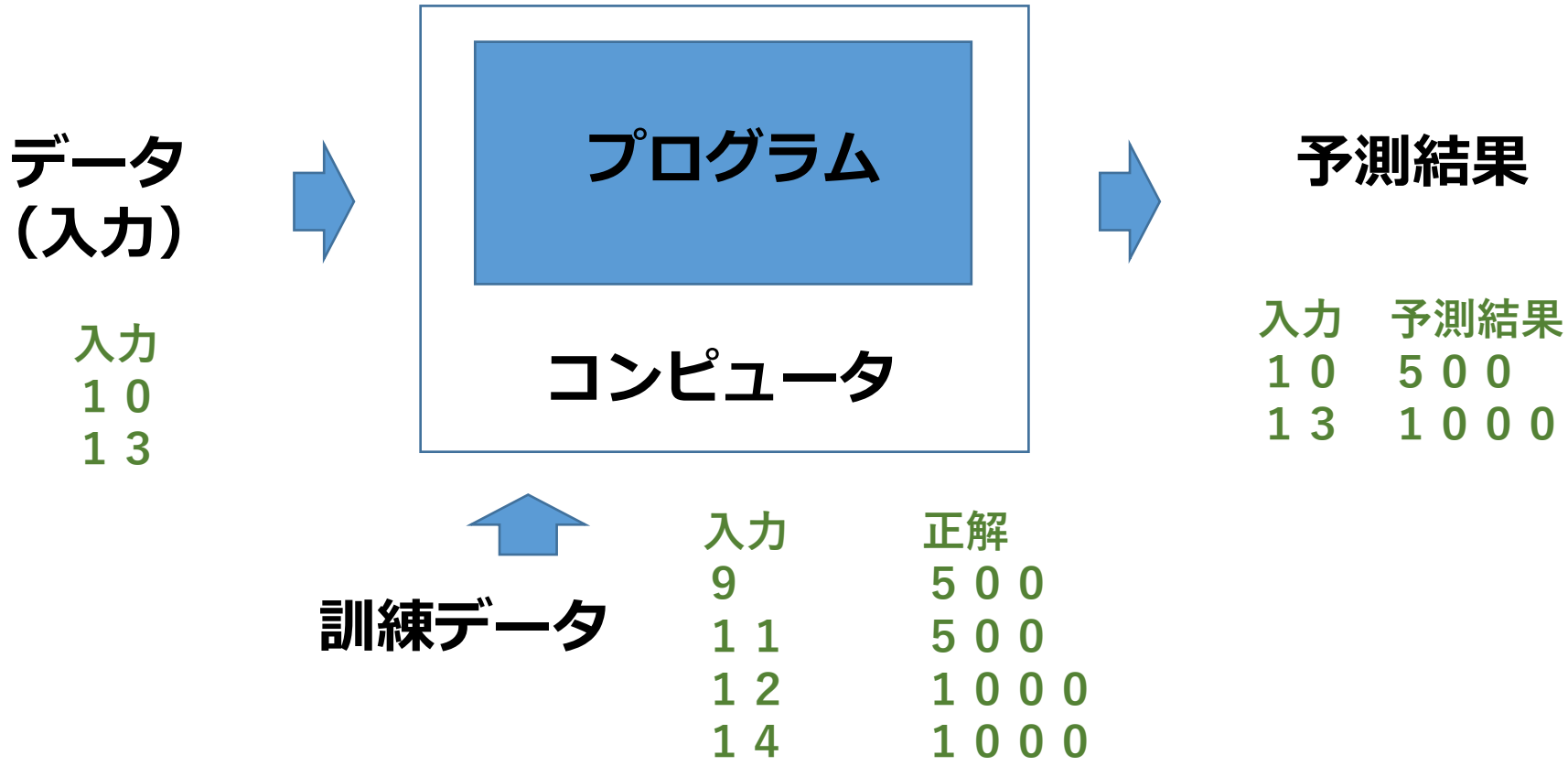
分類、検出、識別、認識（文字認識、画像の認識、音声認識）、予測、合成、翻訳、特徴抽出、ランク付け、情報推薦など、**さまざまな能力を持つ**

- **応用範囲の広さ**:

経済、金融、ロボット、医療、医学研究（遺伝子解析）など、様々な分野で問題解決に活用

機械学習は多様な能力を持ち、経済や金融から医療やロボットまで広範な応用が可能

機械学習によるプロセス



- 機械学習では，訓練データから自動的にパターンや関連性を学習する．
- 学習結果のモデルを利用して、新しいデータ（未知のデータ）に対して，タスクが実行される．人手による規則の記述が不要である．

まとめ

- **機械学習の基本**：データを用いた学習による知的能力の向上
- **訓練データの重要性**：学習は訓練データに依存する
- **汎化**：**未知データへの対応**
- 実践的応用：様々な分野での活用可能性

6-2. 学習と最適化

機械学習と最適化の関係

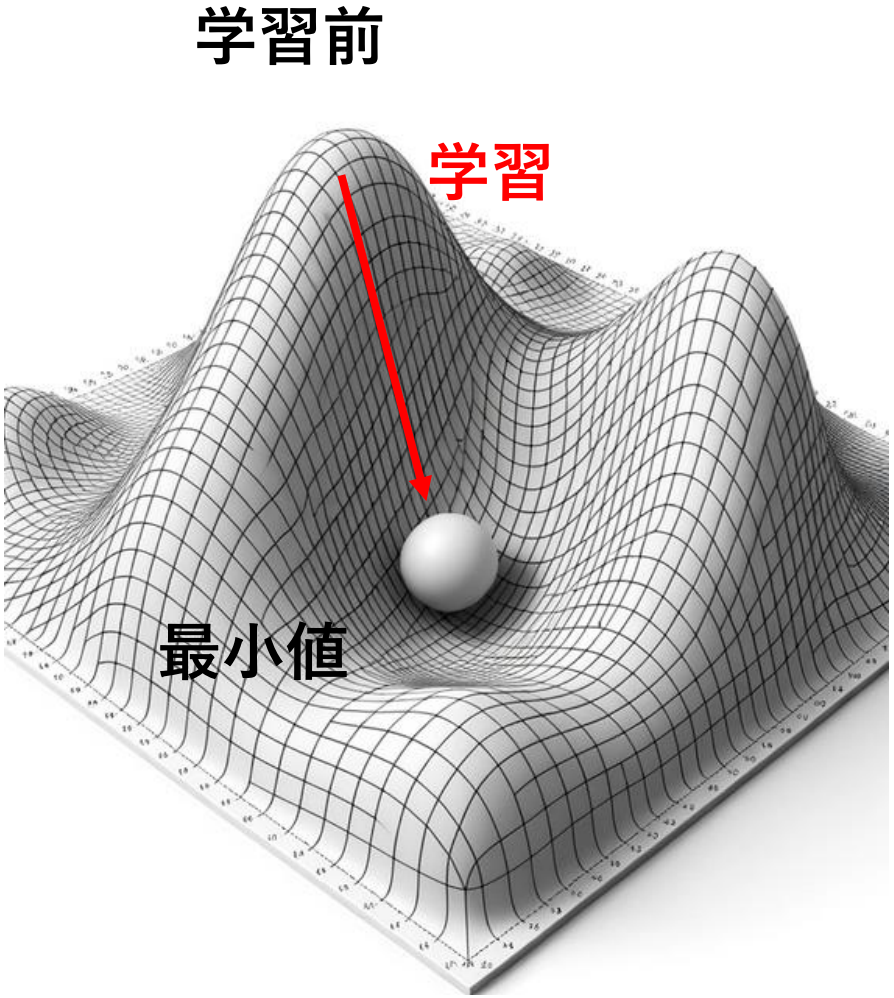
- **機械学習**：**データを用いて**パターンを学習し，自動で分類や予測などの知的タスクを実行する

- **最適化**：**誤差（損失）を最小化**するため，**データに最も適したパラメータを見つける**こと.

- **最適化手法**の代表例：
  - **勾配降下法**：**誤差（損失）が減る方向に少しずつ進む**ことで，最適解に近づく．※　勾配を使用.
  - **ニュートン法**：**勾配降下法よりも速く結果を得る方法**．関数の勾配と曲率を利用することで、誤差（損失）の最小化を効率的に行う　※　勾配と曲率が必要

誤差（損失）の再紹介

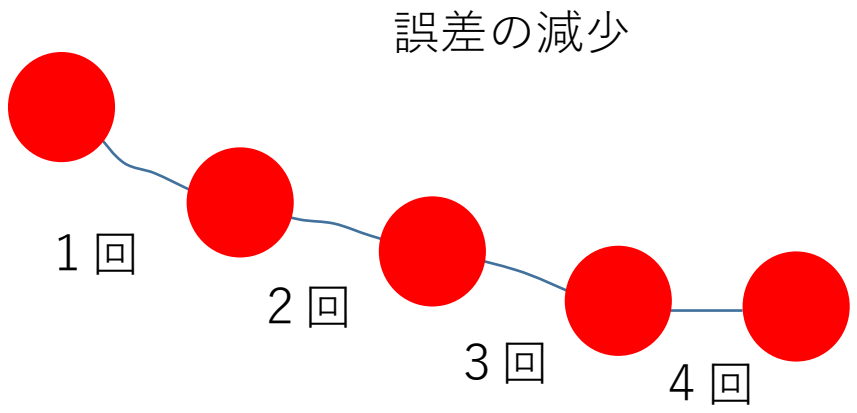
誤差（損失）の値が**最小化**するように学習が行われる



下に行くほど
誤差の値が
小さい場合

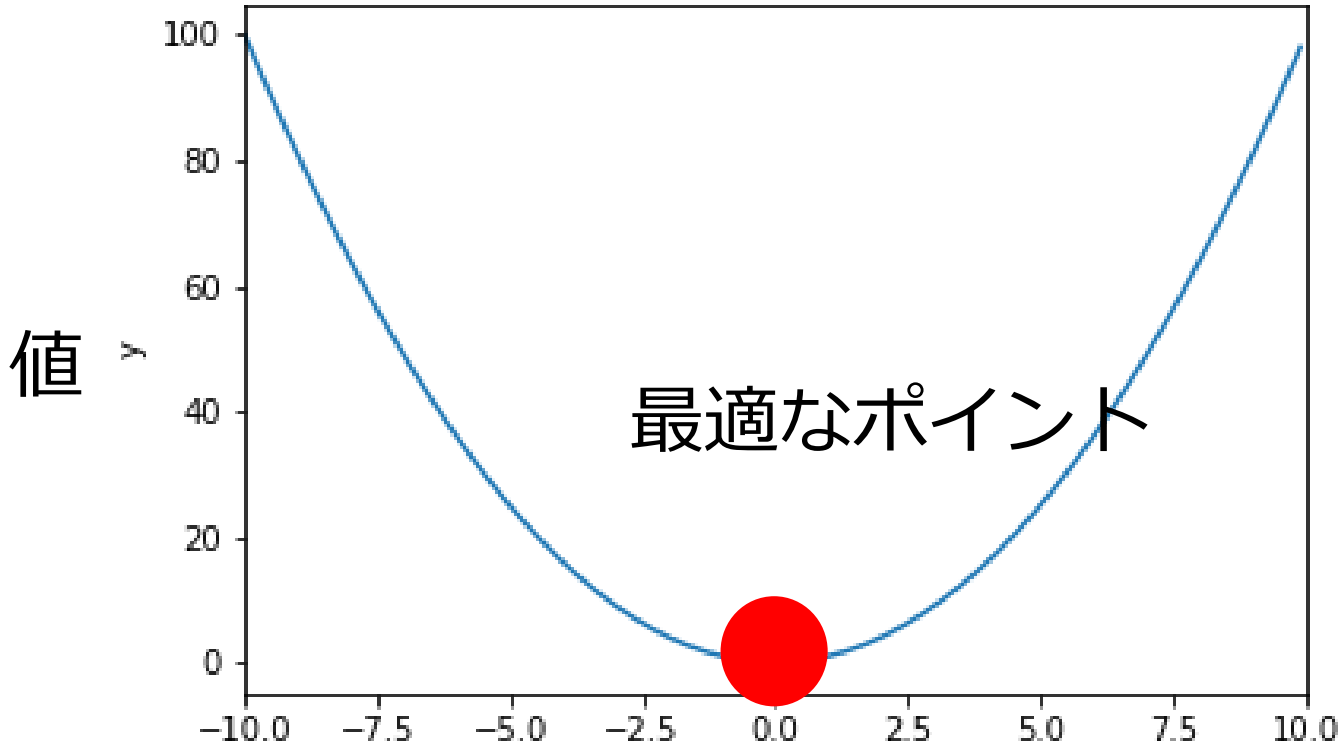
学習の繰り返しの重要性

- １回の学習では，**学習不足**の場合がある．
- **学習の繰り返し**により誤差が減少する．
- 1回，2回，3回，4回と学習を重ねることで，誤差が段階的に小さくなっていく



