Python と深層学習

１．はじめに

Python チュートリアルに参加する機会がありました。深層学習について、内容をまとめました。

「深層学習」は、上図に示すように、1950 年代から提唱された「人工知能」の一つとなります。
2010 年代から始まりますが、機械学習のうちニューラルネットワークと呼ばれる手法の進化形です。TensorFlow を使用します。ライブラリではありますが、フレームワークと呼ばれ、機械学習の全体の処理の流れが実装されてるとも言われます。 2017 年に、TensorFlow2.0 として、ニューラルネットワークの処理に優れる Keras を統合し、より使い勝手が、上がったようです。

本報告書では、 tensorflow.keras を使用したプログラムを紹介しますが、ブラックボックス化されていて、簡単というか、中身が見えてこないです。まずは、どういうものか？を説明する報告書にしたいと思ってます。 また、題材として、手書き文字の分類を、行います。

実行環境は、Google が提供する Colaboratory で行いました。この環境は、CPU ではなく GPU で演算処理を行います。（GPU は画像処理用のデバイスとして、耳にしますが、深層学習においても、並列計算のできる演算処理が必要なようで、NVIDIA 製が親和性が良いようです）
また、TensorFlow2.0 のインストールも完了してるので手間が減ります。

機械学習は、データからパターンを見つけて予測するという手法ですが、深層学習は、さらに、パターンを見つける作業も、一次データから特徴量を抽出する作業も、コンピュータにやってもらうという手法です。
（とはいえ、全体のデザインとか、フィルターの選択等は、人の作業になります）

報告書の構成は　１.はじめに　(ここのページです)

