

# 【適用事例】3層型ニューラルネットワークを用いた数字認識

## 《学修項目》

- 画像データと対応する教師データの読み込み
- ニューラルネットワークの構成と学習法の設定
- ニューラルネットワークの学習と評価
- ニューラルネットワークの構成・学習法の変更による効果

## 《キーワード》

TensorFlow、数字画像認識、MNIST、最適化アルゴリズム、SGD（確率的勾配降下法）、Adam、クロスエントロピー、識別精度(accuracy)、エポック(epoch)、活性化関数、シグモイド関数、ReLU、過学習、ドロップアウト

## 1. はじめに

ここでは、Google社が開発した機械学習用のプラットフォーム TensorFlow を用いて、数字画像認識を行うプログラムを作成する。作成には、Web上でPythonを記述・実行することができるGoogle Colaboratory を用いる。Webの教材ページに、サンプルプログラムをノートブック形式で掲載するので、そのプログラムを実際に実行しながら、動作確認をすると良い。なお、プログラムを完全に理解するには、PythonやNumpy（Python用の数値計算ライブラリ）の使い方を知る必要があるので、必要に応じて学習されたい。

- [TensorFlow](#)
- [Google Colaboratory](#)

## 2. 画像データと対応する教師データの読み込み

最初にTensorFlowとNumpy のモジュールをインポートし、数字の画像データベースであるMNISTを読み込む。

```
In [26]: # TensorFlow, NumPyモジュールのインポートと数字画像データの読み込み

import tensorflow as tf #TensorFlowモジュールのインポート
import numpy as np      #NumPyモジュールのインポート
mnist = tf.keras.datasets.mnist #数字画像データベースの参照変数の設定
(x_train, y_train), (x_test, y_test) = mnist.load_data() #画像の読み込み
print(type(x_train))    #x_trainのデータ型の確認
print(x_train.dtype)    #x_train(多次元配列)の各要素のデータ型の確認
print(x_train.ndim)     #x_train配列の次元数の確認
print(x_train.shape)    #x_train配列の大きさの確認

<class 'numpy.ndarray'>
uint8
3
(60000, 28, 28)
```

ここで、x\_train, y\_train, x\_test, y\_testのデータ型はNumPyの多次元配列(numpy.ndarray)となる。x\_trainは学習用データで数字画像を表現している3次元配列である。各文字画像は28×28画素であり、各画素は0から255の間の整数(uint8)で表現され（グレースケール画像）、このような文字画像が60,000枚、含まれている。一方、y\_trainは学習用データの教師データを示している1次元配列である。各要素は0から9の間の整数値(uint8)で表現され、全体で60,000個が含まれている。x\_test, y\_testはテスト用データの文字画像とその教師データをそれぞれ示しており、デ