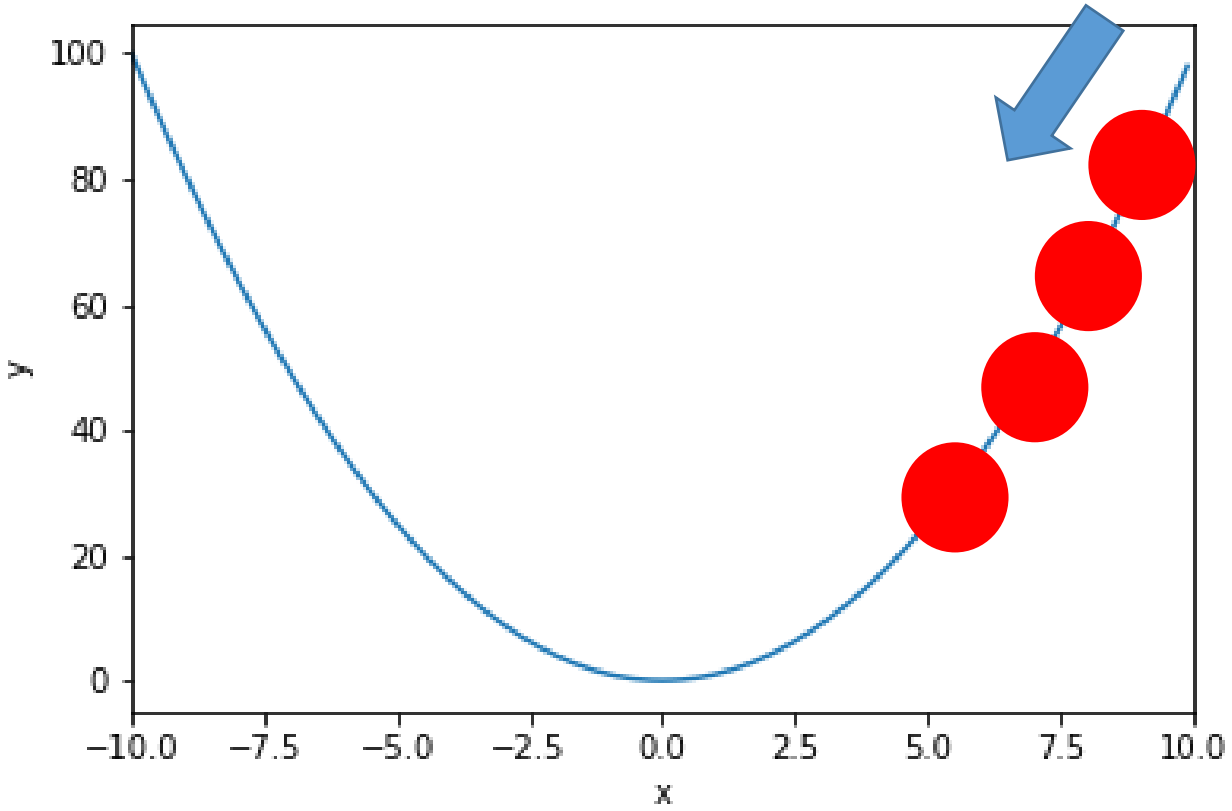
最適化における最適点の探索

パラメータ（横軸）を調整することで最適値に近づいていく.



最適化における最適点の探索

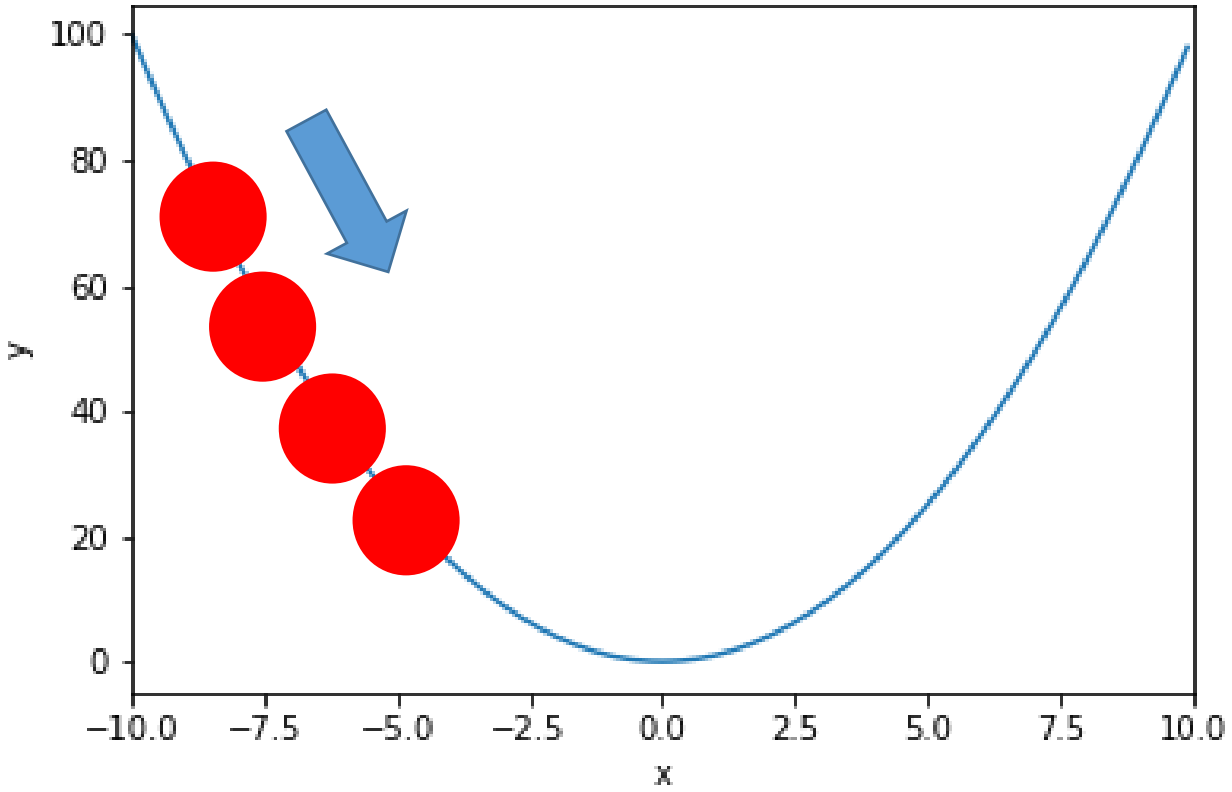
傾きが**右上がり**のとき：
パラメータ値を**減らす**と、**最適に近づく**



１回では最適にならない.
移動を繰り返す必要がある.

最適化における最適点の探索

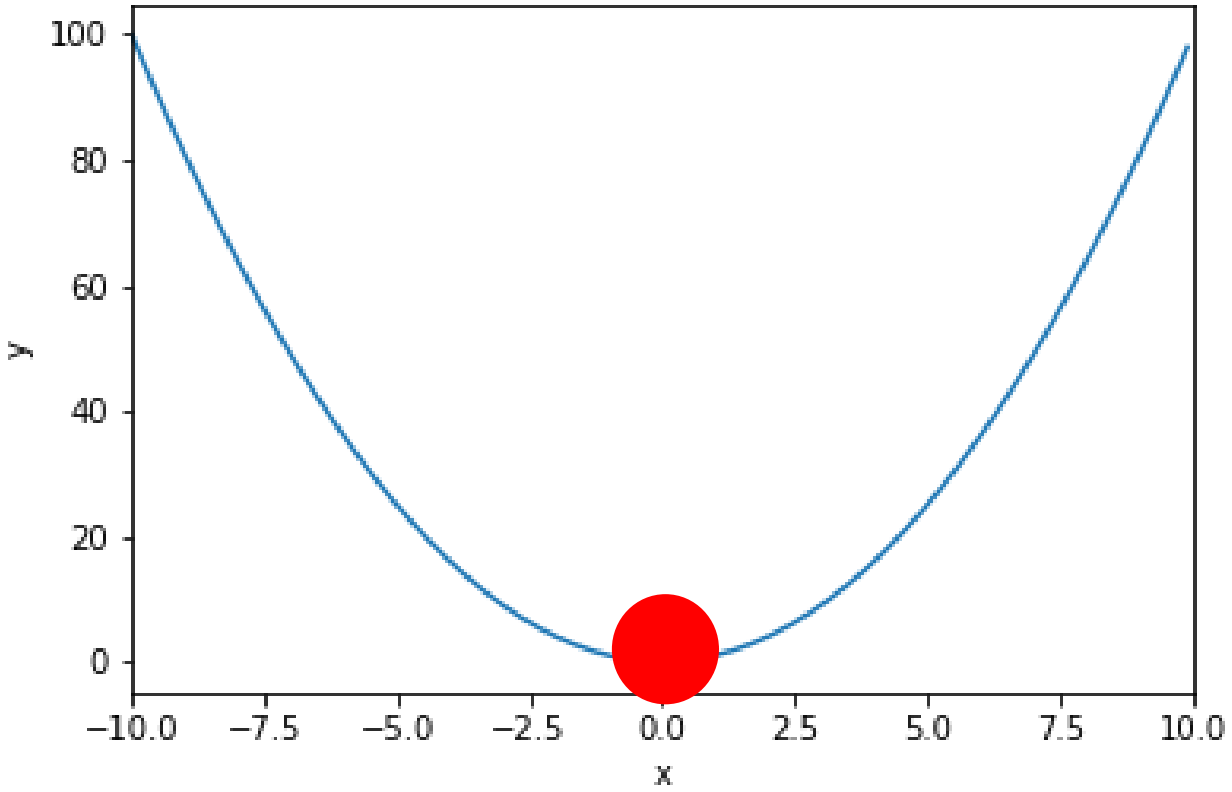
傾きが**右下がり**のとき：
パラメータ値を**増やす**と、**最適に近づく**



１回では最適にならない.
移動を繰り返す必要がある.

最適化における最適点の探索

- 最適のとき**傾きは 0**
- **これ以上の移動は不要である**



演習１　最適化（勾配降下法）

演習の狙い

- 最適化の仕組みの実体験
  - **誤差が減少する方向に進む過程**を数値で確認
  - 終了までの変化を具体的に観察
- プログラムの特徴
  - **各ステップの状態（x値と誤差）を表示**
  - 終了判定による停止
- 具体例による学習
  - **二次関数$f(x)=x^2-4x+4$最小値を求める実例**

演習のポイント

- プログラムの目的
  - **関数 $f(x)=x^2-4x+4$ の最小値は $x=2$ のとき 0**
  - この関数は，平面上の**下に凸な放物線**で、**最小値に向かって降りていく過程**を観察
- パラメータの意味
  - learning_rate=0.1は**一回の移動量**を決める（大きすぎると発散、小さすぎると収束が遅い）
  - **初期値 initial_x=0.0** から，結果 x=2に向かって進んでいく過程を観察（最終結果は x=2 でなく，微小な誤差を含む）

① trinket の次のページを開く

https://trinket.io/python/56b98e9dd8d0

② 実行結果が，次のように表示されることを確認

main.py

```python
# 勾配降下法の実装
def gradient_descent(initial_x, learning_rate=0.1, tolerance=1e-6, max_iterations=100):
    x = initial_x
    for i in range(max_iterations):
        # 現在の勾配を計算
        gradient = f_prime(x)

        # 次の x の値を計算
        x_new = x - learning_rate * gradient

        # 誤差の計算（変数の変化量）
        error = abs(x_new - x)

        # 誤差を小数点以下10桁で表示
        error_str = "{:.10f}".format(error)

        # ステップごとの出力
        print("ステップ", i+1, ": x ≈", x, ", 誤差 =", error_str)

        # 絶対誤差が許容範囲内かをチェック
        if error < tolerance:
            print("収束しました: x ≈", x, "（反復回数:", i+1, "回）")
            return x_new

        # 更新
        x = x_new

    print("収束しませんでした")
    return None

# 初期値の設定と実行
initial_x = 0.0
result = gradient_descent(initial_x)
value = "{:.10f}".format(f(result))
print("xの値:", result, "のとき，関数の値は", value, "になる")
```

Result

```
('ステップ', 32, ': x ≈', 1.99801929579, ', 誤差 =', '0.0003961409')
('ステップ', 33, ': x ≈', 1.99841543669, ', 誤差 =', '0.0003169125')
('ステップ', 34, ': x ≈', 1.99873234918, ', 誤差 =', '0.0002535301')
('ステップ', 35, ': x ≈', 1.99898587927, ', 誤差 =', '0.0002028242')
('ステップ', 36, ': x ≈', 1.99918870347, ', 誤差 =', '0.0001622593')
('ステップ', 37, ': x ≈', 1.99935096279, ', 誤差 =', '0.0001298075')
('ステップ', 38, ': x ≈', 1.99948077029, ', 誤差 =', '0.0001038460')
('ステップ', 39, ': x ≈', 1.99958461633, ', 誤差 =', '0.0000830769')
('ステップ', 40, ': x ≈', 1.99966769321, ', 誤差 =', '0.0000664613')
('ステップ', 41, ': x ≈', 1.99973415449, ', 誤差 =', '0.0000531690')
('ステップ', 42, ': x ≈', 1.99978732351, ', 誤差 =', '0.0000425353')
('ステップ', 43, ': x ≈', 1.99982985882, ', 誤差 =', '0.0000340283')
('ステップ', 44, ': x ≈', 1.99986388716, ', 誤差 =', '0.0000272224')
('ステップ', 45, ': x ≈', 1.99989110961, ', 誤差 =', '0.0000217781')
('ステップ', 46, ': x ≈', 1.99991288774, ', 誤差 =', '0.0000174223')
('ステップ', 47, ': x ≈', 1.99993031003, ', 誤差 =', '0.0000139380')
('ステップ', 48, ': x ≈', 1.99994424799, ', 誤差 =', '0.0000111504')
('ステップ', 49, ': x ≈', 1.9999553984, ', 誤差 =', '0.0000089202')
('ステップ', 50, ': x ≈', 1.9999643186, ', 誤差 =', '0.0000071363')
('ステップ', 51, ': x ≈', 1.99997145489, ', 誤差 =', '0.0000057090')
('ステップ', 52, ': x ≈', 1.99997716388, ', 誤差 =', '0.0000045672')
('ステップ', 53, ': x ≈', 1.99998173112, ', 誤差 =', '0.0000036537')
('ステップ', 54, ': x ≈', 1.99998538486, ', 誤差 =', '0.0000029230')
('ステップ', 55, ': x ≈', 1.99998830785, ', 誤差 =', '0.0000023384')
('ステップ', 56, ': x ≈', 1.99999064621, ', 誤差 =', '0.0000018707')
('ステップ', 57, ': x ≈', 1.99999251693, ', 誤差 =', '0.0000014966')
('ステップ', 58, ': x ≈', 1.99999401351, ', 誤差 =', '0.0000011973')
('ステップ', 59, ': x ≈', 1.99999521078, ', 誤差 =', '0.0000009579')
('収束しました: x ≈', 1.99999521078, '（反復回数:', 59, '回）')
('xの値:', 1.99999616868, 'のとき，関数の値は', '0.0000000000', 'になる')
```

- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

演習２　最適化（ニュートン法）

演習の狙い

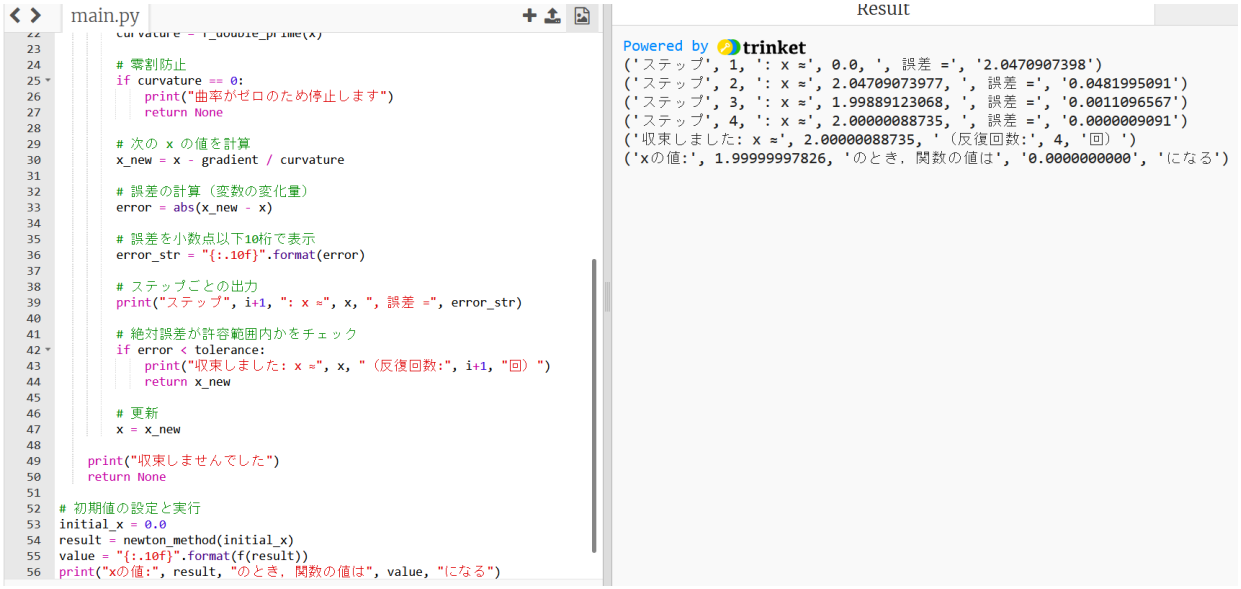
- **最適化手法の実践的な比較**
  - 勾配降下法とニュートン法の効率の違いを実際に観察
  - **同じ初期値 x=0** から**最小値 x=2 に向かう過程の違い**を確認

- **両手法の特徴の違いを理解**
  - **勾配降下法，ニュートン法ともに同じ機能を持つ**が，効率や前提条件（ニュートン法では曲率が必要）に違いがある.

① trinket の次のページを開く

https://trinket.io/python/b6a1de735025

② 実行結果が，次のように表示されることを確認



- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

発展課題（余裕のある人向け）

プログラム内の関数 f は $x^2 - 4x + 4$ を示している

```
def f(x):
    return x**2 - 4*x + 4
```

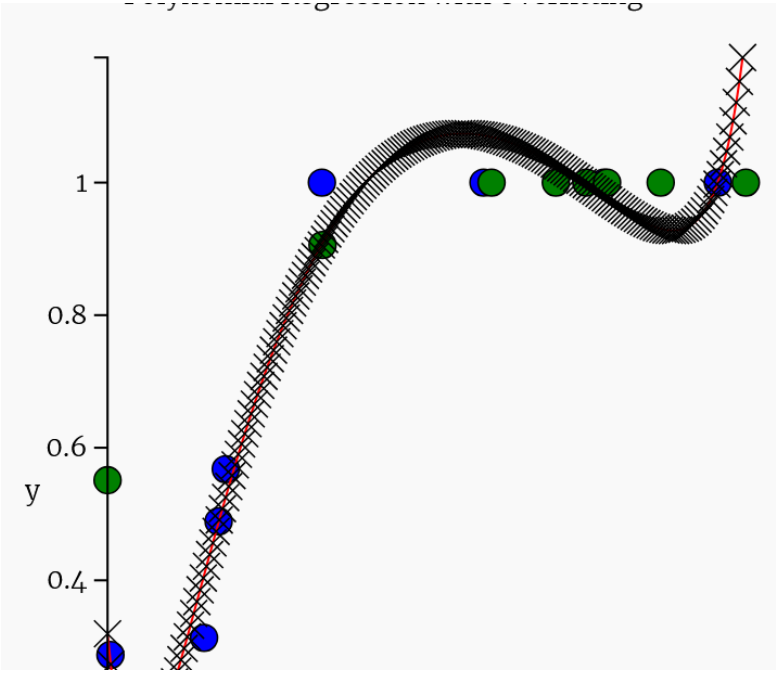
これを書き換えて，別の関数で試してみなさい（例 $x^2 - 2x + 1$）

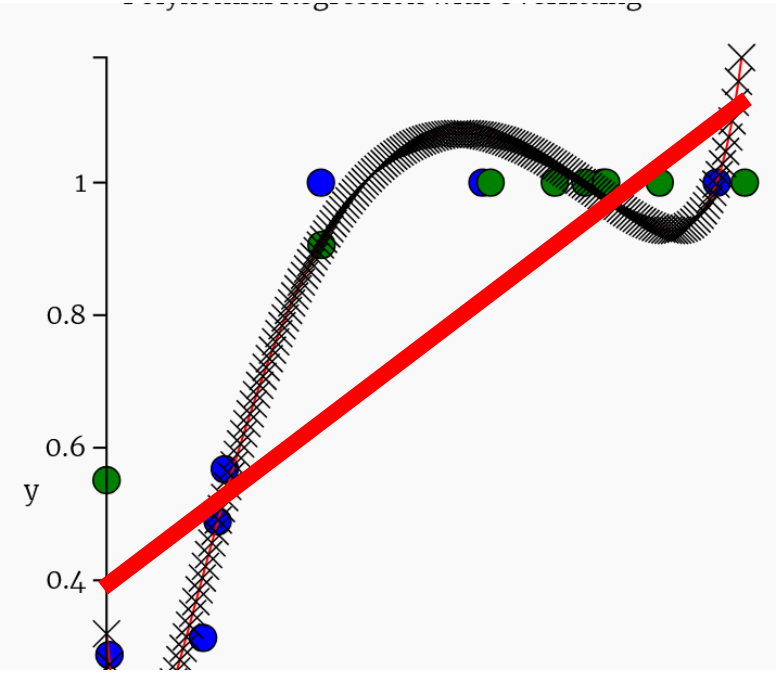
6-3. 過学習

過学習

- **訓練データ**に**過剰に適合する現象.**
- 訓練データから，実在しない特徴やパターンを発見してしまう.
- 訓練データでの**誤差は減少**するが，**未知のデータに対する性能が低下する**.
- 完全な防止は困難だが，様々な技術で抑制を図る.

過学習の例





過学習

訓練データ（青）
に過剰に適合

実際のパターン

シンプルな増加.

演習 3　過学習

演習の狙い

- **訓練データ**

  個数８個，ノイズ付き（青色でプロット）

- **機械学習モデルと学習**
  - 多項式を使用（ニューラルネットワークではない）
  - 勾配降下法を使用
- **過学習の評価と可視化**
  - **訓練データでの誤差**，**検証データの誤差**を表示
  - プロット上で訓練データ(青)、検証データ(緑)、予測線であるモデル(赤)を表示.

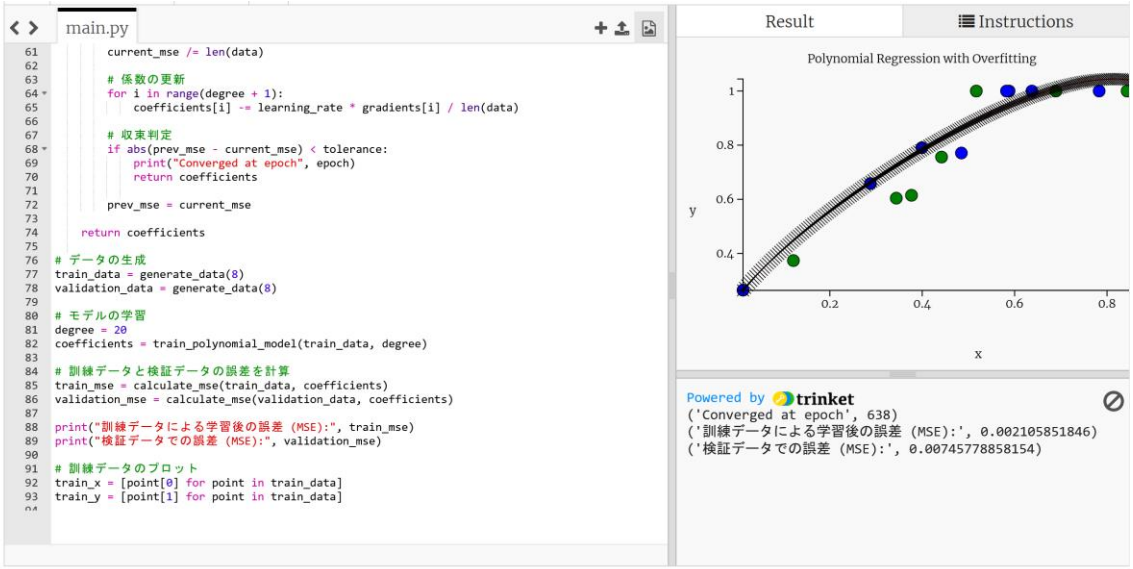
プログラムの要点説明

- モデルの複雑さと汎化性能の関係
  - モデルは **$ax^{19} + bx^{18} + \ldots = 0$** という**多項式**
  - モデルのパラメータ数 20個
  - **モデルが複雑**なほど，**訓練データへの適合度は向上**し，**汎化性能は低下**（ノイズまで学習してしまう）
- **検証データの役割**
  - **訓練に使用していないデータ**で，**本来のデータのパターンを学習を確認**
- **過学習の検出**
  - **訓練データ**での**誤差が小さく，検証データ**での**誤差が大きいとき**は**過学習.**

① trinket の次のページを開く

https://trinket.io/python/755acbb36d6f

② 実行結果が，次のように表示されることを確認



- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

過学習により発生する現象

訓練データに対して
→　**正しい**結果を出す

未知のデータに対し
→　**正しい**結果を出す

汎化の成功

訓練データに対して
→　**正しい**結果を出す

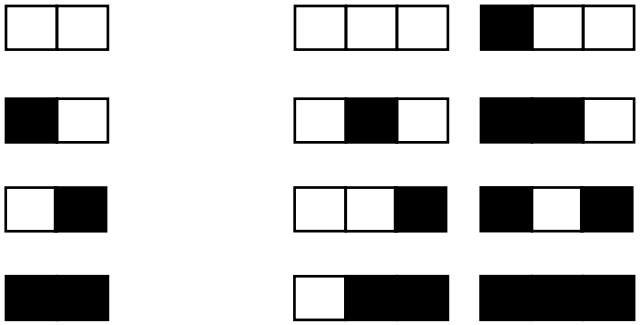
未知のデータに対し
→　**誤った**結果を出す

汎化の失敗
（過学習）

2011年までの常識

機械学習の**難問**：組み合わせが多い場合に，複雑なモデルが必要となり，過学習が発生する.

それほど
難しくない問題



２マスのパターン
（**４通り**）

３マスのパターン
（**８通り**）

難しい問題

１００マスのパターン
1267650600228229401496703205376 通り
（約１兆の１兆の１２６万倍）

組み合わせが多い

2011年までの常識

- **カラー画像: サイズ 32×32 の場合**

  → 画素数: 32 × 32 × 3，階調 256

  約 12000 … 000 通り

7397個の 0

- **ステレオ音声: 10秒間，サンプリング周波数 44100Hz の場合**

  → 10 × 44100，階調 256

  約 1000 … 000 通り

1062033個の 0

2009年以降の技術進展

- 従来からある技術と，新技術の組み合わせにより，性能向上，**過学習の抑止が進展**

【新技術の例】

  - 非線形性の正規化 (rectified nonlinearity), 2009年
  - 活性化関数の ReLU, 2011年
  - ドロップアウト
  - 正則化（L1，L2）
  - 学習率の調整
  - He の初期化, 2015年　など

- 巨大な訓練データの利用，コンピュータの高速化とあいまって　→　ニューラルネットワークの**層やユニット数を増やすことが一定程度可能に.**

学習の繰り返しを行うプログラム例

学習の繰り返し回数は **20**

```
EPOCHS=20
history = m.fit(x=ds_train[0],
          y=ds_train[1],
          epochs=EPOCHS,
          validation_data=(ds_test[0], ds_test[1]),
          callbacks=[tensorboard_callback], verbose=2)
```

訓練データの指定

検証データの指定

学習の繰り返しを行うプログラムと実行結果

同じ訓練データを用いた学習を２０回繰り返す.