２．深層学習　　　　　：ニューラルネットワークについての話です。

３．手書き文字の分類　：今回の題材である、画像の分類に不可欠な

ソフトマックス関数、畳み込み層、プーリング層についての話です。

４．プログラムの説明　：定番の手書き文字の分類のプログラムに説明を追加しました。

５．おまけ　　　　　　：今回の勘所の一つであるパラメータの求め方について追加してます

２，３，まで、読んで、（参考文献に元にしてるので、間違いはないと思います）

さらに、分かりにくいところ、知りたいところは、文献、WEB 等で、調べていただければ、と思います。

４のプログラムは、興味があれば見れば良い程度です。

手書き文字の分類のプログラムが、他の画像認識に展開できるとは思えなかったです。

画像認識について　奥が深い（まさに深層）と思う事

手書き文字の分類では、55,000 枚の学習用画像を用いています。 先日、読んだ文献で、

画像による溶接の溶融池の特徴抽出には、22,000 枚の学習用画像を準備したとありました。

分類の程度によるでしょうが、万単位の学習データが必要なようです。

tensorflow を開発した Google が提供してる無料アプリに ‘レンズ’ があります。

知りたい対象物を、画像に取り込んで、分類して何であるか教えてくれるアプリです。

私は、植物の名前を知りたく、利用しますが、正解と思う事もありますが、沢山の候補を、提示したり、

明らかに　誤りの名前が出てきたりします。学習データが少ないのか、画像処理の工夫がたりないのか、

と思ったりします。（試したことはないですが、植物名検索アプリは、有料のものもあります。）

参考文献

TensorFlow と Keras 動かしながら学ぶディープラーニングの仕組み

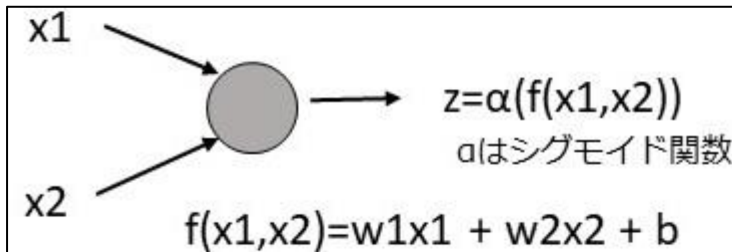
中井　悦司　　株式会社マイナビ出版

----

2 深層学習

2.1 ニューラルネットワークの説明

以下の図が、ニューラルネットワークの最小ユニットで、ニューロン、もしくはノードと呼ばれます。



実際の演算は、変数も多いでしょうし、ノードも多いと思いますが、この単純形で説明します。

目標は、x1,x2 の入力値に対して、出力 f(x1,x2)を予測するということで、以下のステップで行います。

① 出力値を予測する 数式を考えます。上記では f(x1,x2)=w1x1 + w2x2 + b です。

（x が 乗算を意味する'掛ける'ではなく 変数 x ですのでアンダーバーを追加しました）

パラメータである w1,w2,b の最適値を求めることになります。

② パラメータの最適化を行う誤差関数を定義します。

