演習：ニューラルネットワークによる分析・分類

2 次元点の分類を例にした Google Colaboratory と Keras の演習

Google Colab ファイル：**06_2 次元点の分類.ipynb**

2 次元点の分類（線形分割）

学習データ：2Dcrd2Classes_train.csv

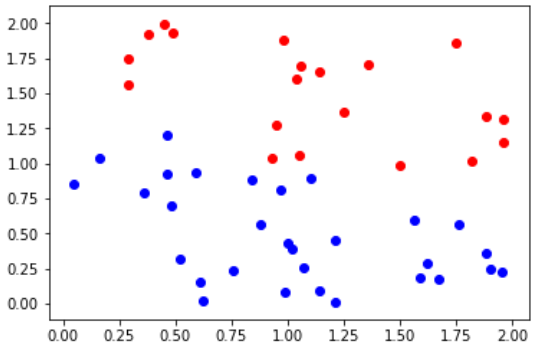
予測データ：2Dcrd2Classes_test.csv

■ 2Dcrd2Classes_train.csv の例

✓ 1 行目は項目名

✓ x 座標と y 座標、その点のラベル（ここでは 0 or 1）

x	y	label
0.88	0.56	1
1.9	0.25	1
1.21	0.01	1
0.52	0.32	1
1.96	1.31	0
0.36	0.79	1
1.76	0.56	1
0.29	1.75	0
0.76	0.24	1



データの読み込みと描画

```
---
import pandas as pd
import matplotlib.pyplot as plt

datadir = "DataFolder" # データがあるフォルダ
csv_train = datadir+"2Dcrd2Classes_train.csv" # データファイルのパス
df_train = pd.read_csv(csv_train)
train_data=df_train.iloc[:,:2] # 先頭列から 2 列分
train_labels=df_train.iloc[:,2] # 2 列目


# 学習データのプロット
for i in range(len(train_labels)):
  if train_labels[i]==0:
    plt.scatter(train_data.iloc[i,0], train_data.iloc[i,1], color="r")
  else:
    plt.scatter(train_data.iloc[i,0], train_data.iloc[i,1], color="b")
plt.show()---
---
```

単純パーセプトロン（とロジスティック回帰）による分類

```
---
import pandas as pd
import tensorflow as tf
import tensorflow.keras as krs
import matplotlib.pyplot as plt

datadir = " DataFolder "
csv_train = datadir+"2Dcrd2Classes_train.csv"
df_train = pd.read_csv(csv_train)
train_data=df_train.iloc[:,:2] # 先頭列から 2 列分
train_labels=df_train.iloc[:,2] # 2 列目

csv_valid = datadir+"2Dcrd2Classes_test.csv"
df_valid = pd.read_csv(csv_valid)
valid_data=df_valid.iloc[:,:2]
valid_labels=df_valid.iloc[:,2]

# DNN の定義
model = krs.Sequential()
model.add(krs.layers.Dense(1,input_shape=(train_data.shape[1],)))
#このレイヤーを追加するとロジスティック回帰
#model.add(krs.layers.Activation("sigmoid"))

# モデルの構築
model.compile(loss='mean_squared_error', optimizer='sgd',
      metrics=['mean_absolute_error'])

# エポック数
EPOCHS = 10

# 学習と履歴の保存 verbose は 0,1,2 を取り、0:表示なし、1:プログレスバー、2:詳細
history = model.fit(train_data,train_labels,epochs=EPOCHS,batch_size=25,
validation_data=(valid_data,valid_labels), verbose=0)

# print(history.history)

# 学習過程の Accuracy の値の描画
plt.plot(history.history['mean_absolute_error'])
plt.plot(history.history['val_mean_absolute_error'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# 学習データのプロット
for i in range(len(train_labels)):
  if train_labels[i]==0:
    plt.scatter(train_data.iloc[i,0], train_data.iloc[i,1], color="r")
  else:
    plt.scatter(train_data.iloc[i,0], train_data.iloc[i,1], color="b")
plt.show()


# 学習モデルによる valid_data のプロット
```

