ニューラルネット(2)

前田利之（まえだ　としゆき）
aipr@e-chan.jp

阪南大学　経営情報学部

(2024.11.1)

出力層の設計

- ニューラルネットは、[　　]にも[　　]にも使える
  - [　　]とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - [　　]とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では[　　]関数を、分類では[　　]関数を用いる

出力層の設計

- ニューラルネットは、 分類 にも [　　] にも使える
  - [　　] とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - [　　] とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では [　　] 関数を、分類では [　　] 関数を用いる

出力層の設計

- ニューラルネットは、 **分類** にも **回帰** にも使える
  - [　　] とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - [　　] とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では [　　] 関数を、分類では [　　] 関数を用いる

出力層の設計

- ニューラルネットは、 **分類** にも **回帰** にも使える
  - **分類** とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - [　　] とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では [　　] 関数を、分類では [　　] 関数を用いる

出力層の設計

- ニューラルネットは、 **分類** にも **回帰** にも使える
  - **分類** とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - **回帰** とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では [　　] 関数を、分類では [　　] 関数を用いる

出力層の設計

- ニューラルネットは、**分類**にも**回帰**にも使える
  - **分類**とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - **回帰**とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では**恒等**関数を、分類では［　　］関数を用いる

出力層の設計

- ニューラルネットは、**分類**にも**回帰**にも使える
  - **分類**とは…データがどのクラスに属するか判断すること。例えば、動物の写真から、それが猫か犬かを分類する、等
  - **回帰**とは…ある入力データから、（連続的な）数値の予測をおこなうこと。例えば、人の写真から、その人の体重を予測する、等
- 一般に、活性化関数として、回帰では**恒等**関数を、分類では**ソフトマックス**関数を用いる

ソフトマックス関数

- $y_k = \dfrac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$
  - $exp(x)$ は指数関数 e^x
  - ☆ e とは何かを知りたい人は、 AI データサイエンス基礎／先端技術論の資料を参照のこと
- 式の分母より、全ての入力信号から影響を受けることになる

ソフトマックス関数

☆ 以下を Colab 上で実行する

```
import numpy as np
a = np.array([0.3, 2.9, 4.0])
exp_a = np.exp(a)
print(exp_a)
sum_exp_a = np.sum(exp_a)
print(sum_exp_a)
y = exp_a / sum_exp_a
print(y)
```

ソフトマックス関数

☆ 実行結果

```
[ 1.34985881 18.17414537 54.59815003]
74.1221542101633
[0.01821127 0.24519181 0.73659691]
```

ソフトマックス関数

☆ 今後のために関数化しておく。'Colab Notebooks'の中の mymodules フォルダの中に softmax0.py という名前で保存する。

```
import numpy as np
def softmax(a):
  exp_a = np.exp(a)
  sum_exp_a = np.sum(exp_a)
  y = exp_a / sum_exp_a
  return y
```

ソフトマックス関数

☆ 前頁がちゃんとできていることを確認する

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
from softmax0 import softmax
a = np.array([0.3, 2.9, 4.0])
print(softmax(a))
```

ソフトマックス関数実装の改良

- このままだと、指数関数の値が [　　] なりすぎ、計算できない（オーバーフロー）のおそれがある
- そこで、定数をかけてみる（詳細は次頁）
- つまり、指数関数の計算をするとき、定数を足しても（引いても）良いということ
- 入力値の [　　] を引けばオーバーフローは防げる！

ソフトマックス関数実装の改良

- このままだと、指数関数の値が 大きく なりすぎ、計算できない（オーバーフロー）のおそれがある
- そこで、定数をかけてみる（詳細は次頁）
- つまり、指数関数の計算をするとき、定数を足しても（引いても）良いということ
- 入力値の [　　] を引けばオーバーフローは防げる！

ソフトマックス関数実装の改良

- このままだと、指数関数の値が **大きく** なりすぎ、計算できない（オーバーフロー）のおそれがある
- そこで、定数をかけてみる（詳細は次頁）
- つまり、指数関数の計算をするとき、定数を足しても（引いても）良いということ
- 入力値の **最大値** を引けばオーバーフローは防げる！

ソフトマックス関数実装の改良

$$
\begin{aligned}
y_k &= \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)} \\
&= \frac{C \cdot exp(a_k)}{C \cdot \sum_{i=1}^{n} exp(a_i)} \\
&= \frac{exp(a_k + \log C)}{\sum_{i=1}^{n} exp(a_i + \log C)}
\end{aligned}
$$

ソフトマックス関数実装の改良

☆ mymodules フォルダの中の softmax0.py を書き換え softmax.py とするか、softmax.py をリンク先から Colab. の mymodules フォルダにコピーする

```
import numpy as np
def softmax(a):
  c = np.max(a)
  exp_a = np.exp(a-c)
  sum_exp_a = np.sum(exp_a)
  y = exp_a / sum_exp_a
  return y
```

ソフトマックス関数の特徴

☆ 前頁の softmax 関数を使うと、以下のようになる
(smax-ex.py をリンク先からコピーして実行する)

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
from softmax import softmax
a = np.array([0.3, 2.9, 4.0])
print(softmax(a))
print(np.sum(softmax(a))
```

ソフトマックス関数の特徴

- softmax() を使うと、前頁のようになる
- →合計は１
- →  として扱える（詳細は次週以降）
- 出力層のニューロン数が重要 (!)
  - 解く問題に応じて調整が必要
  - クラス分類の場合、分類したい数のニューロンを出力層に並べる必要がある