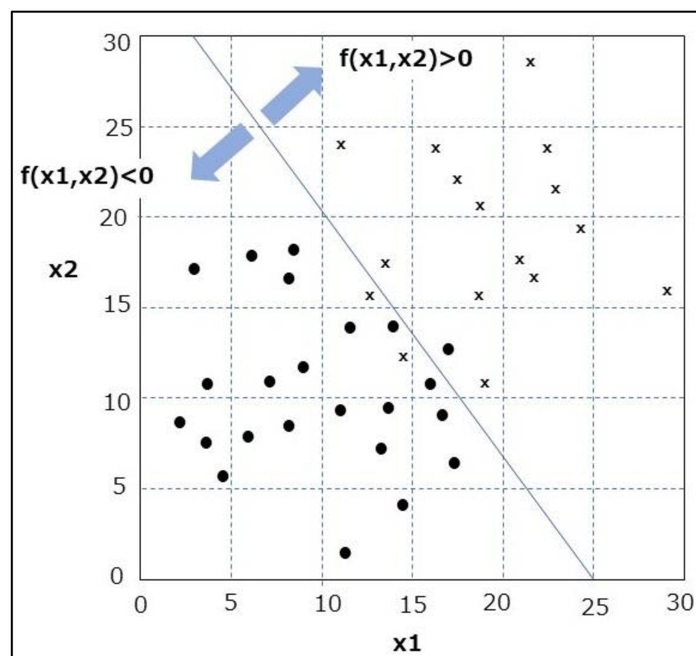
③ 誤差関数を最小するパラメータを求めます。（コンピュータによる計算です）

以上が概略ですが、もう少し説明します。

下の図は、x1,x2 は検査値 ●は非感染、x は感染というの散布図と考えてください。

（リアルなデータではなく、説明用に、作成した図です）

N 個のデータがあるとします。



上述の３ステップに分けます。

① 数式を考える

非感染と感染を分岐する、左上から右下に伸びる直線があります。

直線を、 $f(x1,x2) = w1\underline{x1} + w2\underline{x2} + b = 0$ として

$f(x1,x2)>0$ で、感染　$f(x1,x2)<0$ で非感染 と考えます。

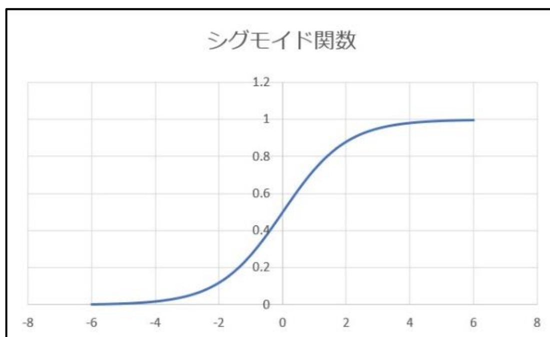
② 誤差関数（E）の定義

　出力を 0～１の確率の値に変換し、確率の扱いで、定義します。

$P(x1,x2) = \alpha(f(x1,x2)) = \alpha(w1\underline{x1} + w2\underline{x2} + b)$

$\alpha(x)$ は シグモイド関数といい、下図のように 0～１へ滑らかに変化します。

（活性化関数の一つで、他にも 同様の関数はあります）



前ページのグラフの N 個の点について、全ての点について、正解の確率を算出します。

（学習データで、非感染、感染の判別はできています。）

全 N 個の確率 P は、各１個づつの確率の掛け算ですので、以下のようになります。

$P = P1 \times P2 \times 1 \cdots \times Pn$

P が最大になるようなパラメータを見つけるのですが、大量の掛け算は、コンピュータの計算に

不向きなため対数関数へ、頭にーをつけて最小化するパレメータの最適値を探します。

(勾配降下法のアルゴリズムを適用します。 ５章のおまけ も参照ください)

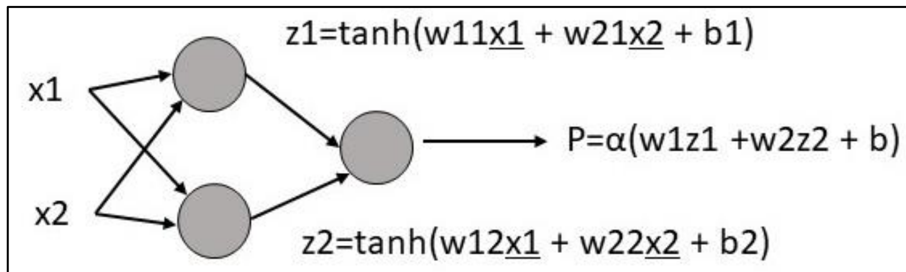
$E = -\log P$

③コンピュータの計算により、E が最小になる パラメータ w1,w2,b を算出します。

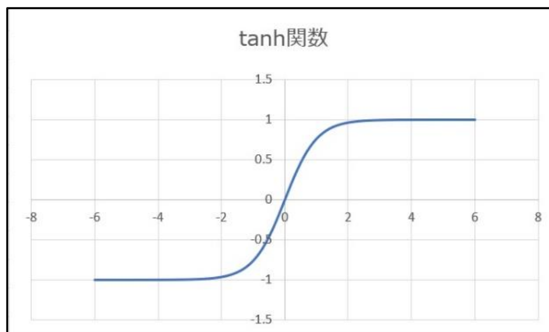
<u>深層学習は直線での分類であり、曲線は定義できません。直線の数を増やすことで、相応としてます。</u>

2.2 単層ニューラルネットワーク

上述の最小ユニットに、下図のように、2 個のノードを持つ中間層（隠れ層）を追加したもので、考えてみます。　散布図を 2 本の直線で分割して、4 つの領域にします。