分岐などが無い連続的に層を連ねる DNN を構築

第 1 層目には入力データの形を明示する。
train_data.shape の出力を確認してみてください。

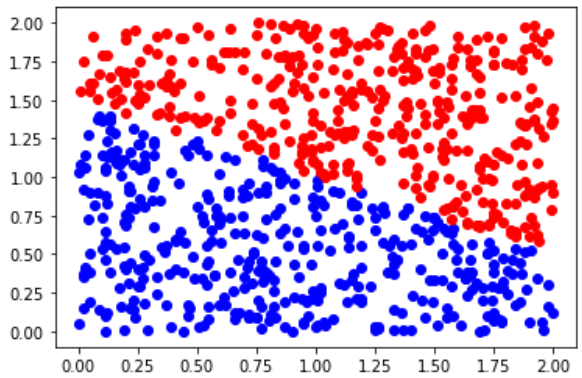
```
for i in range(len(valid_labels)):
  if valid_labels[i]==0:
    plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="r")
  else:
    plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="b")
plt.show()

# 学習モデルによる valid_data の分類
result = model.predict(valid_data)


# 学習モデルによる valid_data の分類結果に基づいたプロット
for i in range(len(valid_labels)):
  if result[i] < 0.5:
    if valid_labels[i] == 0:
      plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="r")
    else:
      plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="m", marker="x")
  else:
    if valid_labels[i] == 1:
      plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="b")
    else:
      plt.scatter(valid_data.iloc[i,0], valid_data.iloc[i,1], color="c", marker="x")
plt.show()
```

- 本来は活性化関数にステップ関数を使いますが、ステップ関数が無いので活性化関数を使わずにニューラルネットワークを構成します。
- 損失関数は「mean_squared_error」(平均二乗誤差)にします。

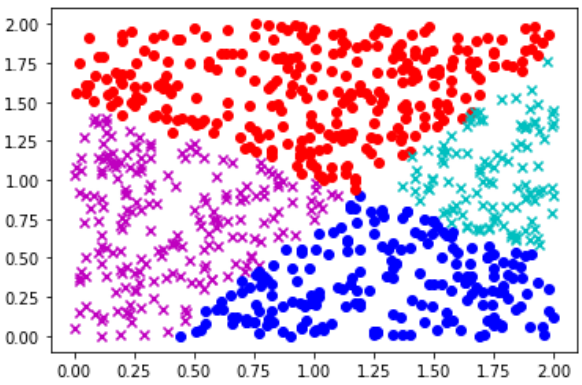
- 予測値が 0.5 未満なら 0, 0.5 以上なら 1 とします。
- 活性化関数にステップ関数を使っていないため、回帰のように連続値になり、0 未満や1より大きな数値になることもあります。
- 活性化関数にシグモイドを使ったロジスティック回帰では、予測値は 0～1 となり、それぞれのラベルに属する確率とみなすことができます。