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

EPOCHS=20
history = m.fit(x=ds_train[0],
          y=ds_train[1],
          epochs=EPOCHS,
          validation_data=(ds_test[0], ds_test[1]),
          callbacks=[tensorboard_callback], verbose=2)

m.evaluate(ds_test[0], ds_test[1], verbose=2)
```

プログラム

```
Epoch 1/20
1875/1875 - 5s - loss: 0.2589 - accuracy: 0.9262 - val_loss: 0.1427 - val_accuracy: 0.9594 - 5s/epoch - 3ms/step
Epoch 2/20
1875/1875 - 4s - loss: 0.1151 - accuracy: 0.9668 - val_loss: 0.0944 - val_accuracy: 0.9713 - 4s/epoch - 2ms/step
Epoch 3/20
1875/1875 - 4s - loss: 0.0778 - accuracy: 0.9763 - val_loss: 0.0808 - val_accuracy: 0.9736 - 4s/epoch - 2ms/step
Epoch 4/20
1875/1875 - 4s - loss: 0.0573 - accuracy: 0.9828 - val_loss: 0.0786 - val_accuracy: 0.9755 - 4s/epoch - 2ms/step
Epoch 5/20
1875/1875 - 4s - loss: 0.0450 - accuracy: 0.9859 - val_loss: 0.0743 - val_accuracy: 0.9765 - 4s/epoch - 2ms/step
Epoch 6/20
1875/1875 - 4s - loss: 0.0343 - accuracy: 0.9892 - val_loss: 0.0762 - val_accuracy: 0.9756 - 4s/epoch - 2ms/step
Epoch 7/20
1875/1875 - 4s - loss: 0.0285 - accuracy: 0.9912 - val_loss: 0.0720 - val_accuracy: 0.9791 - 4s/epoch - 2ms/step
Epoch 8/20
1875/1875 - 4s - loss: 0.0226 - accuracy: 0.9931 - val_loss: 0.0770 - val_accuracy: 0.9785 - 4s/epoch - 2ms/step
Epoch 9/20
1875/1875 - 4s - loss: 0.0183 - accuracy: 0.9943 - val_loss: 0.0774 - val_accuracy: 0.9792 - 4s/epoch - 2ms/step
Epoch 10/20
1875/1875 - 4s - loss: 0.0150 - accuracy: 0.9955 - val_loss: 0.0797 - val_accuracy: 0.9784 - 4s/epoch - 2ms/step
Epoch 11/20
1875/1875 - 4s - loss: 0.0141 - accuracy: 0.9956 - val_loss: 0.0828 - val_accuracy: 0.9784 - 4s/epoch - 2ms/step
Epoch 12/20
1875/1875 - 4s - loss: 0.0107 - accuracy: 0.9965 - val_loss: 0.0821 - val_accuracy: 0.9786 - 4s/epoch - 2ms/step
Epoch 13/20
1875/1875 - 4s - loss: 0.0106 - accuracy: 0.9967 - val_loss: 0.0914 - val_accuracy: 0.9774 - 4s/epoch - 2ms/step
Epoch 14/20