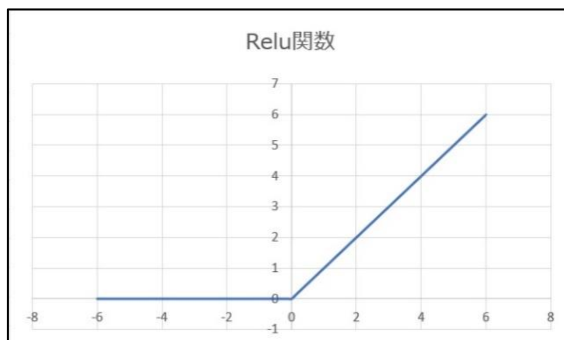


活性化関数として、ハイパボリックタンジェント tanh(x) を用いています。



立上りが早く、-1 から 1へ 変換されます。

ついでに、 ReLU 関数も紹介します。（手書き文字の分類で使用されてます）



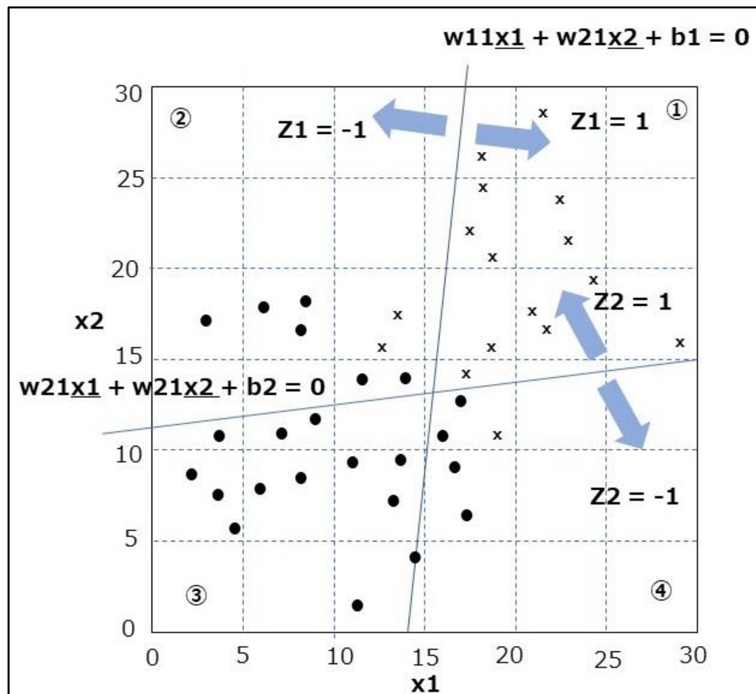
パラメーターの最適化が高速で行えます。

（シグモイド関数とハイパボリックタンジェント関数は、x の増加により、傾きが 0 になるので勾配ベクトルの大きさが小さくなり、最適化処理が進みにくくなるようです。）