はじめは epoch = 10 なので、以下のようにはきれいに分類できません。もう少し epoch を増やして様子を見てみましょう。

valid_data 正解



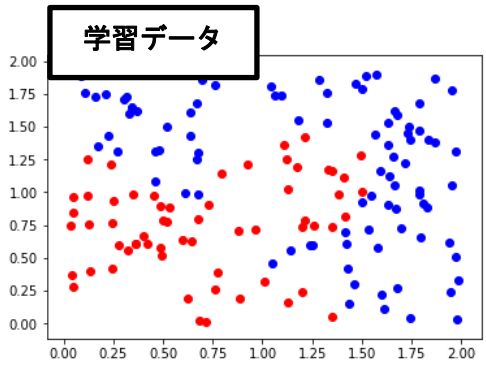
valid_data 予測結果

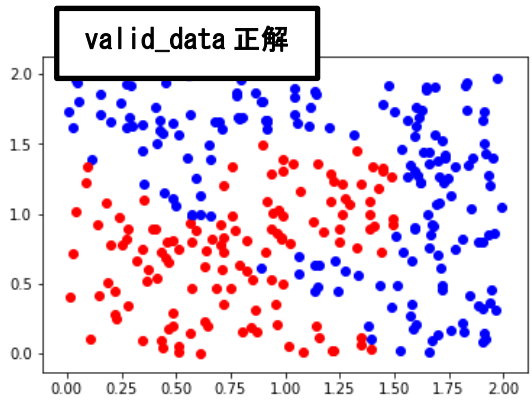


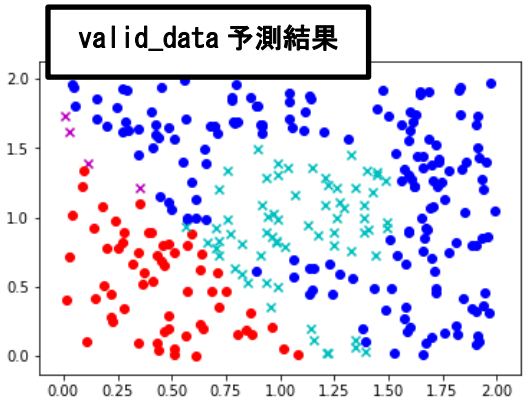
ロジスティック回帰による 2 次元点の分類（非線形分割）

学習データ：2Dcrd2Classes_train_nonliear.csv
予測データ：2Dcrd2Classes_test_nonliear.csv

■　このようなデータは単純なロジスティック回帰では線形にしか分類できないことを確認してください。







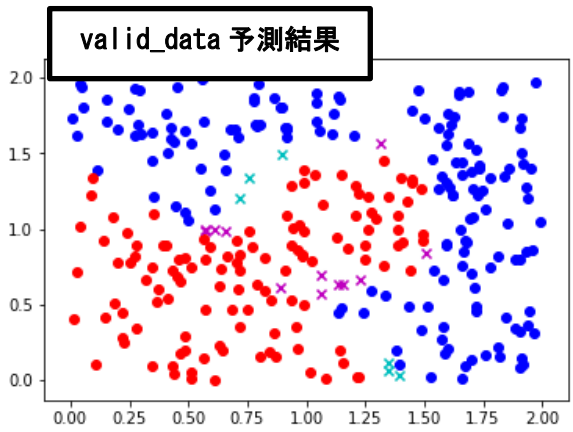
DNN による 2 次元点の分類（非線形分割）

学習データ：2Dcrd2Classes_train_nonliear.csv
予測データ：2Dcrd2Classes_test_nonliear.csv

■　変更部分のみ

```
# DNN の定義
model = krs.Sequential()
model.add(krs.layers.Dense(256,input_shape=(train_data.shape[1],)))
model.add(krs.layers.Activation("tanh"))
model.add(krs.layers.Dense(128))
model.add(krs.layers.Activation("tanh"))

# 全結合層と活性化関数のセットはこの書き方もできる
model.add(krs.layers.Dense(64, activation="tanh"))
model.add(krs.layers.Dense(8, activation="tanh"))
model.add(krs.layers.Dense(1, activation="sigmoid"))
```

■ ディープラーニング（多層のニューラルネットワーク）では非線形の分類が可能なことを確認してください。



構築したニューラルネットワークの概要表示

```
model.summary()
```

```
---
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 256)               768
_________________________________________________________________
activation_8 (Activation)    (None, 256)               0
_________________________________________________________________
dense_17 (Dense)             (None, 128)               32896
_________________________________________________________________
activation_9 (Activation)    (None, 128)               0
_________________________________________________________________
dense_18 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_19 (Dense)             (None, 8)                 520
_________________________________________________________________
dense_20 (Dense)             (None, 1)                 9
=================================================================
Total params: 42,449
Trainable params: 42,449
Non-trainable params: 0
___
```

■ 入力レイヤーは 3 ノード（x、y、バイアス）です。
■ dense_??などレイヤー番号は異なる場合があります。
■ Output Shape はそのレイヤーの出力数（ノード数）です。
■ Param # は NN の重み数です。
  ➢ dense_16 は 256 ノード。入力レイヤーと dense_16 の間のパラメータ数は 3×256=768。
  ➢ dense_17 は 128 ノード。このレイヤーへの入力は前のレイヤーのノード数にバイアスの 1 を足した数となるので、パラメータ数は 257×128=32896。以下同様。

DNN による多次元データの分類

Google Colab ファイル：06_多次元データの分類.ipynb

データ： 2017 年の気象データ
気象データは Humidity（湿度）、Pressure（気圧）、Temperature（気温）、Wind_direction（風向）、Wind_speed（風速）、Weather（天気）の 6 項目となっています。天気は ID 化されています。（晴れ：0、曇り：1、雨：2、雪：3）

学習データ：Denver_train.csv
予測データ：Denver_test.csv

Humidity (%)	Pressure (hPa)	Temperature (K)	Wind_direction (angle)	Wind_speed (m/s)	ID
69	1016	273.66	350	4	3
24	1021	299.93	151	1	0
61	1038	267.1295	302	4	3
44	1016	288.62	240	1	0
70	1043	259.8943333	245	1	3