1875/1875 - 4s - loss: 0.0085 - accuracy: 0.9974 - val_loss: 0.0931 - val_accuracy: 0.9766 - 4s/epoch - 2ms/step
Epoch 15/20
1875/1875 - 4s - loss: 0.0068 - accuracy: 0.9981 - val_loss: 0.0915 - val_accuracy: 0.9781 - 4s/epoch - 2ms/step
Epoch 16/20
1875/1875 - 4s - loss: 0.0076 - accuracy: 0.9977 - val_loss: 0.0885 - val_accuracy: 0.9798 - 4s/epoch - 2ms/step
Epoch 17/20
1875/1875 - 5s - loss: 0.0065 - accuracy: 0.9980 - val_loss: 0.0965 - val_accuracy: 0.9792 - 5s/epoch - 3ms/step
Epoch 18/20
1875/1875 - 5s - loss: 0.0060 - accuracy: 0.9981 - val_loss: 0.0936 - val_accuracy: 0.9785 - 5s/epoch - 3ms/step
Epoch 19/20
1875/1875 - 4s - loss: 0.0063 - accuracy: 0.9981 - val_loss: 0.1017 - val_accuracy: 0.9784 - 4s/epoch - 2ms/step
Epoch 20/20
1875/1875 - 4s - loss: 0.0043 - accuracy: 0.9987 - val_loss: 0.1105 - val_accuracy: 0.9761 - 4s/epoch - 2ms/step
313/313 - 0s - loss: 0.1105 - accuracy: 0.9761 - 394ms/epoch - 1ms/step
[0.1105121374130249, 0.9761000275611877]
```

実行結果

学習の**繰り返し**における，訓練データや検証データでの**誤差**の変化を確認

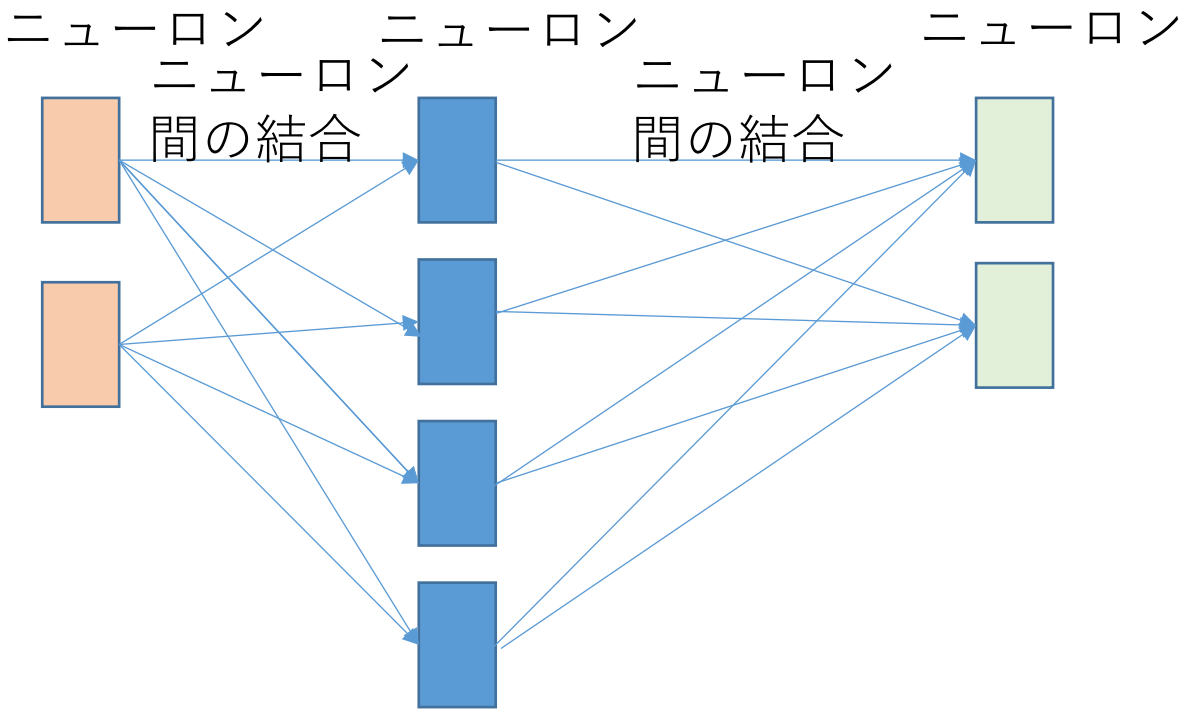
過学習に関するまとめ

- **過学習**は，**訓練データへの過剰適合により発生**
- 訓練データから，実在しない特徴やパターンを発見してしまう.
- **防止**には様々な**技術，調整，巨大な訓練データが必要**
- 防止は完全には不可能であるため，**検証データ**による**検証**（過学習が無いことの確認）が重要

6-4. ニューラルネットワークの基本構造とフォワードプロパゲーション

ニューラルネットワークの仕組み

ニューラルネットワークは，**入力の重みづけ，合計とバイアス，活性化関数の適用**を行う**ニューロン**が**ネットワーク**を形成する



入力の重みづけ，合計とバイアス，活性化関数の適用

1. **入力の重みづけ**: **各入力に対応する重み**を掛ける．例：
   0.3×**0.1**　**-0.5**×**0.8**　**0.2**×**-0.5**
2. **合計とバイアス**: **重みづけした値の合計にバイアス**（数値）を加える．例 **0.03　-0.4　-0.1 の合計は -0.47．これに 0.2 を加える**
3. **活性化関数の適用**: 結果に，**活性化関数**を適用し，**活性度を決定**（**活性度が高いほどニューロンが強く反応している**ことを示す→次のニューロンの入力になる）

0.3　× 0.1　⇒ 0.03

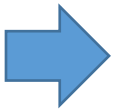
-0.5　× 0.8　⇒ -0.4

0.2　× -0.5　⇒ -0.1

入力　重み

合計
-0.47

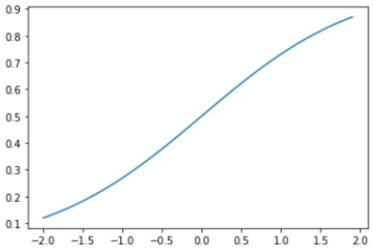
バイアス 0.2



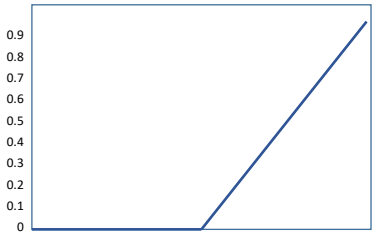
-0.27 に
活性化関数
を適用

活性化関数

- **ニューロン**の**活性度を決定**する関数
- **入力の重みづけ**，**合計とバイアス**の**結果**に，**活性化関数を適用**し，ニューロンの活性度を決定
- **ReLU**や**シグモイド**など，さまざまな種類