ータ構造は、x\_trainとy\_trainと同様である。ただし、文字画像と教師データの個数はそれぞれ10,000個となっている。（上記プログラムのように多次元配列の各属性値はprint文で確認可能）。

```
In [27]: # 数字画像の各画素値を0.0から1.0の値に正規化
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

次に、上記の計算をすることにより、x\_trainとx\_testの各画素値を0.0から1.0の浮動小数点数(float64)に変換する。このように、ニューラルネットワークへの入力値は0から1に正規化することが多い。以下、x\_train, y\_trainの学習データを図示すると次のような構成になっている。なお、x\_train[i] (x\_trainのi番目のデータ)の画像に対する教師データ(正解データ)はy\_train[i]に格納されている。

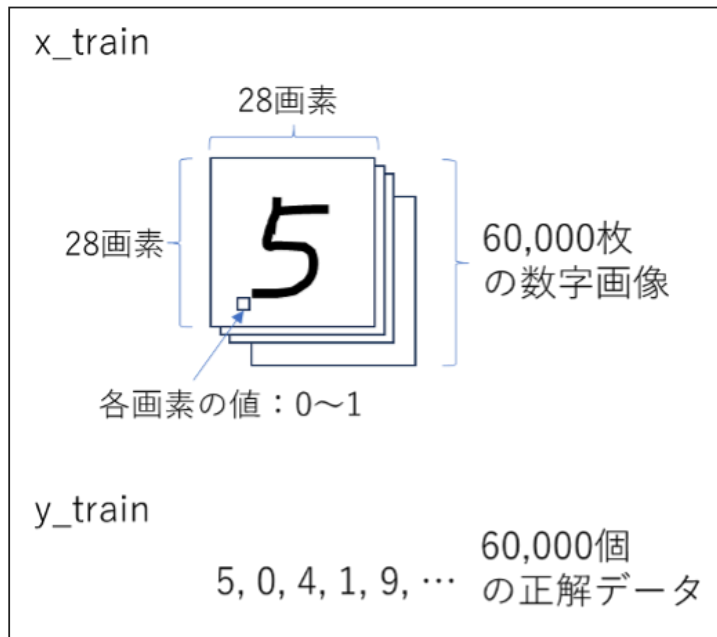


図33 学習データの構成

### 3. ニューラルネットワークの構成と学習法の設定

次に、ニューラルネットワークの構成と学習法を設定する。ここでは、次のように設定してみる。

- 入力層へは、28画素×28画素を1列に並べた784次元のベクトル（各画素は0～1の値）を入力する。よって、入力層のユニット数は784個となる。
- 中間層を64個とし、活性化関数はシグモイド関数を採用。
- 出力層は、数字のクラス数10個に合わせて、ユニット数は10個とし、ソフトマックス関数で各クラスの確率値を出力。
- 最適化アルゴリズムとしてSGD（確率的勾配降下法）を採用。
- 誤差関数としてクラス出力に関するクロスエントロピーを採用
- 学習途中の評価のための尺度として識別精度(accuracy)を指定

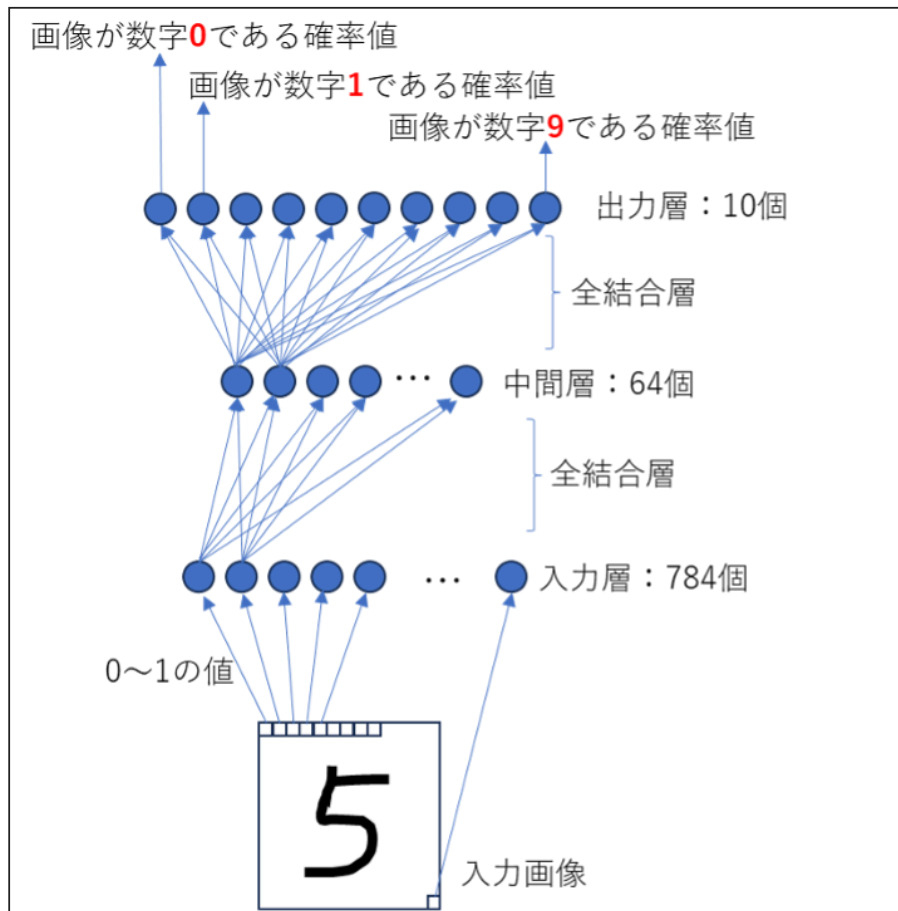
以上を実現するために次のように記述する。