■ デンバーの天気データで，湿度，気温，気圧，風向，風速から天気をどのくらい分類できるかを試してみてください。
- ✓ レイヤーを増やす
- ✓ ユニット数を変える
- ✓ Dropout や BatchNormalization レイヤーを使う。
- ✓ データを標準化する

また、ここでは分類モデル生成し、学習済みモデルをファイルとして保存するプログラムと、学習済みモデルを読み込んで予測するプログラムを別にするようにしています。

学習モデル生成部分

■8～17 行目

```
datadir = " DataFolder "
csv_train = datadir+"Denver_train.csv"
df_train=pd.read_csv(csv_train) #csv ファイル読み込み
train_data=df_train.iloc[:,:5] # 5 列分（0～4 列目）を train_data とする
train_labels=df_train.iloc[:,5] # 5 列目を train_labels とする

csv_valid = datadir+"Denver_test.csv"
df_valid=pd.read_csv(csv_valid)
valid_data=df_valid.iloc[:,:5]
valid_labels=df_valid.iloc[:,5] ---
```

予測に使うデータは 5 列分（0～4 列目）、天気のラベル（正解ラベル）は 5 列目なので、train_data と train_label、valid_data と valid_label にそれぞれ分割します。

■19～29 行目

```
# pandas のメソッドを利用して標準化
# # pandas の mean, std メソッドはそれぞれ列ごとの平均や標準偏差を計算する
# average = train_data.mean()
# stdev = train_data.std()
```

```
# train_data = (train_data - average)/stdev
# valid_data = (valid_data - average)/stdev
# # 予測時にデータを標準化するときに同じ平均値と標準偏差を使うために値を記録しておく
# average = np.array(average, dtype = "double") # params を numpy の array 形式に変換
# np.save("average.npy",average) # numpy 形式で書き出し
# stdev = np.array(stdev, dtype = "double")
# np.save("stdev.npy", stdev)
---
```

最初はコメントアウトしています。この部分は学習データの標準化を行う部分です。列ごとに平均値と標準偏差を求め、(値 － 平均値)÷標準偏差としています。標準化していないデータと標準化したデータで結果を比較してみてください。

■31～33 行目

```
---
num_classes=4
train_labels=krs.utils.to_categorical(train_labels,     num_classes=num_classes)
valid_labels=krs.utils.to_categorical(valid_labels,
      num_classes=num_classes)
---
```

学習に使う正解ラベルは整数値として読み込まれているので、One-hot 形式に変換しています。
print(train_labels.shape)で行列の形を確認したり、print(train_labels)で実際にどのような値が記録されているか確認してみましょう。

■38～55 行目

```
---
model = krs.Sequential()
model.add(krs.layers.Dense(256,input_shape=(train_data.shape[1],)))
model.add(krs.layers.Activation("sigmoid"))

model.add(krs.layers.BatchNormalization())
model.add(krs.layers.Dense(128, activation="sigmoid"))
model.add(krs.layers.BatchNormalization())
model.add(krs.layers.Dense(128, activation="sigmoid"))
model.add(krs.layers.Dropout(0.5))
model.add(krs.layers.BatchNormalization())
model.add(krs.layers.Dense(64, activation="sigmoid"))
model.add(krs.layers.Dense(8, activation="sigmoid"))
model.add(krs.layers.Dense(num_classes, activation="softmax"))

# モデルの構築
model.compile(loss='categorical_crossentropy',
              optimizer="adam",
              metrics=['acc'])
---
```

DNN の定義部分です。天気の ID は 0～3 の整数値なので 4 クラスの分類となります。損失関数は categorical_crossentropy、DNN の最後の活性化関数は Softmax、その直前の Dense レイヤーのノード数は 4 となります。
metrics には acc(accuracy)を指定します。これは正解の割合で、分類問題に使います。

■74 行目

```
---
model.save("Weather_predict_model.h5", include_optimizer=False)
---
```

ここでは学習済みモデルを一度ファイルとして保存し、別のプログラムで予測をすることにしています。

予測部分

■14～17 行目

```
---
# # 標準化するときは次の部分をコメントアウト解除
# average = np.load("average.npy")
# stdev = np.load("stdev.npy")
# valid_data = (valid_data - average)/stdev
---
```

標準化データで学習した場合は、予測用のデータも標準化します。

■24～30 行目

```
---
model = krs.models.load_model("Weather_predict_model.h5", compile = False)

# モデルの構築
optimizer = tf.train.AdamOptimizer(0.001)
model.compile(loss='categorical_crossentropy', optimizer=optimizer,
              metrics=['acc'])
---
```