ソフトマックス関数の特徴

- softmax() を使うと、前頁のようになる
- →合計は１
- → **確率** として扱える（詳細は次週以降）
- 出力層のニューロン数が重要 (!)
  - 解く問題に応じて調整が必要
  - クラス分類の場合、分類したい数のニューロンを出力層に並べる必要がある

手書き文字認識

- ここでは学習済みのパラメータを使い、ニューラルネットの推論を試す
- （学習については次週以降で）
- この推論処理は、 [ ]

[ ] と呼ぶ

手書き文字認識

- ここでは学習済みのパラメータを使い、ニューラルネットの推論を試す
- （学習については次週以降で）
- この推論処理は、順方向伝播

[　　] と呼ぶ

手書き文字認識

- ここでは学習済みのパラメータを使い、ニューラルネットの推論を試す
- （学習については次週以降で）
- この推論処理は、順方向伝播 (forward propagation) と呼ぶ

手書き文字認識(MNIST データセット)

- 機械学習用に公開されているデータセットの１つ
- 簡単な実験から研究用途までいろんなところで使われている
  - ０～９の数字画像
  - 28x28 pixel, grayscale
  - 訓練用 60000, テスト用 10000 枚用意されている

手書き文字認識（準備）

- データセットのダウンロードから画像データの numpy 配列への変換までサポートするスクリプトが用意されている
- mnist.py をリンク先から Colab. の mymodules フォルダにコピーする

手書き文字認識

☆ 前頁がちゃんとできていることを確認する
(mn-ex.py をリンク先からコピーして実行する)

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
from mnist import load_mnist
```

(次頁へ続く)

手書き文字認識

☆ 前頁からの続きで、shape を確認する

```
(x_train, t_train), (x_test, t_test) = \
 load_mnist(flatten=True, normalize=False)
print(x_train.shape)
print(t_train.shape)
print(x_test.shape)
print(t_test.shape)
```

手書き文字認識

- 実行結果：

```
(60000, 784)
(60000,)
(10000, 784)
(10000,)
```

- x_train, t_train, x_test, t_test は、それぞれ「訓練画像」「訓練ラベル」「テスト画像」「テストラベル」

手書き文字認識（本番）

- 学習済みパラメータを利用して認識させる
- Colab. の mymodules フォルダに sample_weight.pkl をリンク先からコピーする

手書き文字認識

☆以下を Colab で実行する （プログラム全体は mnist_prac.py に置いてあります）

```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
from mnist import load_mnist
import pickle
from softmax import softmax
def sigmoid(x):
     return 1 / (1 + np.exp(-x))
def get_data():
    (x_train, t_train), (x_test, t_test) = \ # 本当は１行
    load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test
```

（続く）

手書き文字認識

（前頁からの続き）

```
def init_network():
  with open( \
  "/content/drive/My Drive/Colab Notebooks/mymodules/sample_weight.pkl",\
  'rb') as f: # ↑は長いが、他に手段が無い模様…
      network = pickle.load(f)
  return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)
    return y
```

（続く）

手書き文字認識

（前頁からの続き）

```
x, t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p= np.argmax(y)
    # 最も確率の高い要素のインデックスを取得
    if p == t[i]:
        accuracy_cnt += 1
print("Accuracy:"+str(float(accuracy_cnt)/len(x)))
```

手書き文字認識

- 出力が [　　] になるはず
  - np.argmax 関数は、引数に与えられた配列で最大の値を持つ要素のインデックスを取得する
  - load_mnist で normalize を True にすることで、データの値が 0.0〜1.0 の範囲に収まるように変換している [　　]
  - このような、入力データにたいする変換を [　　] と呼ぶ

手書き文字認識

- 出力が Accuracy:0.9352 になるはず
  - np.argmax 関数は、引数に与えられた配列で最大の値を持つ要素のインデックスを取得する
  - load_mnist で normalize を True にすることで、データの値が 0.0〜1.0 の範囲に収まるように変換している
  - このような、入力データにたいする変換を と呼ぶ

手書き文字認識

- 出力が Accuracy:0.9352 になるはず
  - np.argmax 関数は、引数に与えられた配列で最大の値を持つ要素のインデックスを取得する
  - load_mnist で normalize を True にすることで、データの値が 0.0〜1.0 の範囲に収まるように変換している （正規化）
  - このような、入力データにたいする変換を [　　] と呼ぶ

手書き文字認識

- 出力が Accuracy:0.9352 になるはず
  - np.argmax 関数は、引数に与えられた配列で最大の値を持つ要素のインデックスを取得する
  - load_mnist で normalize を True にすることで、データの値が 0.0〜1.0 の範囲に収まるように変換している（正規化）
  - このような、入力データにたいする変換を前処理と呼ぶ

手書き文字認識（おまけ）

- このニューラルネットワークは、
  「入力→重み１→重み２→重み３・出力」
  が
  「784 → 784x50 → 50x100 → 100x10 → 10」
  という次元でデータが流れている
- ここで、入力を「100x784」、出力を「100x10」という行列データで流すと、並列（高速）処理が可能→ 
- （…詳細は次週以降）

手書き文字認識（おまけ）

- このニューラルネットワークは、
  「入力→重み１→重み２→重み３・出力」
  が
  「784 → 784x50 → 50x100 → 100x10 → 10」
  という次元でデータが流れている
- ここで、入力を「100x784」、出力を「100x10」という行列データで流すと、並列（高速）処理が可能→ **「バッチ処理」**
- （…詳細は次週以降）

おしまい

質問・コメント等あれば、何でもお気軽に
aipr@e-chan.jp に
メールください！