パラメータは、w1,w11,w12,w2,w21,w22,b,b1,b2 の 9 個になります。

次ページに散布図を示します。ノードは 2 個ありますので、2 個の直線で、分割します。

直線は w11x1 + w21x2 + b1 = 0 と w21x1 + w22X2 + b2 = 0 です。

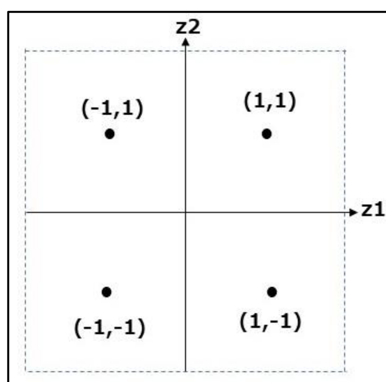


出力は $z1 = \tanh(w11\underline{x1} + w21\underline{x2} + b1)$

$z2 = \tanh(w12\underline{x1} + w22\underline{x2} + b2)$ です。

ハイパボリックタンジェント関数で、変換しているので、 1 または -1 に近い値になります。

すなわち、 z1 と z2 の散布は 以下のようで、(z1,z2) の数値を示しています。

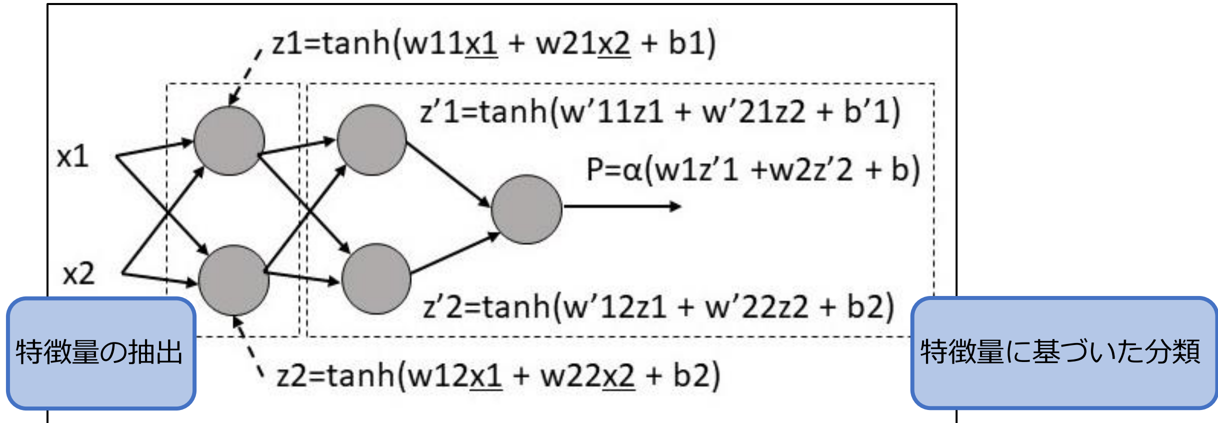


出力層の出力は、確率値 $P = α(w1z1 + w2z2 + b)$ となり、

2.1 と同様に、9 個のパラメータの最適値を計算します。

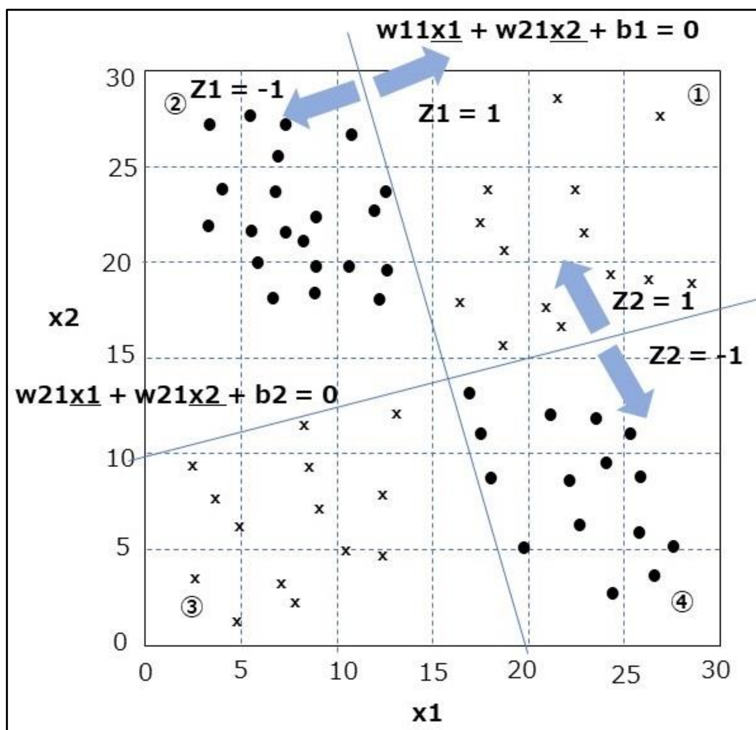
2.3 多層ニューラルネットワーク

次は、以下のように、中間層（隠れ層）2 層になった場合を、考えます。



パラメータは、w1,w11,w'11,w12,w'12,w2,w21,w'21,w22,w'22,b,b1,b'1,b2,b'2

の 15 個になります。 以下のような散布図の時に有効です。



直線での分割において、

互いに交差する位置に異なる

タイプのデータがある場合,

直線での分類はできません。

左図で示すと、

①と③が x データで、

②と④が●データです。

直線では無理です。

このような場合、2 層目の

中間層（隠れ層）で

処理して分類します。

1 層目の z1、z2 を「特徴変数」といい、特徴量を抽出します。（まずは、特徴を見てみましょう ）

2 層目で、特徴量の判定をおこない、確率 P に落とし込んで、最適値を計算します。

