『TensorFlow で学ぶディープラーニング入門 畳み込みニューラルネットワーク徹底解説』 読書ノート

中井悦司（なかい えつじ）による 2016 年 9 月 27 日の本。

`https://github.com/enakai00/jupyter_tfbook`

p. 19

4 次関数によるモデル:

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$$

ここで x は月であり $x \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$。$y$ は x 月について予測される月中平均気温（℃）。w_0, w_1, w_2, w_3, w_4 はパラメータ。

p. 20

誤差関数 (error function) として二乗誤差 (squared error) を考える:

$$E = \frac{1}{2} \sum_{n=1}^{12} (y_n - t_n)^2$$

ここで y_n は n 月の予測気温。t_n は n 月の実際の気温である。

p. 22

直線を表す数式:

$$f(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

この形だと有名な $y = ax + b$ よりも x_1 と x_2 を対称に扱える。
$f(x_1, x_2) = 0$ が境界となるほか、境界から離れるほど $f(x_1, x_2)$ の値が $\pm\infty$ に近づく。

p. 23

検査結果 (x_1, x_2) のときの感染確率 $P(x_1, x_2)$ を sigmoid 関数 σ を使って表す:

$$P(x_1, x_2) = \sigma(f(x_1, x_2))$$

p. 24

1 層のノードからなるニューラルネットワークの例。

$$z = \sigma(f(x_1, x_2))$$

where

$$f(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2$$

p. 25

2 層のノードからなるニューラルネットワークの例。

$$z = \sigma(f(z_1, z_2))$$

where

$$z_1 = \sigma(f_1(x_1, x_2))$$
$$z_2 = \sigma(f_2(x_1, x_2))$$

where

$$f_1(x_1, x_2) = w_{10} + w_{11} x_1 + w_{12} x_2$$
$$f_2(x_1, x_2) = w_{20} + w_{21} x_1 + w_{22} x_2$$
$$f(z_1, z_2) = w_0 + w_1 z_1 + w_2 z_2$$

このニューラルネットワークには w_n という形の合計 9 個のパラメータが含まれている。
ニューラルネットワークをより複雑にするためにノードを増やす方法には次の 2 つがある。

- 層の個数を増加する
- 層にあるノードの個数を増加する

p. 30

x 月と予測気温 y との関係:

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$$

2 乗誤差による誤差関数 E。ただし y_n は n 月の予測気温。t_n は n 月の実際の気温。

$$E = \frac{1}{2} \sum_{n=1}^{12} (y_n - t_n)^2$$

これらを合わせると n 月の予測気温 y_n を次のように書ける。

$$\begin{aligned}
\boldsymbol{y} &= (y_1, y_2, \cdots, y_{11}, y_{12}) \\
y_n &= w_0 + w_1 n + w_2 n^2 + w_3 n^3 + w_4 n^4 \\
&= w_0 n^0 + w_1 n^1 + w_2 n^2 + w_3 n^3 + w_4 n^4 \\
&= \sum_{m=0}^{4} w_m n^m \\
y_n &= y_n(w_0, w_1, w_2, w_3, w_4) \\
\boldsymbol{y} &= \boldsymbol{y}(w_0, w_1, w_2, w_3, w_4)
\end{aligned}$$

この y_n を上記の誤差関数 E に代入して次の式を得る。

$$E(\boldsymbol{y}) = \frac{1}{2}\sum_{n=1}^{12}(y_n - t_n)^2$$

$$E(w_0, w_1, w_2, w_3, w_4) = \frac{1}{2}\sum_{n=1}^{12}\left(\sum_{m=0}^{4} w_m n^m - t_n\right)^2$$

これを最小にする $w_0, \cdots, w_4$ を求めたい。$w_0, \cdots, w_4$ のそれぞれによる偏微分の値を 0 とした次の連立方程式を解くことになる。

$$\frac{\partial E}{\partial w_m}(w_0, w_1, w_2, w_3, w_4) = 0 \quad \text{for } m = 0, \cdots, 4$$

p. 31

偏微分とは、複数の変数を持つ関数をある 1 つの変数で微分することである。1 変数の関数 $y = f(x)$ について最小値や最大値を求めるときに、微分係数を 0 とした次の方程式を解くのと本質的には同じだ。

$$\frac{df}{dx}(x) = 0$$

この式は、入力 x において関数 f がどう動くかを問題にしている。入力 x の付近でわずかに x が動くときに関数 f の微小な動きとの比率が 0 であるとは、x の変化が f に影響していないということだ。つまりグラフはそこにおいて極小値や極大値や停留値を取る。x をそのように制約して求めるものが与式である。

p. 32

次の 2 次関数を考える。

$$h(x_1, x_2) = \frac{1}{4}(x_1^2 + x_2^2)$$

その偏微分は次のようになる。

$$\frac{\partial h}{\partial x_1}(x_1, x_2) = \frac{1}{2}x_1$$
$$\frac{\partial h}{\partial x_2}(x_1, x_2) = \frac{1}{2}x_2$$

