第 5 章

方法

「OZ10 5 章。実際に君が何を行ったのか全然書いてない。まずそれを詳しく説明して、その後に実装方法が来るべき。」

5.1　MNIST データセット

MNIST データセット（Mixed National Institute of Standards and Technology database）[8] は 60000 個の訓練データと 10000 個のテストデータからなる、手書きの数字のデータセットである。それぞれのデータは手書き数字の画像と、表している数字の正解のラベルからなる。各画像は 28 × 28 ピクセルで、各ピクセルは 0（白）から 255（黒）までの値でその明度を表している。MNIST データセットは、最初の 60000 個が訓練セット、後ろの 10000 個がテストセットにはじめから分かれている。このデータセットは非常によく使われており、新しい分類アルゴリズムが登場するとまず MNIST データセットで性能を測るので、機械学習の初心者が必ず触れる題材であると言われている。図 5.1 は、MNIST データセットの訓練セットからランダムに取り出された 50 個のデータである。各画像の右下には正解のラベルの値も付してある。

図 5.1　MNIST データセットの 50 個のデータ

5.2　動作環境

実装は以下の環境のもとで行った。なお Keras は TensorFlow に付属のものを用いた。

- OS：macOS Catalina 10.15.2
- Python：3.6.6
- TensorFlow：1.11.0
- Keras on TensorFlow：2.1.6

5.3　フィードフォーワード・ニューラルネットワークによる学習

5.3.1　ベースとなるフィードフォーワード・ニューラルネットワークの構築

活性化関数や隠れ層の数などを変化させたときにニューラルネットワークの性能にどのような違いが生じるかを試したかったので、できるだけシンプルな、隠れ層が 1 層のフィードフォーワード・ニューラルネットワークのプログラムを作成した。機械学習のためのオープンソースライブラリである Keras を用いてソースコードを書いた。

まず MNIST データセットの各画像データはサイズが 28×28 の 2 次元の配列であるが、これを 784 次元のベクトルに変換した。各ピクセル値は 0 から 255 までの整数であるが、これを 0 から 1 までの値に変換した。またデータのラベルに対応したワンホットベクトルを作成しておいた。次に学習をさせるニューラルネットワークを構築した。各ユニットが次の層のすべてのユニットに影響する全結合型ニューラルネットワークとし、隠れ層は 1 層でそのユニット数は 100 としておいた。入力層は 784 ユニット、出力層は 10 個の分類をするので 10 ユニットである。入力層から隠れ層への活性化関数は ReLU を、隠れ層から出力層への活性化関数はソフトマックス関数とした。コスト関数は交差エントロピーを採用し、出力とラベルのワンホットベクトルからこれを計算した。誤差逆伝搬によりコスト関数の勾配から、入力層と隠れ層、隠れ層と出力層の間の重みパラメータおよび隠れ層、出力層の各ユニットについてのバイアスのパラメータの更新を行なった。ただし最適化アルゴリズムは確率的勾配降下法とし、ミニバッチのサイズは 1000 としておいた。この処理を $60000/1000 = 60$ 回繰り返した。ただしミニバッチは確率的に選ばれるため、この反復によって 60000 個の訓練データすべてが用いられるわけではない。次に 10000 個のテストデータを用いて予測を行い、正解率を算出した。ここまでの反復を 1 エポックという。エポック数は 100 として、合計で $100 \times 60 = 6000$ 回のパラメータ更新が行われるようにした。初期設定を箇条書きにまとめると次のようになる。これ以降ではこれらを少しずつ変化させて性能を比較していった。

- 隠れ層：層の数は 1 でユニット数は 100
- 入力層から隠れ層への活性化関数：ReLU
- 隠れ層から出力層への活性化関数：ソフトマックス関数
- コスト関数：交差エントロピー
- 最適化アルゴリズム：確率的勾配降下法（SGD）
- ミニバッチのサイズ：1000
- エポック数：100