ファイルに書き出した学習済みモデルを読み込みます。TensorFlow 1.x の Keras では compile 情報を書きだしたモデルに含めることができないので、再定義します。TensorFlow 2.x では compile 情報も含めることができるようになっています。なお、現在 Colab では ver 2.x が使用できるので上記部分はコメントアウトしています。

■33～36 行目

```
---
test_predictions = model.predict(valid_data)
print(test_predictions[30])
result =np.argmax(test_predictions,1)
print(result[30])
---
```

model.predict(予測用データ)で予測ができます。categorical_crossentropy では、予測結果は、

```
[9.0812296e-01 7.7803515e-02 1.4065598e-02 7.9311776e-06]
```

のように、各ラベルに属する確率として出力されます。このうち最大値を持つインデックスがそのデータが属するラベルになります。
np.argmax で最大値を持つインデックスを取得できます。第 2 引数は 0:列方向、1:行方向の指定です。

（オプション）DNN による多次元データの回帰

データ：ボストンの住環境と住宅価格
学習データ：house_cost_train.csv
予測データ：house_cost_test.csv

住環境データは次の値です。

1. 1 人あたりの犯罪率
2. 25,000 平方フィート以上の敷地の住宅地の割合
3. 町ごとの非小売業が占める広さの割合
4. チャールズ川境界線をあらわすダミー変数（境界線：1、そうでない：0）
5. 酸化窒素濃度（1,000 万分の 1）
6. 住居当たりの平均部屋数
7. 1940 年以前に建設された所有者占有ユニットの割合
8. 5 つのボストン雇用センターに対する加重距離
9. 放射状の高速道路へのアクセス可能性の指標
10. 1 万ドルあたりの固定資産税の税率
11. 町別の生徒と教師比率
12. 1000×（Bk-0.63）^2　Bk：町別の黒人比率
13. Lower status の人口比率
14. 住宅価格

- 住環境データから住宅価格を予測するモデルを試してみてください。
- この場合、1 つの値を予測するため、損失関数は mean_squared_error、DNN の最後の Dense レイヤーのノード数は 1 となる。回帰のため、DNN の最後のレイヤーに活性化関数は使用しない。

MNIST による手書き数字分類

Google Colab ファイル：**06_MNIST 分類.ipynb**
Colab 内に解説とコメントがあるので、ここでは要点のみ解説する。

MNIST データの取り寄せは 12 行目
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

x_train ： 学習用（訓練・教師用）データ
y_train ： 学習用（訓練・教師用）ラベル
x_test ： 検証用データ
y_test ： 検証用ラベル
学習用データは 60,000 個、検証用データは 10,000 個

x_train は多次元リストで、x_train[0]が数字の 5, x_train[1]が数字の 0 と、それぞれ 28x28 の画素データになっている。
元画像が 2 次元画像なので、1 次元変換するが、後で比較するため、x_train を変換したデータを x_train_in とする。同様に、x_test も変換して x_test_in とする。Numpy の reshape を使う次のコードで 2 次元のリストを 1 次元に変換できる。

```
x_train_in = x_train.reshape(mnist_num, 28*28)
x_test_in = x_test.reshape(mnist_test_num, 28*28)
```

ニューラルネットワークの設計は以下の部分。
```
model = krs.Sequential()
model.add(krs.layers.Dense(32, activation='relu', input_shape=(784,))) # 最初の入力
model.add(krs.layers.Dense(16, activation='relu')) # 第 2 層
model.add(krs.layers.Dense(10, activation='softmax'))
```

活性化関数は ReLU 関数を用いる。ReLU は負の値はノイズとして切り捨て、正の値はそのまま活かす。画像の画素値に負の値は存在しないので、画像を扱う場合は ReLU を使用する。また、最後は 0 から 9 までの 10 通りに対する確率のため、softmax 関数を用いる。
しかしこのまま学習回数やニューロンの数、層を増やしても精度は向上しない。そこで、最後に正規化を行う。一般的に、ニューラルネットワークでは入力値が大きくなると損失関数が発散したり振動したりしやすくなる。ここでは 0-255 までの値を 0-1 の範囲に変換する。

```
x_train_nr = x_train # 他データに移す
x_test_nr = x_test
x_train_nr = x_train_nr.astype('float32') # 型変換
x_train_nr /= 255 # 255 で割る
```