このとき次のように書いて勾配ベクトルと呼ぶ。

$$\nabla h(x_1, x_2) = \begin{pmatrix} \frac{\partial h}{\partial x_1}(x_1, x_2) \\ \frac{\partial h}{\partial x_2}(x_1, x_2) \end{pmatrix} = \begin{pmatrix} \frac{1}{2}x_1 \\ \frac{1}{2}x_2 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

勾配ベクトル $\nabla h(x_1, x_2)$ の向きは、壁を登る方向に一致する。勾配ベクトルの大きさ $||\nabla h(x_1, x_2)||$ は、壁を登る傾きに一致する。

p. 33

勾配降下法:

$$\boldsymbol{x}_{\text{new}} = \boldsymbol{x} - \nabla h$$

p. 34

次の 2 つの 2 次関数、h_1 と h_2 を考える。

$$h_1(x_1, x_2) = \frac{3}{4}(x_1^2 + x_2^2)$$
$$h_2(x_1, x_2) = \frac{5}{4}(x_1^2 + x_2^2)$$

それぞれの勾配ベクトルとして次のものが導ける。

$$\nabla h_1 = \frac{3}{2}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$\nabla h_2 = \frac{5}{2}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

この後者について $\boldsymbol{x}_{\text{new}} = \boldsymbol{x} - \nabla h$ という素朴な勾配降下法で $\boldsymbol{x}$ を更新していくと、パラメータ $\boldsymbol{x} = (x_1, x_2)$ は発散する。

よって学習率 c を用いて次のようにする。

$$\boldsymbol{x}_{\text{new}} = \boldsymbol{x} - c\nabla h$$

二乗誤差 $E(w_0, w_1, w_2, w_3, w_4)$ を最小化するのにも同じことをする。

$$\boldsymbol{w}_{\text{new}} = \boldsymbol{w} - \epsilon\nabla E(\boldsymbol{w})$$

ここで勾配ベクトル $\nabla E(\boldsymbol{w})$ は次のものである。

$$\nabla E(\boldsymbol{w}) = \begin{pmatrix} \frac{\partial E}{\partial w_0}(\boldsymbol{w}) \\ \vdots \\ \frac{\partial E}{\partial w_4}(\boldsymbol{w}) \end{pmatrix}$$

p. 38 環境準備

TensorFlow をインストール済みの Docker イメージが本書から提供されているので、それを使えばたいていの環境で同じコードを簡単に動かせるとのこと。

Ubuntu OS について考える。

```
$ sudo apt install docker.io
$ docker --version
Docker version 18.09.2, build 6247962
```

In the past this way was discouraged as the docker package was super outdated. The universe sources are fairly recent now.

This is now the best way nowadays, since the Ubuntu repos are keeping up with the docker releases.

```
# Docker Hub で Docker イメージらを検索する。
sudo docker search enakai00
# Docker イメージを Docker Hub から取得する。
sudo docker pull enakai00/jupyter_tensorflow
# 取得済みのイメージらを一覧する。
sudo docker images
# Docker コンテナを一覧する。
sudo docker ps     # 実行中のもののみ表示する
sudo docker ps -a  # 停止中のものも含めて表示する
# Docker コンテナをシャットダウンする。
sudo docker stop example
# Docker コンテナを削除する。
sudo docker rm example
```

```
# コンテナを生成して起動する。
sudo docker run enakai00/jupyter_tensorflow
# 名前つきでコンテナを生成する。
sudo docker run --name nakai enakai00/jupyter_tensorflow
# ホスト側のポートをコンテナ側のポートに転送する。http://localhost:8888/ で繋がるようになる。
sudo docker run --name nakai -p 8888:8888 enakai00/jupyter_tensorflow
# ログイン用のパスワードを書籍の通りに指定する。
sudo docker run --name nakai -p 8888:8888 -e PASSWORD=passw0rd enakai00/jupyter_tensorflow
```

p. 48 TensorFlow クイックツアー

```
pip3 install tensorflow
```

p. 59

予測気温の式。

$$y(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 = \sum_{m=0}^{4} w_m x^m$$

p. 67

(x_1, x_2) 平面を分割する直線の式:

$$f(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

この式は境界線から離れるにしたがって $\pm\infty$ に向かって変化する。また、未知のパラメータ w_0, w_1, w_2 を含んでいる。

sigmoid 関数:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

検査結果 (x_1, x_2) に対して、ウィルスに感染している確率:

$$P(x_1, x_2) = \sigma(f(x_1, x_2))$$

教師データを考える。n 番目の入力を (x_{1n}, x_{2n}) とする。n 番目の正解を $t_n = 0, 1$ とする。感染している場合が $t_n = 1$ である。現在のモデルによると、感染している確率は $P(x_{1n}, x_{2n})$ となる。この値に応じて確率的に予測してみる。

n 番目のデータを正しく予測する確率を P_n として、次が成り立つ。

- $t_n = 1$ の場合：$P_n = P(x_{1n}, x_{2n})$
- $t_n = 0$ の場合：$P_n = 1 - P(x_{1n}, x_{2n})$