プログラムのソースコードとそのより詳しい説明は付録 A の A.1 に示した。

5.3.2　活性化関数の比較

活性化関数を ReLU からシグモイド関数に変えてプログラムを実行し性能を比較した。

5.3.3　コスト関数の比較

コスト関数を交差エントロピーから平均二乗誤差に変えてプログラムを実行し性能を比較した。

5.3.4　最適化アルゴリズムの比較

最適化アルゴリズムを確率的勾配降下法から Adam に変えて性能を比較した。

5.3.5　エポック数、バッチサイズの変更

エポック数を 10, 1000 に、バッチサイズを 100, 10000, 100000 にそれぞれ変更したときの結果を表示させ、それらを比較した。

5.3.6　隠れ層を 2 層に増やした場合の性能評価

隠れ層を 1 層追加して、 2 層にして性能を比較した。ただしユニット数をいくつにすれば良いのかわからなかったため、第 1 層目の隠れ層のユニット数は 100 のまま、追加した第 2 層目の隠れ層のユニット数を 10, 25, 50, 100, 200 のように変化させたときのものを出力した。逆に第 2 の隠れ層のユニット数を 100 にしたまま、第 1 の隠れ層のユニット数を同じように変化させて比較した。なお追加した第 2 層目の隠れ層の活性化関数は第 1 層目と同じく ReLU とした。

5.3.7　隠れ層を 3 層以上に増やした場合の性能評価

節 5.3.6 を実行することによって、出力層に向かうにつれてユニット数を少なくしてゆく「漏斗型」のフィードフォーワード・ニューラルネットワークが、精度良く数字を予測することを経験的に発見した。そこでネットワークをより深くするとき、どのように細くしてゆくのがいいかを検証した。具体的には a をある自然数、$n_{\text{classes}} = 10$ をクラスの数、隠れ層の数を N としたとき（入力層と出力層も合わせれば全体で $(N+2)$ 層のネットワークになる）、隠れ層のユニット数を順方向に $10 \times a^N,\ 10 \times a^{N-1},\ \ldots,\ 10 \times a^2,\ 10 \times a$ のように、指数関数的に減らしていくことを考えた（図 5.2）。ただし入力の次元が 784 であるから第 1 隠れ層のユニット数がそれよりも小さくなるように、$n_{\text{classes}} = 10$ と a の値を決めたのち $10 \times a^N < 784$ を満たす最大の隠れ層数 N について検証した。そのような数の組み合わせは表 5.1 のように 7 通りあるので、それらを全て試した。ただし隠れ層の活性化関数はすべて ReLU とした。またソースコードの上で隠れ層を増やすのには反復文を用いて手間を省く工夫をした。具体的には付録 A のソースコード A.2 のように変更した。

比較のため、隠れユニット数を順方向に $10 \times a^1,\ 10 \times a^2,\ \ldots,\ 10 \times a^N$ のように増加させていった「逆漏斗型」のニューラルネットワークの性能を $a = 2,\ N = 6$ の場合に測定した。それとは別に、ユニット数が 210 で一定で層数 6 の「寸胴型」ニューラルネットワーク（総隠れユニット数は