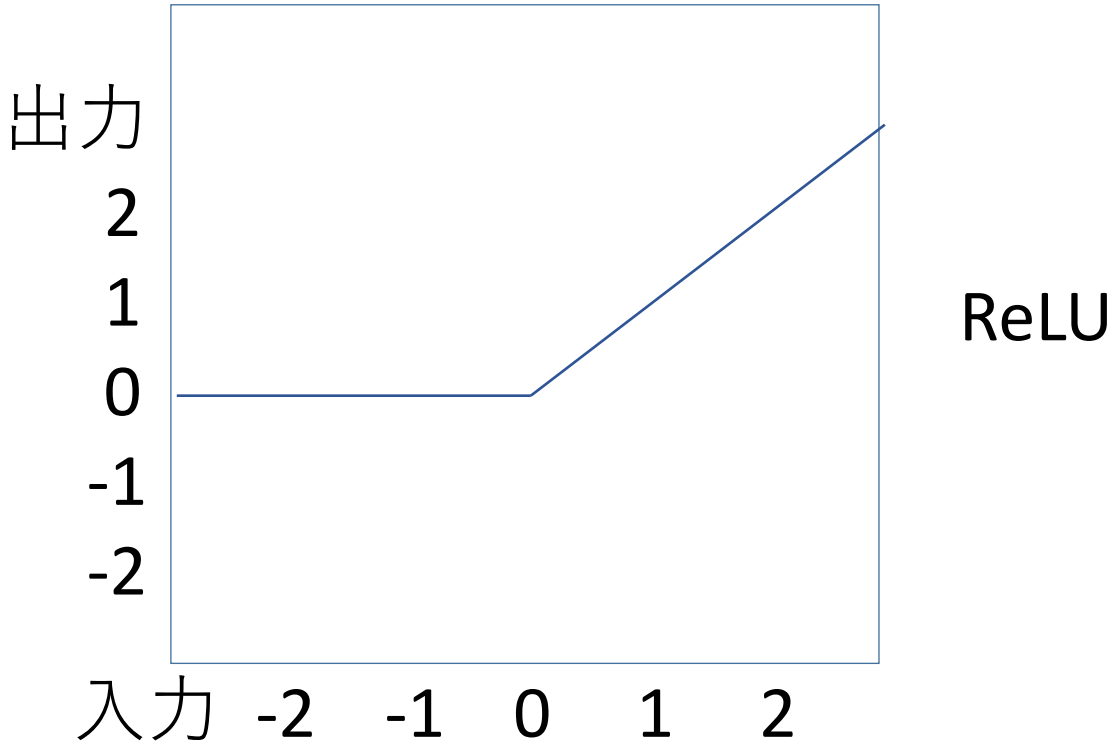
シグモイド



ReLU

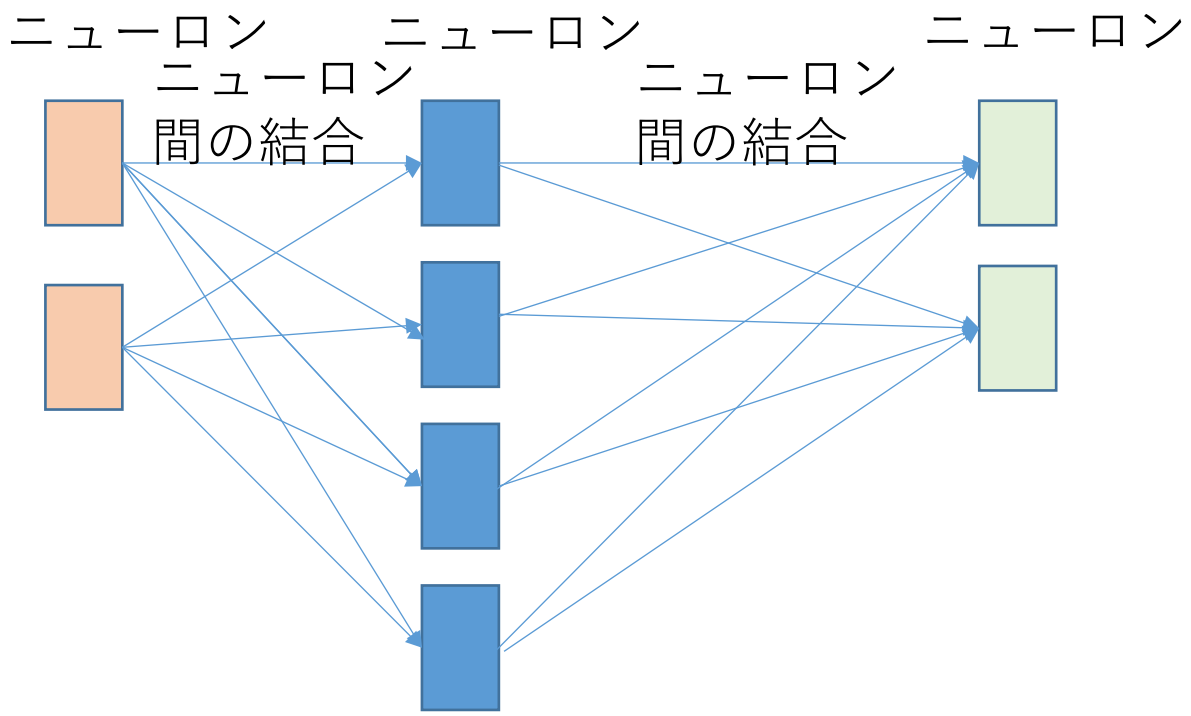
ReLU

- **2011年に提案**された**活性化関数**
- 入力が**0以下の場合は0を返し，0より大きい場合はその値をそのまま返す**特性を持つ
- シンプルな構造ながら高い性能を示し，現代のディープラーニングで広く使用されている．



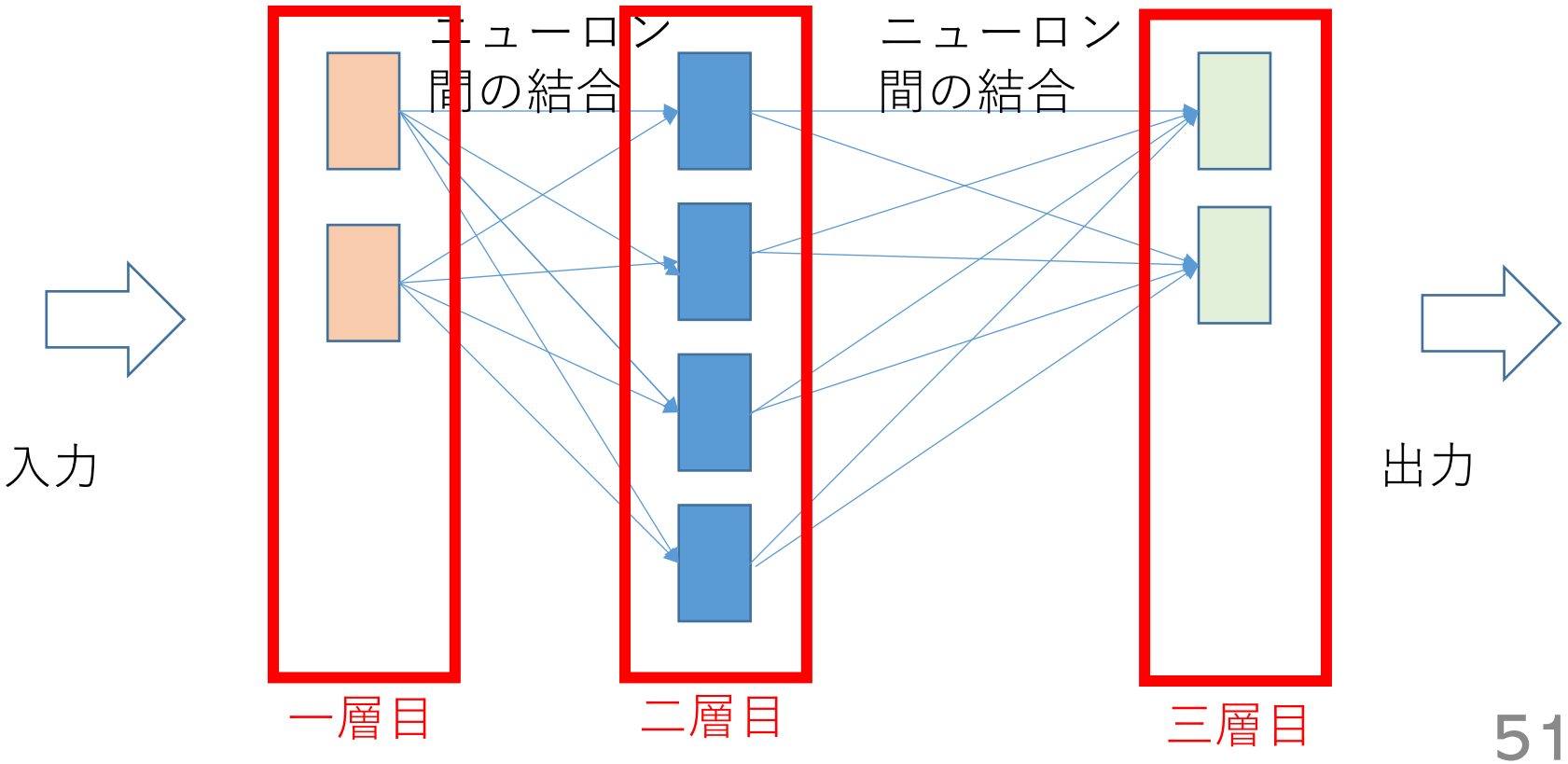
フォワードプロパゲーション

- **フォワードプロパゲーション**は，**データ**が**入力層**から**出力層**へと**順方向に伝播**する**基本的な処理過程**である.
- 各層のニューロンは，**前層からの入力を処理**し，その**結果を次層に伝達**する.



ニューラルネットワークの層構造

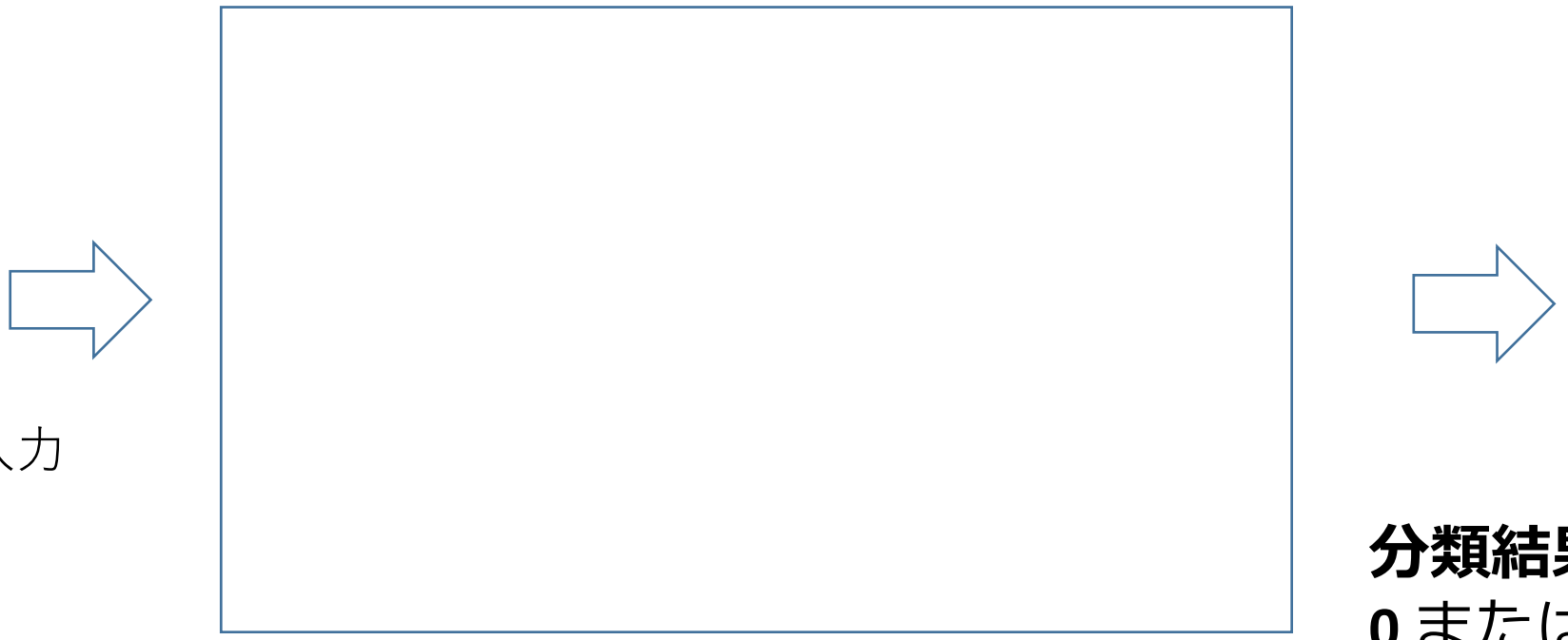
ニューラルネットワークの基本構造であり，**複数の層**から成る．**データは入力から出力への一方向**に流れ，**各層**は**同じ種類のニューロンで構成**される．



6-5. ニューラルネットワークを用いた分類

３種類の中から１つに分類する場合

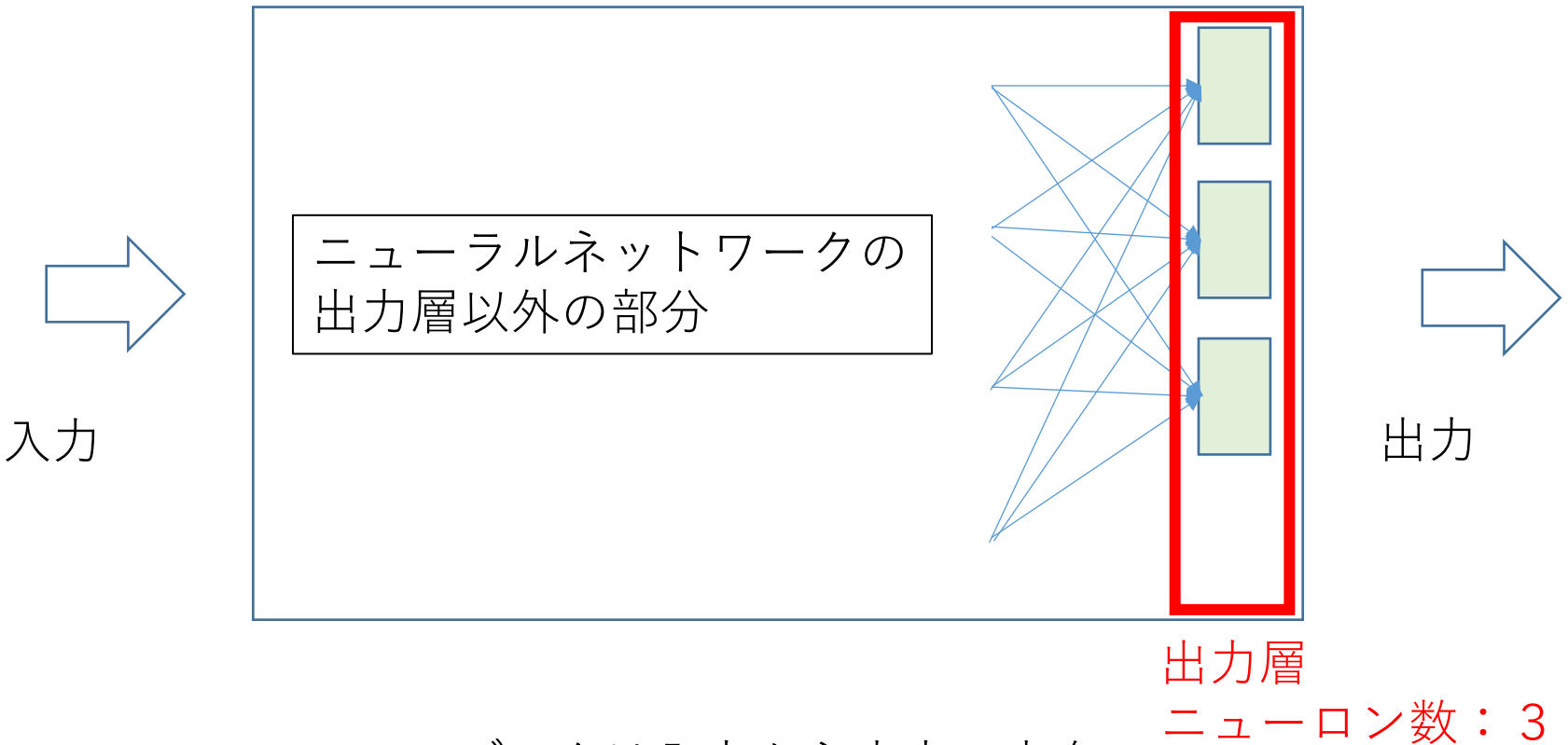
- 入力データを受け取る
- 分類結果として0または1または2を出力する



分類のためのネットワーク構造

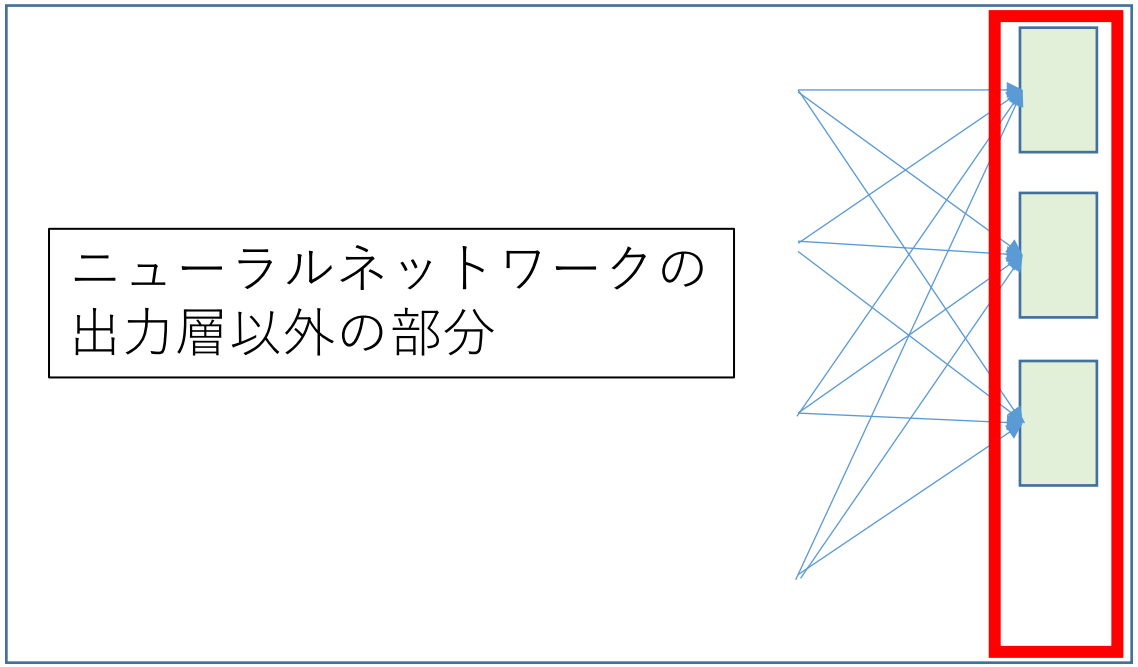
- データは入力から出力の方向へ流れる
- 出力層のニューロン数は3（分類する種類の数）

出力層のニューロン数：**3**　にする　　ニューロン



出力層について，１つが強く
活性化するように調整

ニューロン



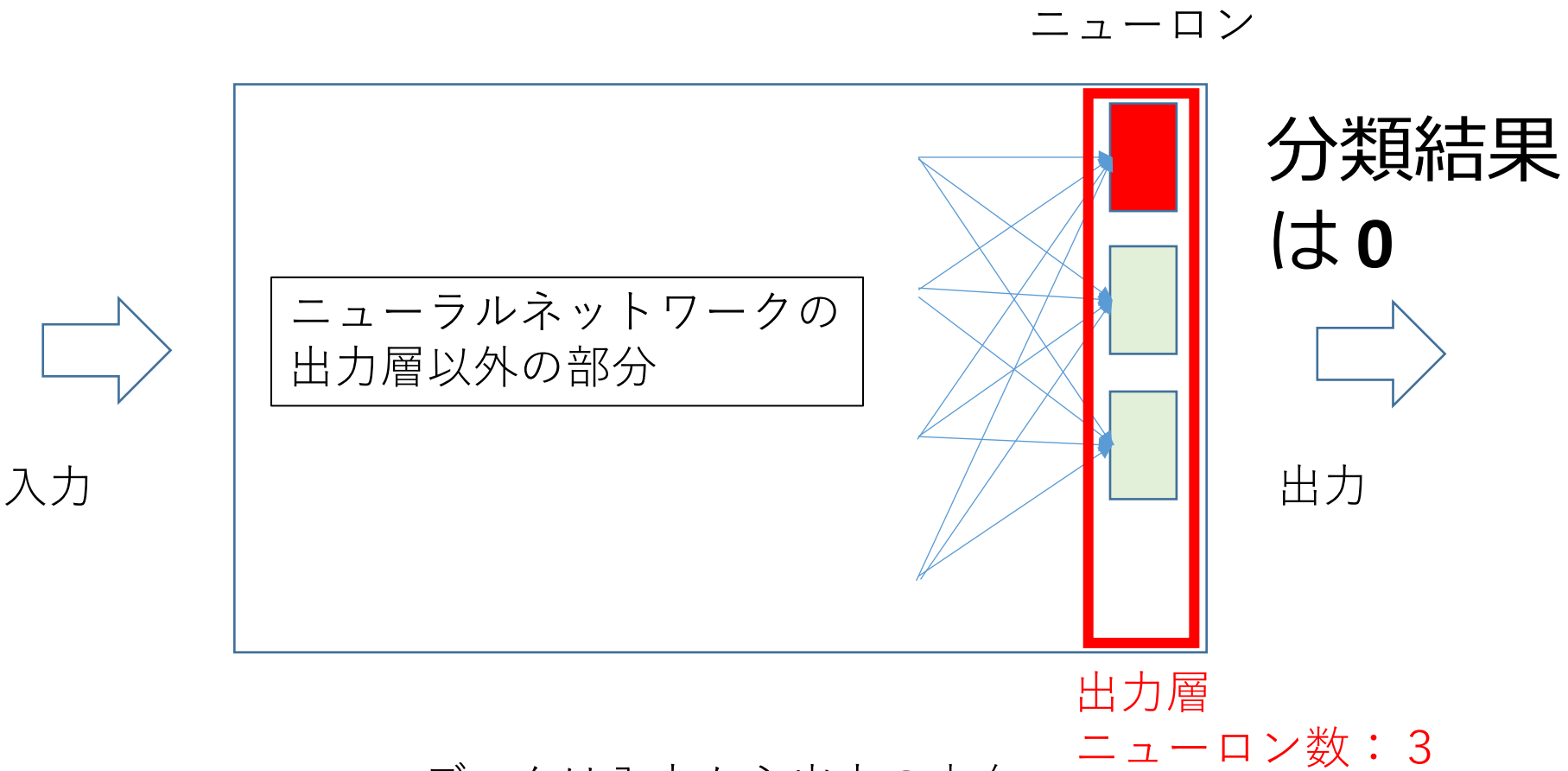
入力

出力

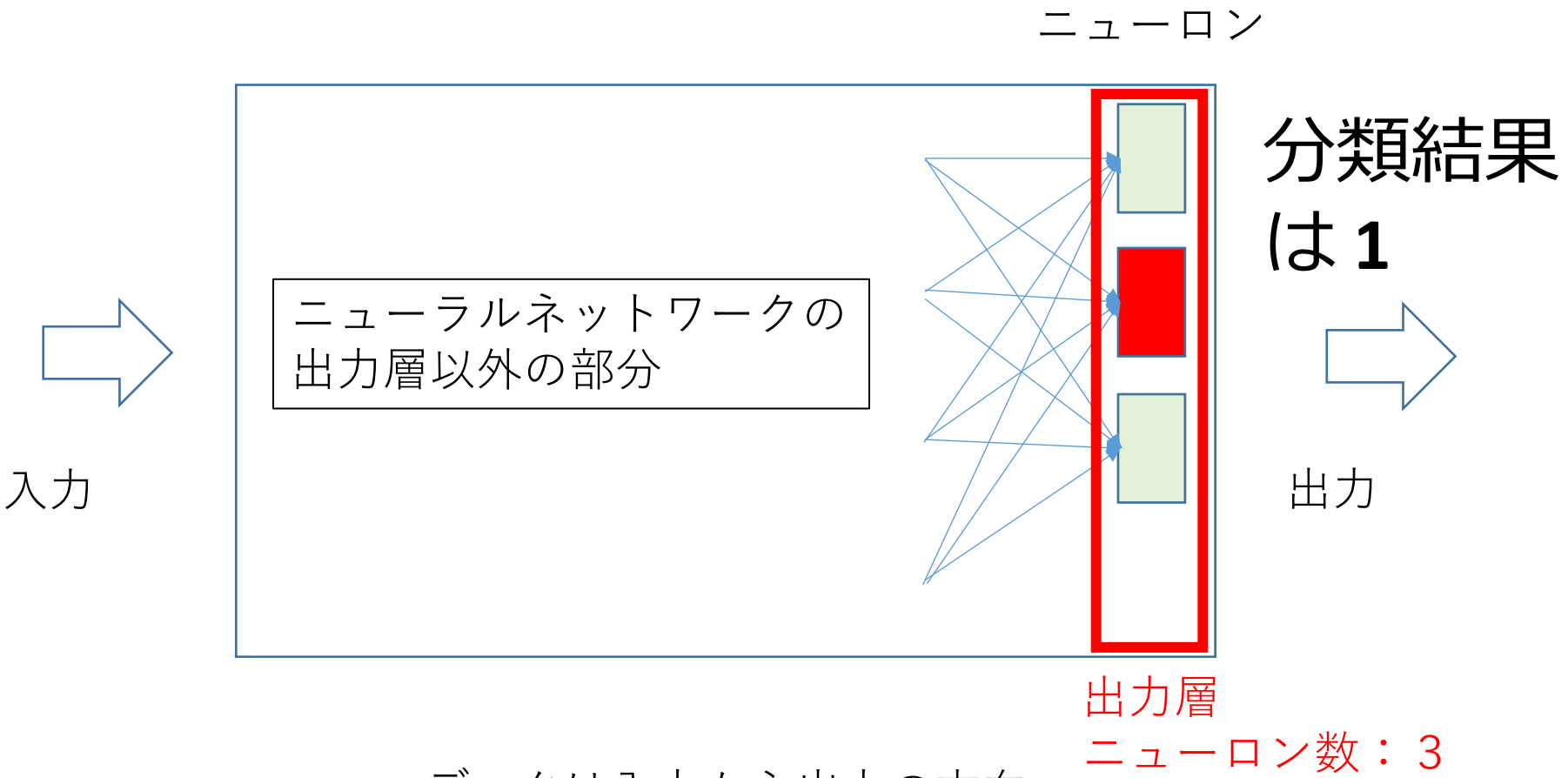
出力層
ニューロン数：３

データは入力から出力の方向へ

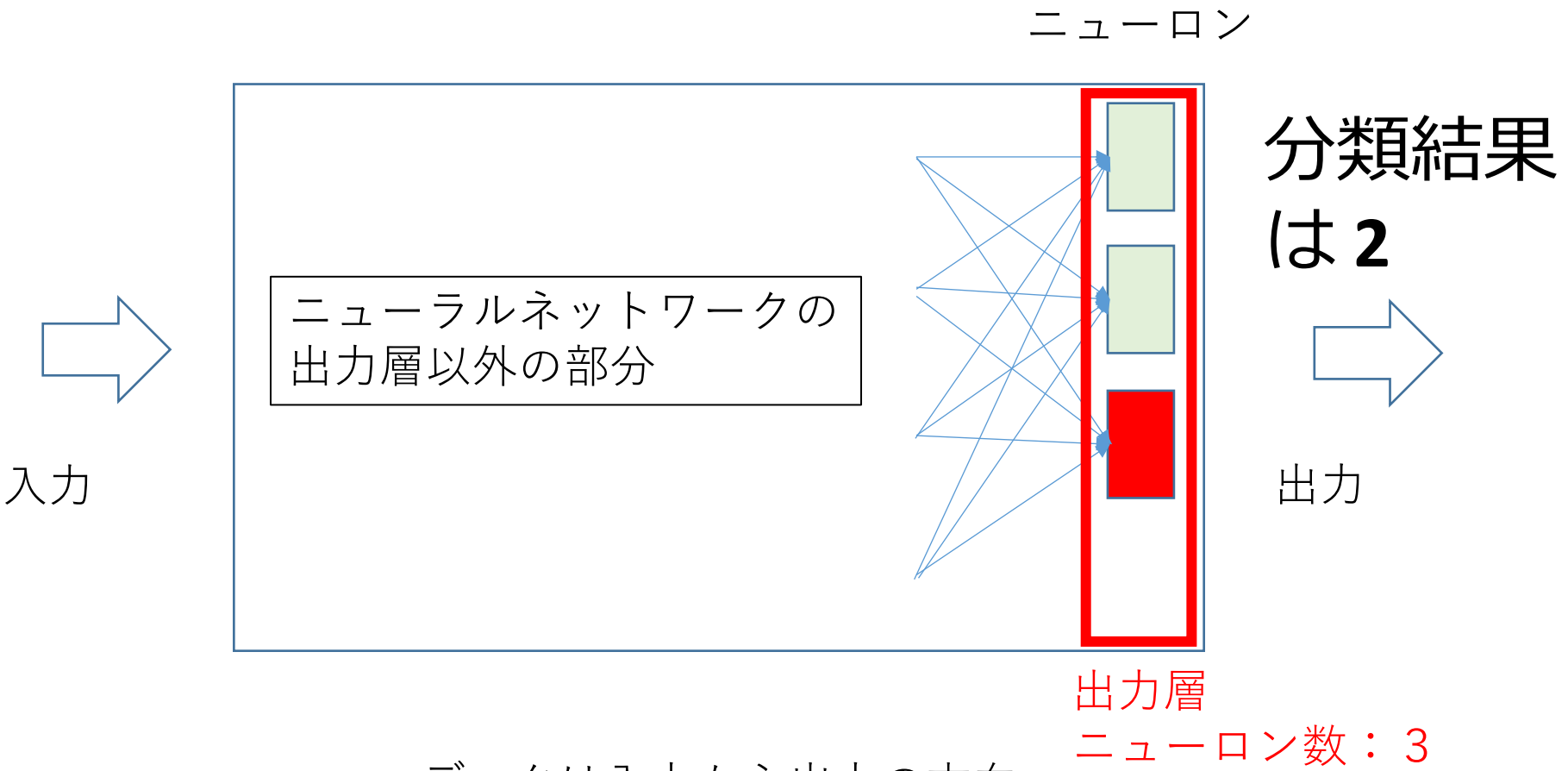
最も強く活性化したニューロンの番号が分類結果となる
（分類結果は0，1，2のいずれかとなる）



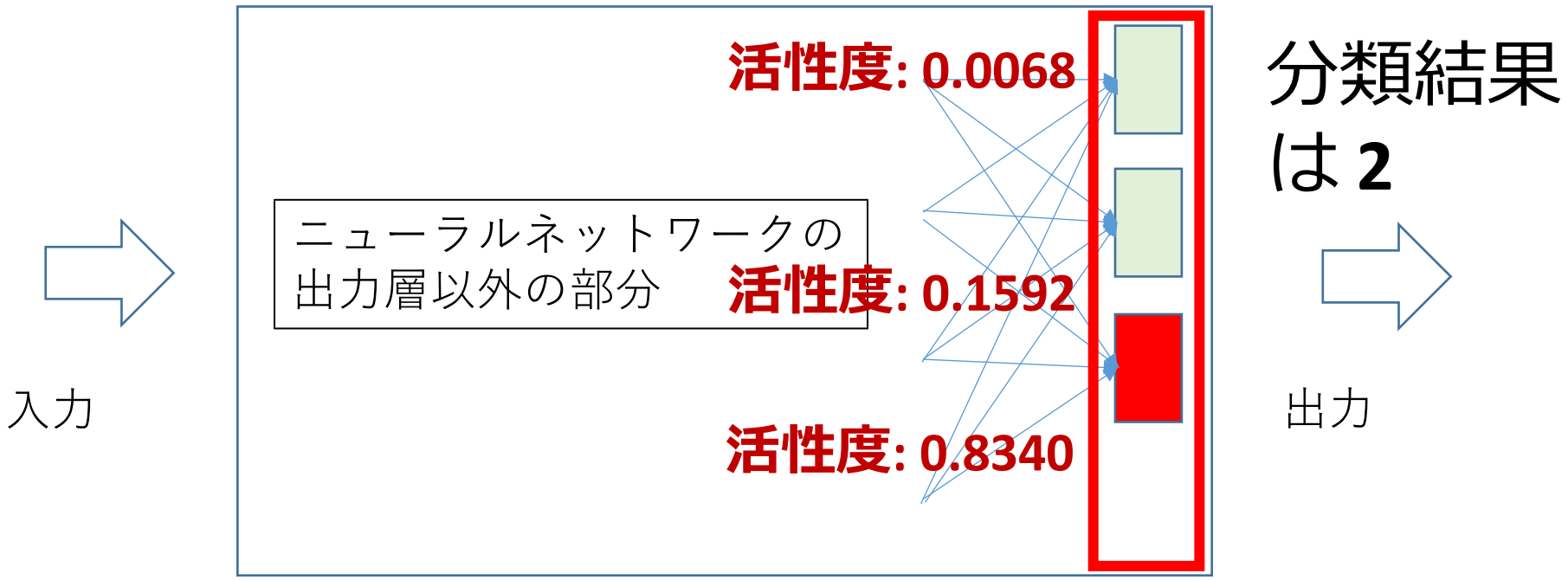
最も強く活性化したニューロンの番号が分類結果となる
（分類結果は0，1，2のいずれかとなる）



最も強く活性化したニューロンの番号が分類結果となる
（分類結果は0，1，2のいずれかとなる）



- 活性度は数値（例：0 から 1 の数値）
- **最も活性度の値が高いもの**が選ばれて**分類結果**となる
  例：活性度: 0.0068 活性度: 0.1592 活性度: 0.8340
  → 最も高い0.8340に対応する2が分類結果



ニューラルネットワークを用いた分類のまとめ

- 出力層の**最も活性度の高いニューロン**が**分類結果を決定**
- 誤差が存在する可能性がある

演習４　ニューラルネットワークを用いた分類

演習の狙い

未学習のニューラルネットワークで，分類を実行.

- **ネットワーク構造の観察：**
  - 第1層 2ニューロン
  - 第2層 4ニューロン
  - 第3層 3ニューロン（クラス分類用）
- **処理の流れ：**
  - **ランダムに初期化された重みとバイアス**を使用
  - ReLU関数による活性化（第1層，第2層）
  - Softmax関数による分類結果の出力（第3層）

Softmax関数は各ニューロンの活性度（数値）を[0,1]の範囲に変換．さらに，全ての出力の合計が1になるように正規化．この関数により，出力を確率として扱うことができるようになる．分類に適している

プログラムの要点説明

- 3層のニューラルネットワーク
- 重みとバイアスはランダムに初期化
- フォワードプロパゲーションを実行
- 未学習の状態
- **3つの入力データ([0.5,0.1], [0.1,0.8], [0.2,0.3])をクラス0,1,2に分類**する

① trinket の次のページを開く

https://trinket.io/python/44d08b044eb7

② 実行結果が，次のように表示されることを確認

main.py