```
In [28]: # モデル構築(A)
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(64, activation='sigmoid'), #活性化関数：シグモイド関数を指定
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='sgd',
```

```
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
metrics=['accuracy'])
```

上記の構造を図示すると次のとおりである。



## 4. ニューラルネットワークの学習と評価

次に、学習用データを用いて構成したニューラルネットワークを使って学習を行う。ここでは、学習用データとして、先ほど読み込んだ `x_train` と `y_train` のデータを用いる。`x_train` は60,000枚の数字画像（グレースケール画像）であり、`y_train` はそれらの各画像に対応する正解データ(0から9の数字)である。学習を実施するには、下記プログラムを実行する。

In [29]: `## モデルの学習`

```
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5  
1875/1875 [=====] - 6s 3ms/step - loss: 1.5011 - accuracy: 0.6872  
Epoch 2/5  
1875/1875 [=====] - 5s 3ms/step - loss: 0.7589 - accuracy: 0.8442  
Epoch 3/5  
1875/1875 [=====] - 6s 3ms/step - loss: 0.5564 - accuracy: 0.8707  
Epoch 4/5  
1875/1875 [=====] - 5s 3ms/step - loss: 0.4690 - accuracy: 0.8825  
Epoch 5/5  
1875/1875 [=====] - 5s 3ms/step - loss: 0.4200 - accuracy: 0.8908
```

Out[29]: `<keras.src.callbacks.History at 0x78b9c05d2ad0>`

ここで、エポック(epoch)とは、数字画像全体(60,000枚)を一通り学習すると1回とカウントする数値なので、この場合は5回繰り返し学習していることになる。学習結果は次のとおりである。なお、学習時のネットワークの重みの初期

値が乱数によって設定されるため、学習ごとに微妙に結果の値が異なる点を留意されたい。

これより、学習データについては、89.08%の認識率が得られていることがわかる。次に、テストデータについての認識率を評価してみよう。

In [30]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 3ms/step - loss: 0.3863 - accuracy: 0.8982
0.8981999754905701
```

この結果から、テストデータの認識率は 89.82% となっていることがわかる。

## 5. ニューラルネットワークの構成・学習法の変更による効果

次に、学習に用いるニューラルネットワークの構造や各種パラメータを変更して、さらにテストデータの認識率を向上させることができるかを検証してみよう。

### 5.1 中間層の活性化関数をシグモイド関数からReLUに変更

最初に中間層の活性化関数(\*1) をシグモイド関数からReLUに変更してみる。プログラムで、sigmoidの部分でreluに変更する（下記 activation='relu' の部分）。

(\*1) 利用可能な活性化関数は次のURLを参照：<https://keras.io/api/layers/activations/>

In [31]: # モデル構築(B)

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(64, activation='relu'), #活性化関数 : ReLUを指定
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='sgd',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [32]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.6652 - accuracy: 0.8297
Epoch 2/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.3397 - accuracy: 0.9063
Epoch 3/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.2906 - accuracy: 0.9179
Epoch 4/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.2613 - accuracy: 0.9264
Epoch 5/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.2399 - accuracy: 0.9325
```

Out[32]: <keras.src.callbacks.History at 0x78b9ea526230>

In [33]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.2221 - accuracy: 0.9355
0.9355000257492065
```

他のプログラムは前回と同様である。このモデルを用いた結果、学習データに対する認識率は93.25%、テストデータに対する認識率は93.55%に向上する。

## 5.2 中間層のユニット数を64個から128個に変更

次に、中間層のユニット数を64個から128個に増やしてみる。プログラムで、下記 `tf.keras.layers.Dense(128, activation='relu')` の部分を修正する。

```
In [34]: # モデル構築(C)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層：ユニット数128、活性化関数：ReLUを
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='sgd',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
In [35]: ## モデルの学習

model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.6627 - accuracy: 0.8339
Epoch 2/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.3369 - accuracy: 0.9057
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.2864 - accuracy: 0.9194
Epoch 4/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.2555 - accuracy: 0.9280
Epoch 5/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.2325 - accuracy: 0.9358
```

```
Out[35]: <keras.src.callbacks.History at 0x78b9c02e0070>
```

```
In [36]: # 認識率の評価

test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.2189 - accuracy: 0.9379
0.9379000067710876
```