判定は、コンピュータの計算の中で処理されますが、論理回路の XOR 回路を構成しています。

（(z1,z2) を (1,1) ,(-1,-1) と (1,-1),(-1,1) のグループに分類する手法です。）

3 手書き文字の分類

3.1 概略

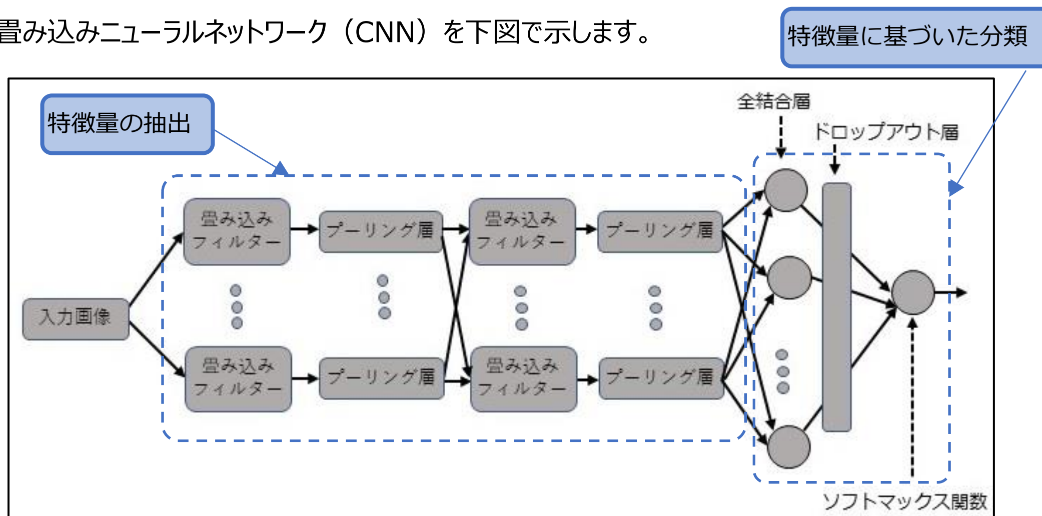
深層学習の応用例として、画像処理、自然言語処理　等があり、ここでは、画像処理のひとつである手書き文字の分類に、取り組みます。

（画像処理は身近な存在で、溶接学会誌に、溶融池の画像を、深層学習の適用により、特徴量を抽出する内容がありました。）

今回の手書き文字の分類は、画像としては　28 x 28 ピクセルの画像で、ピクセル 784 個の集まりであり、784 次元空間上に配置された点の集合になります。

前項までの 2 次元（x1,x2）が、784 次元（x1,x2,・・・・・・・,x784）になります。
同じ数字に対応する画像は、784 次元空間上で互いに近い場所に集まっているだろうという想像します。この空間を 10 個の領域に分割して、その領域に対応する数字を予測していきます。

畳み込みニューラルネットワーク（CNN）を下図で示します。



入力画像を、いきなり分類しても正解率はあがりません。特徴量の抽出する層を追加して、効率を上げます。（畳み込みフィルター・プーリング層と記されている箇所です。）

前項までの感染の有無の分類ではなく、10 個の領域に分類します。ソフトマックス関数による確率の変換という用法で対応します。

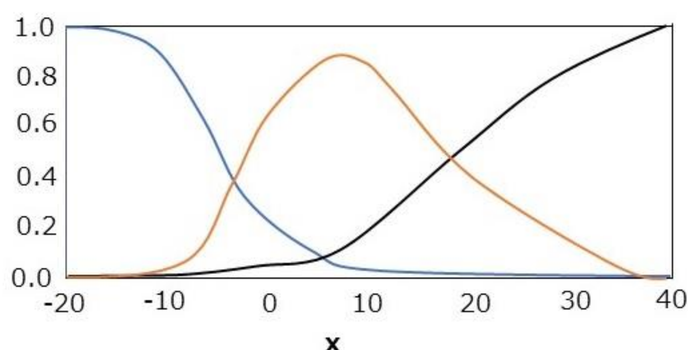
以下、ソフトマックス関数、畳み込みフィルター、プーリング層　について　説明します。

（ドロップアウト層は、過学習に対応するもので、ノードを一部使用しないしくみです。詳細は次の機会）

3.2 ソフトマックス関数

ソフトマックス関数により確率への変換にしますが、3 つに分類するとして、各確率の合計値を 1 となるようになります。1 次元で 3 つの領域に分類する例ですが、下図に示します。