$210 \times 6 = \sum_{k=1}^{6} 2^k = 1260$ で同じになる）についても性能を測定した。

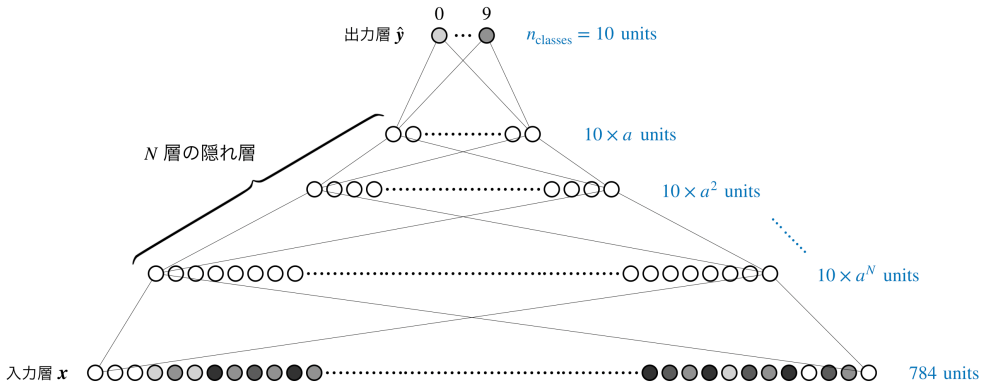

図 5.2　隠れユニット数が指数的に減少するニューラルネットワーク

表 5.1　a と N の組み合わせ

a	2	3	4	5	6	7	8
N	6	3	3	2	2	2	2

5.3.8　ドロップアウトの効果

ドロップアウトは畳み込みニューラルネットワークで用いられる手法であるが、その中でも出力層に近くにある、部分的にフィードフォーワード・ニューラルネットワークとみなせる部分で用いられるため、節 5.3.1 のニューラルネットワークを用いて検証を行った。

節 5.3.1 のベースとなるニューラルネットワークの隠れ層と出力層の間にドロップアウト層を挿入し、ドロップアウトの割合を $p = 0$（ドロップアウトをしない）, 0.25, 0.5, 0.75 のように変化させたときの出力をみた。ただし収束を早めるため最適化アルゴリズムは Adam を採用し、エポック数、ミニバッチのサイズはドロップアウト率に合わせて変化させた。

5.4　畳み込みニューラルネットワークによる学習

5.4.1　ベースとなる畳み込みニューラルネットワークの構築

隠れ層を畳み込み層、プーリング層などに変化させたときに畳み込みニューラルネットワークの性能にどのような違いが生じるかを試したかったので、できるだけシンプルな、隠れ層が 1 層の畳み込みニューラルネットワークのプログラムを作成した。エポック数は 20, ミニバッチのサイズは 1000 とした。

まず MNIST データセットの各画像データをそのまま扱うため、これを 60000 × 28× の配列に変換した。各ピクセル値は 0 から 255 までの整数であるが、これを 0 から 1 までの値に変換した。またデータのラベルに対応したワンホットベクトルを作成しておいた。次に学習をさせるニューラルネットワークを構築した。各ユニットが次の層のすべてのユニットに影響する全結合型ニューラルネットワークとし、隠れ層は 1 枚のフィルターを持つ 1 層の畳み込み層とし、そのフィルターのサイズは 3 × 3 としておいた。畳み込み層の出力サイズが入力と変わらないように、適宜ゼロパディングを行っておいた。入力層から隠れ層への活性化関数は ReLU を用いた。この隠れ層の出力のサイズは 1000（ミニバッチのサイ

ズ）×1（フィルターの数）×28 × 28（画像のサイズ）の 4 次元であるが、次の出力層に入れるため、これを 1000 × (1 · 28 · 28) の 2 次元に変換する必要があった。隠れ層から出力層への活性化関数はソフトマックス関数とした。最適化アルゴリズムは確率的勾配降下法とし、エポック数は 20 として、合計で 20 × 60 = 1200 回のパラメータ更新が行われるようにした。ここでパラメータはフィルターの 3 × 3 = 9 個の値と、全体に加える 1 つのバイアスである。初期設定を箇条書きにまとめると次のようになる。これ以降ではこれらを少しずつ変化させて性能を比較していった。

- 隠れ層：畳み込み層
  - フィルターのサイズ：3 × 3
  - フィルターの数：1
- 入力層から隠れ層への活性化関数：ReLU
- 隠れ層から出力層への活性化関数：ソフトマックス関数
- コスト関数：交差エントロピー
- 最適化アルゴリズム：確率的勾配降下法（SGD）
- ミニバッチのサイズ：1000
- エポック数：20

プログラムのソースコードとそのより詳しい説明は付録 A の A.1 に示した。

5.4.2　フィルターのサイズと枚数の比較

3 × 3 のフィルター数を 1 枚ずつ 6 枚まで増やし、学習の結果を出力させた。またフィルター数は 1 枚に固定して、そのサイズを 4 × 4, 5 × 5, 6 × 6 と変化させてその結果を表示させた。

5.4.3　プーリング層の追加

次に畳み込み層の代わりに最大プーリング層を通して実行結果をみた。プーリングカーネルのサイズは 2 × 2 とした。また平均プーリングを行ってどのような差異が現れるか確認した。最大プーリングカーネルのサイズを 3 × 3, 4 × 4 のように変化させたとき実行結果がどのように変化するかを試した。