```
    print()
    return X

def main():
    # サンプルデータの準備
    X = prepare_sample_data()
    y = [0, 1, 2]  # データ1はクラス0、データ2はクラス1、データ3はクラス2

    # ネットワークの初期化と構造表示
    W1, b1, W2, b2 = initialize_weights_and_biases()
    display_network_structure()

    print("===== 学習前の状態 ========================")
    display_weights_and_biases(W1, b1, W2, b2)

    # フォワードプロパゲーション（学習前）
    Z1, A2, Z2, A3 = forward(X, W1, b1, W2, b2)
    print("学習前の活性度:")
    display_layer_activations(X, A2, A3)
    pred = predict(X, W1, b1, W2, b2)
    print("学習前の予測クラス:".format(str(pred)))

    # バックプロパゲーションによる学習（1回）
    W1, b1, W2, b2 = backpropagation(X, y, Z1, A2, Z2, A3, W1, b1, W2, b2)

    print("===== 学習後の状態 ========================")
    display_weights_and_biases(W1, b1, W2, b2)

    # フォワードプロパゲーション（学習後）
    Z1, A2, Z2, A3 = forward(X, W1, b1, W2, b2)
    print("学習後の活性度:")
    display_layer_activations(X, A2, A3)
    pred = predict(X, W1, b1, W2, b2)
    print("学習後の予測クラス: {}".format(str(pred)))

main()
```

Result　Instructions

```
===== 学習後の状態 ========================
重みとバイアス:
1層→2層の重み:
  ニューロン1 →: -0.259 -0.717 0.683 0.783
  ニューロン2 →: -0.936 0.890 0.155 -1.003
2層のバイアス: -0.639 -0.238 -0.748 0.043
()
2層→3層の重み:
  ニューロン1 →: -0.193 0.117 -0.279
  ニューロン2 →: -0.317 -0.313 -0.880
  ニューロン3 →: -0.611 0.739 0.540
  ニューロン4 →: 0.535 0.550 -0.883
3層のバイアス: -0.634 0.704 -0.639
()
学習後の活性度:
第1層のニューロンの活性度（ニューロン2個）:
データ 1: [0.500, 0.100]
データ 2: [0.100, 0.800]
データ 3: [0.200, 0.300]
()
第2層のニューロンの活性度（ニューロン4個）:
データ 1: [0.5, 0.1] -> [0.000, 0.000, 0.000, 0.335]
データ 2: [0.1, 0.8] -> [0.000, 0.402, 0.000, 0.000]
データ 3: [0.2, 0.3] -> [0.000, 0.000, 0.000, 0.000]
()
第3層のニューロンの活性度（ニューロン3個）:
データ 1: [0.5, 0.1] -> [0.184, 0.703, 0.114]
データ 2: [0.1, 0.8] -> [0.178, 0.680, 0.141]
データ 3: [0.2, 0.3] -> [0.172, 0.656, 0.171]
学習後の予測クラス: [1, 1, 1]
```

- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

6-6. ニューラルネットワークの学習

ニューラルネットワークの学習プロセス

- **訓練データ**により**ニューラルネットを動作**させる
- **結果**と**正解**を照合し，**誤差**を得る
- ニューロン間の**結合の重み**と**バイアス**を調整し，**誤差を減らす**

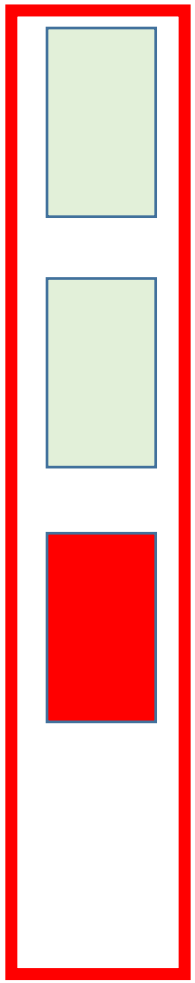
正解と誤差

正解は 2
であるとする

活性度: 0.0068

活性度: 0.1592

活性度: 0.8340



誤差: 0.0068 ⟷ **あるべき値: 0**

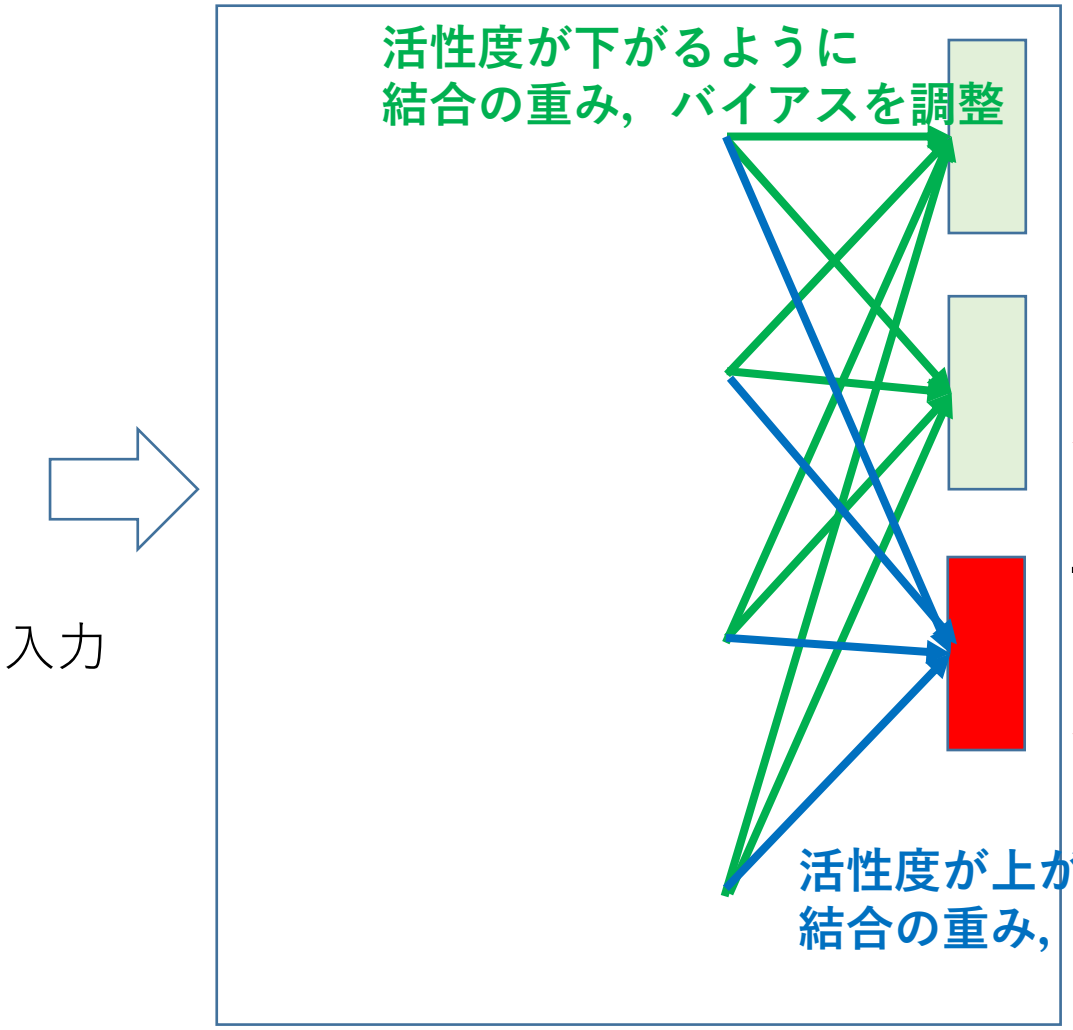
誤差: 0.1592 ⟷ **あるべき値: 0**

誤差: - 0.1760 ⟷ **あるべき値: 1**

誤差をもとに、結合の重みを自動調節

ニューラルネットワークの学習

正解は 2
であるとする

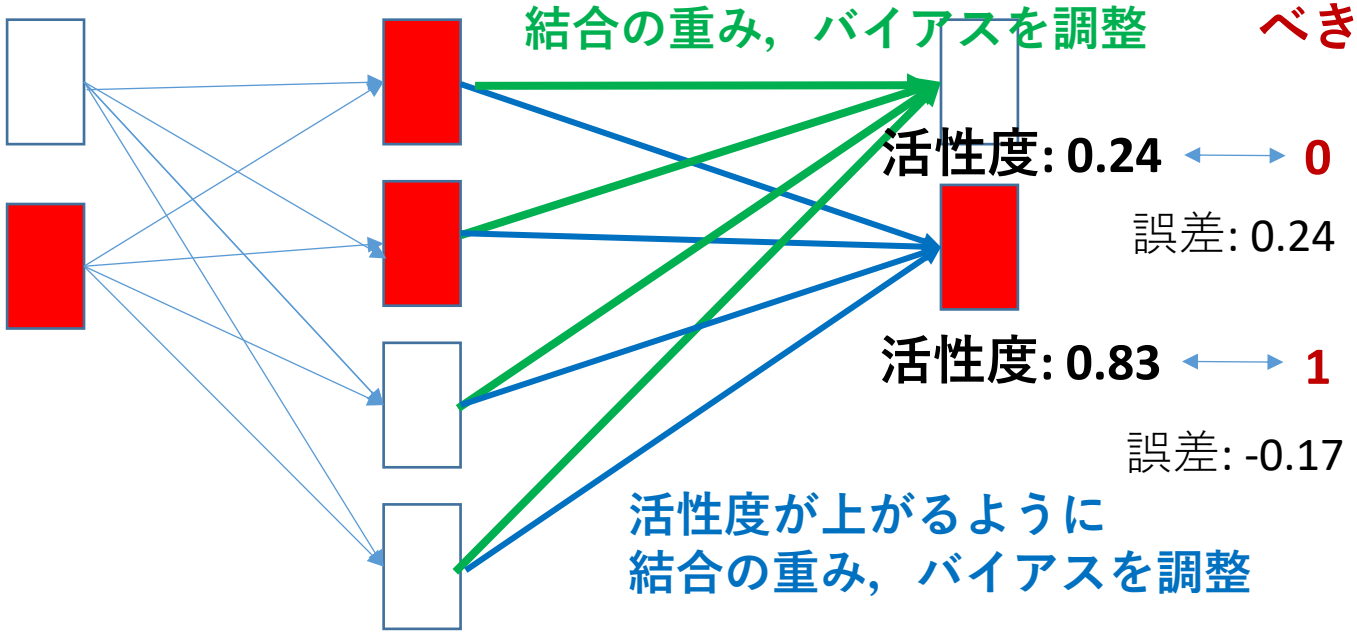


誤差: 0.0068
あるべき値: 0
活性度: 0.0068

誤差: 0.1592
あるべき値: 0
活性度: 0.1592

誤差: - 0.1760
あるべき値: 1
活性度: 0.8340

ニューラルネットワークの動作イメージ

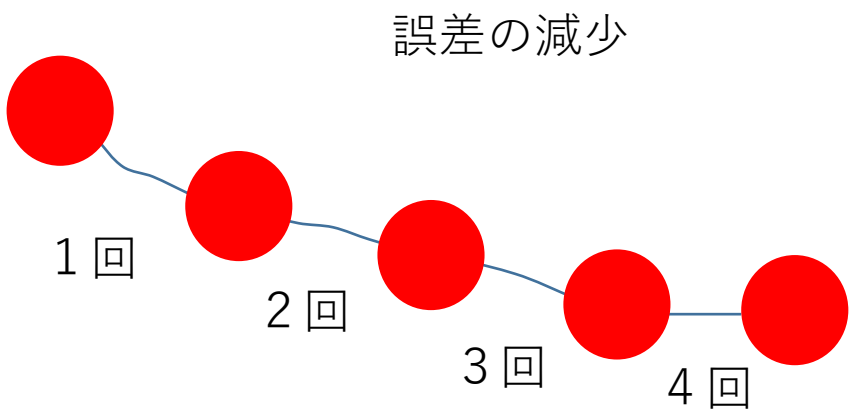


より前の層の結合の
重み，バイアスも調整される

学習の繰り返しの重要性

同じ訓練データを繰り返し使用する

- **訓練データ**を１回使っただけでは，**学習不足**の場合がある.
- 同じ**訓練データ**を**繰り返し**使用することで，誤差をさらに減らす
- 1回，2回，3回，4回と進むにつれて誤差が減少する



演習４　ニューラルネットワークの学習

演習の狙い

バックプロパゲーションの役割理解

- 学習プロセスの観察
- 学習前後の重みとバイアスの変化

プログラムの要点説明

- **バックプロパゲーションの実現**（backpropagation関数）
  - 3層目の誤差計算: 出力と正解の差（delta3）
  - 2層目の誤差逆伝播: 3層の誤差×2-3層間の重みW2とReLU導関数の積（delta2）
  - 重みとパイアスの調整:

    2-3層間の重みW2，3層のバイアスb2

    1-2層間の重みW1，2層のバイアスb1
- 学習効果の可視化
  - 学習前後の重みとバイアスの変化表示
  - 各層のニューロン活性度の変化表示
- **学習回数は１回（１つめ），学習回数１００回（２つめ）の２通り**

学習回数１回

① trinket の次のページを開く

https://trinket.io/python/44d08b044eb7

② 実行結果が，次のように表示されることを確認

main.py