（グラフ作成したかったのですが、関数が見つからず、手書きで作成しました、滑らかでなくすいません。）



繰り返しになりますが、ソフトマックス関数の出力は、0 から 1.0 の実数になり、出力の総和は、1 になります。これは、「確率」です。手書き文字認識では、0～9 の文字ですので、10 個の分の確率を算出して、最も「確率」の高い文字に分類していきます。

このソフトマックス関数を、M 次元空間を、K 個の領域に分類する場合の数式は、

全部で K 個の一次関数を用意します。

$f_k(x_1, \cdots\cdots, x_M) = w_{1k}x_1 + \cdots\cdots\cdots\cdots + w_{Mk}x_M + b_k \quad (k = 1, \cdots, K)$

点$(x_1, \cdots\cdots, x_M)$ が k 番目の領域である確率は、ソフトマックス関数を用いて、以下となります。

$P_k(x_1, \cdots\cdots, x_M) = e^{fk(x1, \cdots\cdots, xM)} / \sum_{i=1}^{K} e\, fk(x1, \cdots\cdots, xM)$

手書き文字の分類に適用すると、784 次元を 10 個に分類するので、以下となります。

$f_k(x_1, \cdots, x_{784}) = w_{1k}x_1 + \cdots + w_{784k}x_{784} + b_k \quad (k = 1, \cdots, 10)$

$P_k(x_1, \cdots\cdots, x_{784}) = e^{fk(x1, \cdots\cdots, x784)} / \sum_{i=1}^{10} e\, fi(x1, \cdots\cdots, x784)$

次に、2.1 で示した、正解になる確率を最大化するパラメータを算出するという計算を行って、

パラメータを設定します。

3.3 畳み込みフィルター

入力されて画像に、フィルターをかけて、特徴を分かりやすくする手法です。ここでは、28x28 のピクセルに、縦、横のエッジを抽出するフィルタを持ちます。

1 例として、左に画像の各ピクセルの色濃度の配列しました。　中央が、縦エッジを抽出するフィルタで、右が計算の結果です。

3	0	2	2
1	0	3	1
3	2	2	0
1	2	2	1

-1	0	1
-2	0	2
-1	0	1

a=2	c=2
b=1	d=4

a = 3x-1 + 0x0 + 2x1 + 1x-2 + 0x0 + 3x2 + 3x-1 + 2x0 + 2x1 =2

b = 1x-1 + 0x0 + 3x1 + 3x-2 + 2x0 + 2x2 + 1x-1 + 2x0 + 2x1 =1

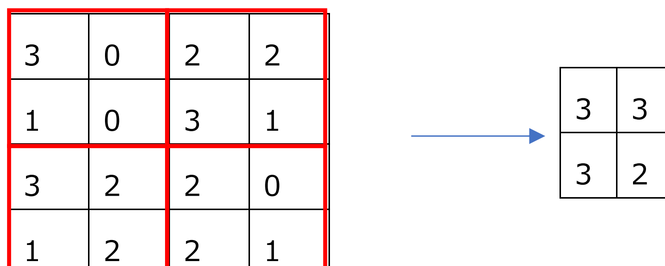
c = 0x-1 + 2x0 + 2x1 + 0x-2 + 3x0 + 1x2 + 2x-1 + 2x0 + 0x1 =2

d = 0x-1 + 3x0 + 1x1 + 2x-2 + 2x0 + 0x2 + 2x-1 + 2x0 + 1x1 =-4 (絶対値)

このフィルタは、横に伸びたエッジを、プラスとマイナスでキャンセルするので、縦に伸びたエッジを抽出します。（この例では顕著ではないですが）また、4x4 の枠が、2x2 になるので、あらかじめ周囲に、0 の枠を追加して、フィルタリングすることも行います。（padding と言います）