結果、学習データとテストデータに対する認識率は、それぞれ93.58%、93.79%と若干ではあるが向上がみられた。その後、さらにユニット数を256個とする実験も行ったが、テストデータに対する認識率は、ほとんど変わらなかったため、ここでは128個に固定する。

## 5.3 最適化アルゴリズムをSGDからAdamに変更

次に、最適化アルゴリズム(\*)をSGDからAdamに変更してみる。プログラムでは、下記 `optimizer='adam'` の部分を修正する。

(\*) 利用可能な最適化アルゴリズムは次のURLを参照：<https://keras.io/api/optimizers/>

```
In [37]: # モデル構築(D)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層：ユニット数128、活性化関数：ReLUを
    tf.keras.layers.Dense(10)
])
```

```
model.compile(optimizer='adam', # 最適化アルゴリズム : Adam
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [52]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1039 - accuracy: 0.9678
Epoch 2/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.0992 - accuracy: 0.9683
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.0934 - accuracy: 0.9708
Epoch 4/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.0876 - accuracy: 0.9725
Epoch 5/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.0820 - accuracy: 0.9733
```

Out[52]: <keras.src.callbacks.History at 0x78b9c03ad4b0>

In [39]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0814 - accuracy: 0.9744
0.974399983882904
```

この結果、学習データとテストデータに対する認識率は、それぞれ 97.33%、97.44%と向上した（以後、Adamに固定）。

## 5.4 中間層を4層構造に変更

次に、中間層の数を1つ増やして、全体で4層構造のニューラルネットワークにしてみる。ここでは、中間層の第1層は128個のままとし、第2層として64個のものを追加する。使用する活性化関数はともにReLUである。

In [40]: # モデル構築(E)

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層 : ユニット数128、活性化関数 : ReLUを
    tf.keras.layers.Dense(64, activation='relu'), #中間層 : ユニット数64、活性化関数 : ReLUを指
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam', # 最適化アルゴリズム : Adam
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [41]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.2434 - accuracy: 0.9271
Epoch 2/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.0984 - accuracy: 0.9703
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.0694 - accuracy: 0.9777
Epoch 4/5
1875/1875 [=====] - 8s 4ms/step - loss: 0.0521 - accuracy: 0.9830
Epoch 5/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.0411 - accuracy: 0.9868
```

Out[41]: <keras.src.callbacks.History at 0x78b9ac796ef0>

In [42]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 4ms/step - loss: 0.0745 - accuracy: 0.9790
0.9789999723434448
```

この結果、学習データとテストデータに対する認識率は、それぞれ 98.68%、97.90%となった。これより、学習データについては向上し、テストデータについては若干の向上が見られる(層数を増やすと、計算コストがかかるため、以後は中間層は1層として実験をすることにする)。

## 5.5 学習回数（エポック）を増やす

次に、学習回数（エポック）を5から10に変更してみよう。

In [43]: # モデル構築(D)へ戻す

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層：ユニット数128、活性化関数：ReLUを
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam', # 最適化アルゴリズム：Adam
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [44]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=10) #学習回数(エポック数)を10に変更
```

```
Epoch 1/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.2613 - accuracy: 0.9260
Epoch 2/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.1144 - accuracy: 0.9664
Epoch 3/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0793 - accuracy: 0.9751
Epoch 4/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0591 - accuracy: 0.9817
Epoch 5/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0458 - accuracy: 0.9863
Epoch 6/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0360 - accuracy: 0.9888
Epoch 7/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0283 - accuracy: 0.9909
Epoch 8/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0234 - accuracy: 0.9926
Epoch 9/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0193 - accuracy: 0.9937
Epoch 10/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0157 - accuracy: 0.9951
```