付録 A

実装に用いたソースコード

A.1　1 隠れ層のフィードフォーワード・ニューラルネットワークのソースコード

ソースコード A.1　1 隠れ層のニューラルネットワーク

```
import tensorflow as tf

# (1) データのインポート
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# (2)データの変形
import numpy as np
from tensorflow.python.keras.utils import np_utils
# (2-1)
x_train = x_train.reshape(60000, 784) # (2-1-1)
x_train = x_train.astype('float32') # (2-1-2)
x_train = x_train / 255 # (2-1-3)
num_classes = 10 # (2-1-4)
y_train = np_utils.to_categorical(y_train, num_classes) # (2-1-5)
# (2-2)
x_test = x_test.reshape(10000, 784)
x_test = x_test.astype('float32')
x_test = x_test / 255
y_test = np_utils.to_categorical(y_test, num_classes)

# (3) ネットワークの定義
np.random.seed(1) # (3-1)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
model = Sequential() # (3-2)
model.add(Dense(100, input_dim=784, activation='relu')) # (3-3)
model.add(Dense(10, activation='softmax')) # (3-4)
model.compile(loss='categorical_crossentropy',
```

```
30              optimizer='sgd', metrics=['accuracy']) # (3-5)
31
32 # (4) 学習
33 num_epochs = 100
34 batchsize = 1000 # (4-1)
35
36 import time # (4-2)
37 startTime = time.time() # (4-3)
38 history = model.fit(x_train, y_train, epochs=num_epochs, batch_size=batchsize,
39                     verbose=1, validation_data=(x_test, y_test)) # (4-4)
40 score = model.evaluate(x_test, y_test, verbose=0, batch_size=batchsize) # (4-5)
41
42 # (5) 計算結果の表示
43 print('Test␣loss:', score[0])
44 print('Test␣accuracy:', score[1])
45 print("Computation␣time:{0:.3f}␣sec".format(time.time() - startTime))
46 print(model.summary())
```

まず (1) で MNIST データセットのデータをインポートしている。x_train は 60000 × 28 × 28 の配列変数で、それぞれの要素には画像のグレースケールを表す 0 から 255 までの整数値で格納されている。y_train はサイズが 60000 の 1 次元配列で、それぞれの要素には 0 から 9 まで正解のラベルが格納される。同様に 10000 個のテストデータが (x_test,y_test) に格納される。

(2) でデータの配列を変形した。まず Python の拡張モジュール NumPy と np_utils をインポートしておいた。(2-1-1) で 60000 × 28 × 28 の配列を 60000 × 784 の配列に変換した。x_train を int 型（整数）から float 型（32 ビットの不動少数点数）に変換し (2-1-2)、0 から 1 までの実数に変換した (2-1-3)。分類するクラスの数を num_classes=10 として (2-1-4)、y_train をワンホットベクトルに変換した (2-1-5)。同じ処理をテストデータについても行った (2-2)。

(3) でニューラルネットワークの定義をした。まず NumPy による乱数を固定するために、seed 値を固定した (3-1)。これをすることで再現性のある分析や処理を行うことができる。model を Sequential() で定義した (3-2)。パーセプトロンが信号を次のパーセプトロンにつないでゆき、これをつなげていくというモデルが Sequential である。隠れ層のユニット数を 100, 活性化関数には ReLU を使うことをで指定した (3-3)。出力層のユニット数が 10 で、活性化関数にソフトマックス関数を使うことを (3-4) で指定した。最後にコスト関数には交差エントロピーを、最適化アルゴリズムには確率的勾配法（SGD）を採用した (3-5)。また metrics=['accuracy'] とすることで、テストデータでネットワークの性能を測る際の汎化性能を、テストデータにおける正解率（accuracy）でみた。