```
    print()
    return X

def main():
    # サンプルデータの準備
    X = prepare_sample_data()
    y = [0, 1, 2]  # データ1はクラス0、データ2はクラス1、データ3はクラス2

    # ネットワークの初期化と構造表示
    W1, b1, W2, b2 = initialize_weights_and_biases()
    display_network_structure()

    print("===== 学習前の状態 ==========")
    display_weights_and_biases(W1, b1, W2, b2)

    # フォワードプロパゲーション（学習前）
    Z1, A2, Z2, A3 = forward(X, W1, b1, W2, b2)
    print("学習前の活性度:")
    display_layer_activations(X, A2, A3)
    pred = predict(X, W1, b1, W2, b2)
    print("学習前の予測クラス:".format(str(pred)))

    # バックプロパゲーションによる学習（1回）
    W1, b1, W2, b2 = backpropagation(X, y, Z1, A2, Z2, A3, W1, b1, W2, b2)

    print("===== 学習後の状態 ==========")
    display_weights_and_biases(W1, b1, W2, b2)

    # フォワードプロパゲーション（学習後）
    Z1, A2, Z2, A3 = forward(X, W1, b1, W2, b2)
    print("学習後の活性度:")
    display_layer_activations(X, A2, A3)
    pred = predict(X, W1, b1, W2, b2)
    print("学習後の予測クラス: {}".format(str(pred)))

main()
```

Result　Instructions

```
===== 学習後の状態 ==========
重みとバイアス:
1層→2層の重み:
  ニューロン1 →: -0.259 -0.717 0.683 0.783
  ニューロン2 →: -0.936 0.890 0.155 -1.003
2層のバイアス: -0.639 -0.238 -0.748 0.043
()
2層→3層の重み:
  ニューロン1 →: -0.193 0.117 -0.279
  ニューロン2 →: -0.317 -0.313 -0.880
  ニューロン3 →: -0.611 0.739 0.540
  ニューロン4 →: 0.535 0.550 -0.883
3層のバイアス: -0.634 0.704 -0.639
()
学習後の活性度:
第1層のニューロンの活性度（ニューロン2個）:
データ 1: [0.500, 0.100]
データ 2: [0.100, 0.800]
データ 3: [0.200, 0.300]
()
第2層のニューロンの活性度（ニューロン4個）:
データ 1: [0.5, 0.1] -> [0.000, 0.000, 0.000, 0.335]
データ 2: [0.1, 0.8] -> [0.000, 0.402, 0.000, 0.000]
データ 3: [0.2, 0.3] -> [0.000, 0.000, 0.000, 0.000]
()
第3層のニューロンの活性度（ニューロン3個）:
データ 1: [0.5, 0.1] -> [0.184, 0.703, 0.114]
データ 2: [0.1, 0.8] -> [0.178, 0.680, 0.141]
データ 3: [0.2, 0.3] -> [0.172, 0.656, 0.171]
学習後の予測クラス: [1, 1, 1]
```

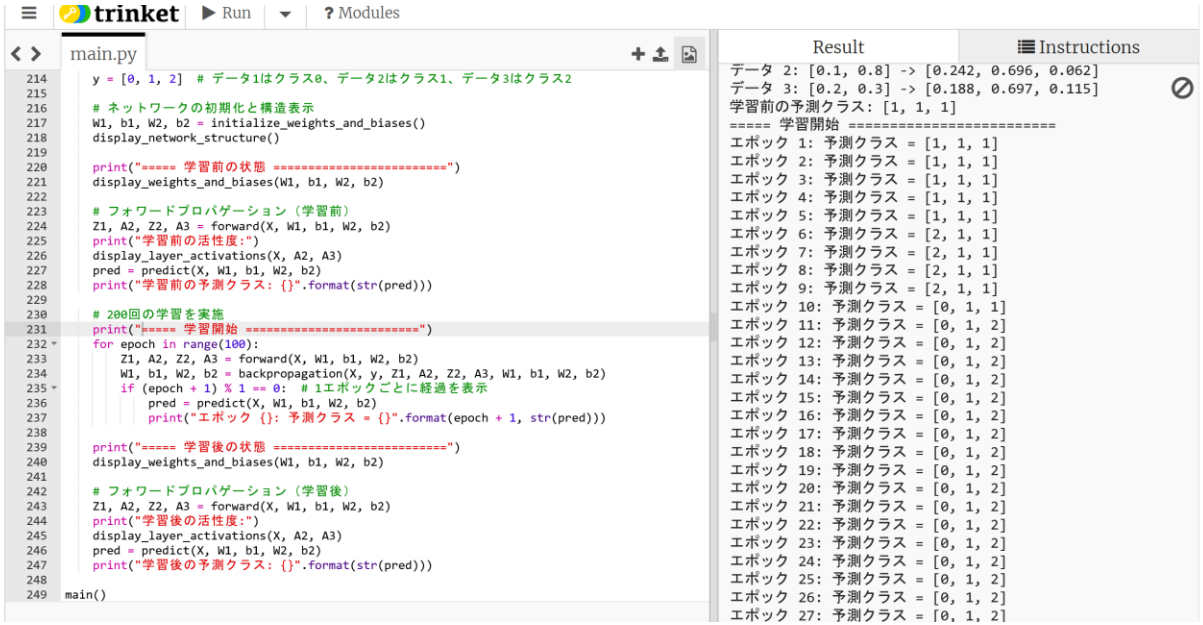
- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能

学習回数１００回

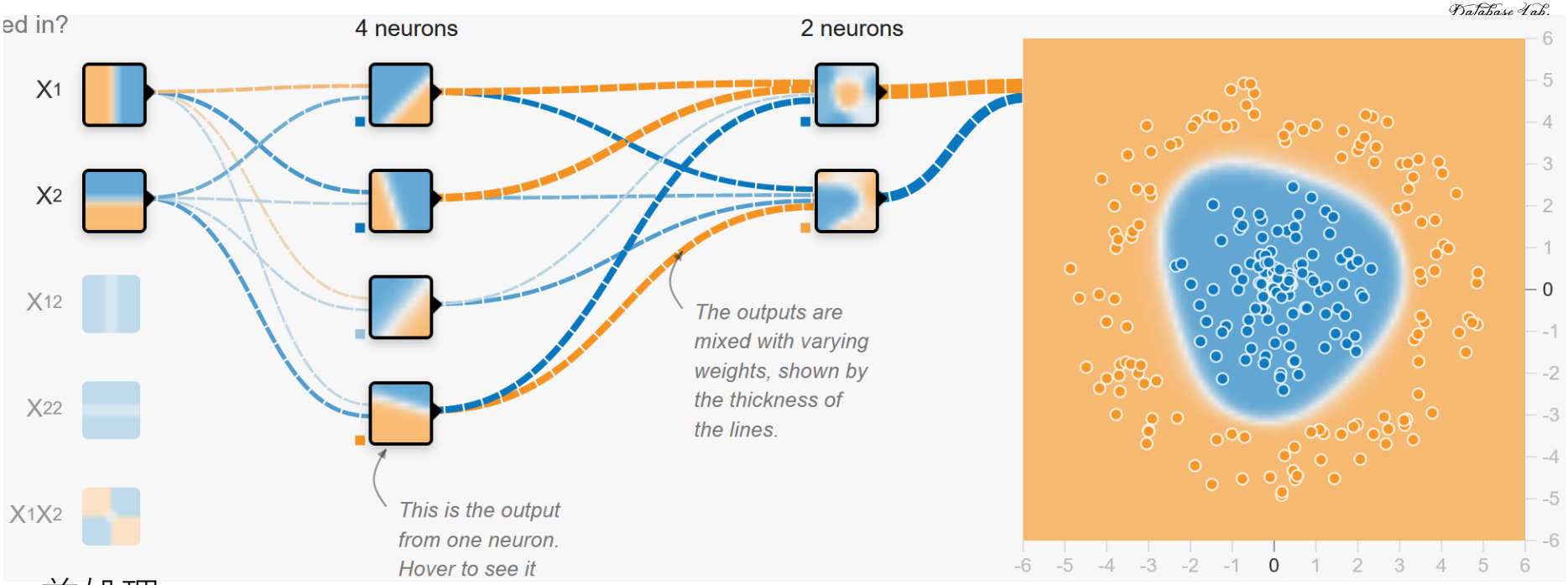
③ trinket の次のページを開く

https://trinket.io/python/de02afe28320

④ 実行結果が，次のように表示されることを確認



- 実行が開始しないときは、「**実行ボタン**」で**実行**
- ソースコードを**書き替えて再度実行**することも可能



前処理
（データが青い部分にあれば活性化）

データが中央にあれば活性化

→結合→ 1層目 →結合→ 2層目

ニューラルネットワーク

URL: https://playground.tensorflow.org/

ニューラルネットワークのまとめ

- **ニューラルネットワーク**は，単純な原理で動作
- **ニューロン**は**複数の入力**を受け取り，**結合の重み**，**バイアス**，**活性化関数**を用いた処理を行う.
- **フォワードプロパゲーション：入力から出力の方向へ一方向にデータが流れる**.
- **結合の重みとバイアスを調整（誤差の最小化）**して，**望みの出力を実現**.

データを持ち他学習により，結合の重みとバイアスを適切に調整