1 つの数式にまとめる。

$$P_n = \{P(x_{1n}, x_{2n})\}^{t_n} \{1 - P(x_{1n}, x_{2n})\}^{1-t_n}$$

N 個のデータすべてに正解する確率 P を考えたい。

$$P = P_1 \times P_2 \times \cdots \times P_N = \prod_{n=1}^{N} P_n$$

$$P = \prod_{n=1}^{N} \{P(x_{1n}, x_{2n})\}^{t_n} \{1 - P(x_{1n}, x_{2n})\}^{1-t_n}$$

「与えられたデータを正しく予測する確率を最大化する」という方針でパラメータを調整する方法を「最尤推定法 (maximum likelihood estimation)」という。

掛け算を大量に含む数式は TensorFlow では効率がよくない。よって次の誤差関数を使う:

$$E = -\log P$$

なお対数については次の操作ができる:

$$\log ab = \log a + \log b$$
$$\log a^n = n \log a$$

これにより次の変形ができる。

$$\begin{aligned}
E &= -\log P \\
&= -\log \prod_{n=1}^{N} \{P(x_{1n}, x_{2n})\}^{t_n} \{1 - P(x_{1n}, x_{2n})\}^{1-t_n} \\
&= -\sum_{n=1}^{N} \log \left[\{P(x_{1n}, x_{2n})\}^{t_n} \{1 - P(x_{1n}, x_{2n})\}^{1-t_n} \right] \\
&= -\sum_{n=1}^{N} \left[\log \{P(x_{1n}, x_{2n})\}^{t_n} + \log \{1 - P(x_{1n}, x_{2n})\}^{1-t_n} \right] \\
&= -\sum_{n=1}^{N} [t_n \log P(x_{1n}, x_{2n}) + (1 - t_n) \log \{1 - P(x_{1n}, x_{2n})\}]
\end{aligned}$$

implementation

```
# p. 51

import matplotlib.pyplot as plt  # グラフ描画
import numpy as np               # 数値計算
import tensorflow as tf          # 機械学習

print("tf.__version ==", tf.__version__)  # 1.13.1
# tf.logging.set_verbosity(tf.logging.ERROR)  # warning を非表示にする。

# Tensor 型。
x = tf.placeholder(tf.float32,  # float32 型を行列要素の型とする
                   [None, 5])   # 行列のサイズを?x5 とする

# パラメータ。RefVariable 型。
w = tf.Variable(tf.zeros([5, 1]))  # -> warning

# 予測値のモデル。Tensor 型。?x1 行列。
y = tf.matmul(x, w)

# 観測値。Tensor 型。
t = tf.placeholder(tf.float32,
```

```
                    [None, 1])

# 損失関数。Tensor 型。
loss = tf.reduce_sum(tf.square(y-t))  # 二乗誤差

# Operation 型。
# パラメータ更新アルゴリズムに Adam を選ぶ。損失関数 loss の出力を最小化するように指定する。
train_step = tf.train.AdamOptimizer().minimize(loss)

# セッションを用意する。
sess = tf.Session()
# 変数を初期化する。
# sess.run(tf.initialize_all_variables())  # -> warning
sess.run(tf.global_variables_initializer())

# 観測値。教師データ。
train_t = np.array([5.2, 5.7, 8.6, 14.9, 18.2, 20.4,
                    25.5, 26.4, 22.8, 17.5, 11.1, 6.6])
# -> [ 5.2  5.7  8.6 14.9 18.2 20.4 25.5 26.4 22.8 17.5 11.1  6.6]
train_t = train_t.reshape([12, 1])
# -> [[ 5.2][ 5.7][ 8.6][14.9][18.2][20.4][25.5][26.4][22.8][17.5][11.1][ 6.6]]

train_x = np.zeros([12, 5])
for row, month in enumerate(range(1, 13)):  # row == month - 1
    for col, n in enumerate(range(0, 5)):   # col == n == 0, 1, 2, 3, 4
        train_x[row][col] = month**n

i = 0
for _ in range(100_000):
    i += 1
    sess.run(train_step, feed_dict={x:train_x, t:train_t})
    if i % 10_000 == 0:
        # 現在のパラメータでの損失を得る。
        loss_val = sess.run(loss, feed_dict={x:train_x, t:train_t})
        print("Step: {}, Loss: {}".format(i, loss_val))

w_val = sess.run(w)  # np.ndarray

# 予測気温 y(x) = w_0 + w_1*x + w_2*x^2 + w_3*x^3 + w_4*x^4
#              = Sigma(from m=0 to 4) w_m*x^m
```

```
def predict(x):
    result = 0.0
    for n in range(0, 5):
        result += w_val[n][0] * x**n
    return result
def plot():
    fig = plt.figure()
    subplot = fig.add_subplot(1, 1, 1)
    subplot.set_xlim(1, 12)
    subplot.scatter(range(1, 13), train_t)
    linex = np.linspace(1, 12, 100)  # 区間 [1, 12] で 100 要素。
    liney = predict(linex)
    subplot.plot(linex, liney)
    plt.show()
plot()
```