(4) でニューラルネットワークに学習をさせ、その結果を計算させた。エポック数を 100, ミニバッチに用いる訓練データの数を 1000 にし、あとで調節できるようにそれぞれ num_epochs と batchsize という変数に格納しておいた (4-1)。プログラムの実行時間を計測するために、time モジュールをインポートしておき (4-2)、実行開始時刻を startTime とした (4-3)。エポックごとの学習の評価値を表示させるために verbose=1 とし、学習を行った (4-4)。最後にテストデータでのコスト関数と汎化誤差（ここでは正解率）を計算させた (4-5)。score は 2 成分からなる配列で、第 0 成分にテストデータでのコスト関数、第 1 成分に正解率が格納される。

(5) で学習結果を表示させた。テストデータでのコスト関数を Testloss、正解率を Testaccuracy、

実行時間を Computationtime で表示させた。print(mode.summary()) でモデルの層に対するパラメータの数を表示させた。

また節 5.3.3 では、Python の for 文を用いソースコード A.1 の (3-3) から (3-5) の部分をソースコード A.2 のように変更した。

ソースコード A.2　隠れ層を増やすために加えた変更

```
num_hidden_layers = 2
a = 8
num_first_hidden_units = num_classes * a ** num_hidden_layers
model.add(Dense(num_first_hidden_units, input_dim=784, activation='relu'))
for k in range(2, num_hidden_layers + 1):
    num_hidden_units = num_classes * a ** (num_hidden_layers + 1 - k)
    model.add(Dense(num_hidden_units, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd', metrics=['accuracy'])
```

A.2　1 隠れ層の畳み込みニューラルネットワークのソースコード

ソースコード A.3　1 隠れ層の畳み込みニューラルネットワーク

```
import tensorflow as tf

# (1) データのインポート
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# (2) データの変形
import numpy as np
from tensorflow.python.keras.utils import np_utils
# (2-1)
x_train = x_train.reshape(60000, 28, 28, 1) # (2-1-1)
x_train = x_train.astype('float32')
x_train /= 255
num_classes = 10
y_train = np_utils.to_categorical(y_train, num_classes)
# (2-2)
x_test = x_test.reshape(10000, 28, 28, 1)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test, num_classes)

# (3) ネットワークの定義
np.random.seed(1)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

```
26 from tensorflow.keras.layers import Activation, Flatten, Dense, Dropout
27 import time
28
29 model = Sequential()
30 model.add(Conv2D(1, (3, 3), # (3-1)
31                  padding='same', # (3-2)
32                  input_shape=(28, 28, 1), activation='relu'))
33 model.add(Flatten()) # (3-3)
34 model.add(Dense(10, activation='softmax'))
35 model.compile(loss='categorical_crossentropy',
36               optimizer='sgd', metrics=['accuracy'])
37
38 # (4) 学習
39 startTime = time.time()
40
41 num_epochs = 20
42 batchsize = 1000
43 history = model.fit(x_train, y_train, batch_size=batchsize, epochs=num_epochs,
44                     verbose=1, validation_data=(x_test, y_test))
45 score = model.evaluate(x_test, y_test, verbose=0)
46
47 # (5) 計算結果の表示
48 print('Test␣loss:', score[0])
49 print('Test␣accuracy:', score[1])
50 print("Computation␣time:{0:.3f}␣sec".format(time.time() - startTime))
51 print(model.summary())
```

フィードフォーワード・ニューラルネットワークの学習のときと同様、(1) で MNIST データセットのデータをインポートし、(2) でデータの配列を変形した。ただし画像のデータをベクトルに展開せず、(2-1-1) で $60000 \times 28 \times 28 \times 1$ のまま使用した点が異なる。テストデータも同様に処理した (2-2)。

(3) でニューラルネットワークの定義をした。model は Sequential() とした。1 枚の 3×3 のフィルターを学習するパラメータとし (3-1)、出力のサイズが変わらないようにパディングを追加した (3-2)。活性化関数は ReLU とした。この層の出力サイズは $28 \times 28 \times 1$ だが、出力層に入力するのにこれを 784 のベクトルに変形した (3-3)。出力層の活性化関数はソフトマックス関数、コスト関数は交差エントロピー、最適化アルゴリズムは確率的勾配降下法（SGD)、汎化性能を正解率とした。

あとはフィードフォーワード・ネットワークの実装と同様、(4) で学習、(5) で学習の結果を表示させた。