Out[44]: <keras.src.callbacks.History at 0x78b9ac620b50>

In [45]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0751 - accuracy: 0.9797
0.9797000288963318
```

結果、学習データとテストデータに対する認識率は、それぞれ 99.51%、97.97%となった。



## 5.6 ドロップアウト機能の導入

モデルの設定によっては、学習データについては認識率が向上するが、テストデータについては認識率が下がってしまう場合がある。この傾向は、学習データに過剰に適応した過学習（overfitting）の状態を示している。ここでは、このような過学習の傾向を抑制するために、ドロップアウトの機能を導入してみよう。ここでは、出力層以外のユニットのうち、2割をランダムに無効化して学習することにする。

In [46]: # モデル構築(D) + Dropout 20%

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層：ユニット数128、活性化関数：ReLUを
    tf.keras.layers.Dropout(0.2), # 2割をドロップアウト
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam', # 最適化アルゴリズム：Adam
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [47]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=5) #学習回数(エポック数) は5を維持
```

```
Epoch 1/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.3005 - accuracy: 0.9134
Epoch 2/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1438 - accuracy: 0.9574
Epoch 3/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.1072 - accuracy: 0.9675
Epoch 4/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.0877 - accuracy: 0.9725
Epoch 5/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.0747 - accuracy: 0.9764
```

Out[47]: <keras.src.callbacks.History at 0x78b9ac409a80>

In [48]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0690 - accuracy: 0.9787
0.9786999821662903
```

これにより、学習データとテストデータに対する認識率は、それぞれ 97.64%、97.87%となり、ドロップアウトの適用前より認識率が若干向上しているのがわかる。

同じ条件で、ドロップアウトの確率を4割にした場合は、学習・テストデータ両方で、若干、認識率が下がってしまった。これより、あまり多くのユニットを削減してしまうと、認識率の低下につながってしまうため、適切な値を見つける必要がある。

In [49]: # モデル構築(D) + Dropout 40%

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'), #中間層：ユニット数128、活性化関数：ReLUを
    tf.keras.layers.Dropout(0.4), # 4割をドロップアウト
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam', # 最適化アルゴリズム：Adam
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```



In [50]: ## モデルの学習

```
model.fit(x_train, y_train, epochs=5) #学習回数(エポック数) は5を維持
```

```
Epoch 1/5
1875/1875 [=====] - 7s 3ms/step - loss: 0.3498 - accuracy: 0.8949
Epoch 2/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1848 - accuracy: 0.9452
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1470 - accuracy: 0.9563
Epoch 4/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1256 - accuracy: 0.9612
Epoch 5/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.1124 - accuracy: 0.9654
```

Out[50]: <keras.src.callbacks.History at 0x78b9ac345780>

In [51]: # 認識率の評価

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0841 - accuracy: 0.9747
0.9746999740600586
```

## 5.7 評価結果のまとめ

以上、今回の構造やパラメータ変更によって得られた結果を下表にまとめる。このように、ニューラルネットワークの学習を行う際には、様々なパラメータを試して、最適なモデルを探る必要がある。

表9 各種パラメータを変更した場合の認識率

| 中間層1:ユニット数 | 活性化     | 中間層2:ユニット数 | 活性化  | 最適化  | dropout | epoch | 認識率:学習 | 認識率:テスト |
|------------|---------|------------|------|------|---------|-------|--------|---------|
| 64         | sigmoid | ----       | ---- | SDG  | ----    | 5     | 89.08% | 89.82%  |
| 64         | ReLU    | ----       | ---- | SDG  | ----    | 5     | 93.25% | 93.55%  |
| 128        | ReLU    | ----       | ---- | SDG  | ----    | 5     | 93.58% | 93.79%  |
| 128        | ReLU    | ----       | ---- | Adam | ----    | 5     | 97.33% | 97.44%  |
| 128        | ReLU    | 64         | ReLU | Adam | ----    | 5     | 98.68% | 97.90%  |
| 128        | ReLU    | ----       | ---- | Adam | ----    | 10    | 99.51% | 97.97%  |
| 128        | ReLU    | ----       | ---- | Adam | 0.2     | 5     | 97.64% | 97.87%  |
| 128        | ReLU    | ----       | ---- | Adam | 0.4     | 5     | 96.54% | 97.47%  |

# memo