3.4 プーリング層

特徴を明確にするため（白いか黒いかが分かれば良い）ので、解像度を落として、詳細情報を消します。具体的には、28x28 ピクセルの画像を、2x2 ピクセルをブロックにして、14x14 のピクセル画像に、変換します。ピクセル値は、ブロック内の最大値を採用します。



3	0	2	2
1	0	3	1
3	2	2	0
1	2	2	1

3	3
3	2

4.プログラムの説明

```
# 基本的なライブラリ
import numpy as np
import matplotlib.pyplot as plt
from pandas import DataFrame
```

＃の後は、コメント文でプログラムではないです

import　○○○ でライブラリをインポートします。
numpy, pandas ,matplotlib は標準的なもので、
今回 tensorflow をインポートしました。
mnnist はライブラリに用意されている MNIST(手書き画像のデータ)で,ダウンロードします。

```
# データセットのためのライブラリ
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# MNIST データをダウンロードして NumPy の配列に変換
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape(
                    (len(train_images), 784)).astype('float32') / 255
test_images = test_images.reshape(
                    (len(test_images), 784)).astype('float32') / 255
```

ピクセルの濃度を 0 ～ 1 の浮動小数点に変換します。

```
train_labels = tf.keras.utils.to_categorical(train_labels, 10)
test_labels = tf.keras.utils.to_categorical(test_labels, 10)
```

正解のラベルを、ワンホット・エンコーディングに変換します。

```
## 学習データの 1 枚目のデータを確認
train_images[0]
```

実際は 784 個の出力です。省略してます。

出力内容

```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.21568628, 0.6745098 ,
       0.8862745 , 0.99215686, 0.99215686, 0.99215686, 0.99215686,
       0.95686275, 0.52156866, 0.04313726, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        ], dtype=float32)
```

```
# 1 枚の画像は 28×28 ピクセルのデータ
train_images[0].shape
```

出力内容

```
(784,)
```

```
# 1 枚目の画像は"5"を表している
train_labels[0]
```

ワンホット・エンコーディングという表現方法です。
６番目が　１　になっているので、　５　になる

出力内容

```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

```
# 先頭 10 枚の画像を視覚的に確認してみる
fig = plt.figure(figsize=(8, 4))
for c, (image, label) in enumerate(zip(train_images[:10], train_labels[:10])):
    subplot = fig.add_subplot(2, 5, c+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.set_title('%d' % np.argmax(label))
    subplot.imshow(image.reshape((28, 28)),
                   vmin=0, vmax=1, cmap=plt.cm.gray_r)
```

figsize=(8,4) は図面サイズで、単位はインチです。

for ループで、enumerate()関数 images と labels を 引数にしている

2 が行の数、5 が列の数、c+1 が何番目の配置

目盛は無しの設定

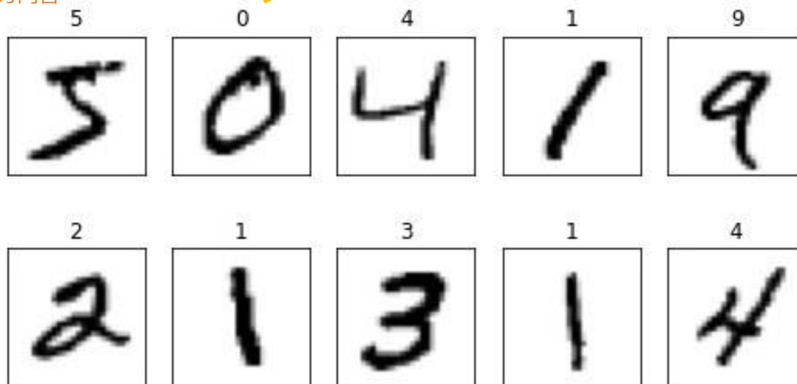
リストの中から最大値をとる要素のインデクスを返す。

28x28 の 2 次元データに変換している。

vmin,vmax の値はカラーマップの下限と上限、cmap はカラーマップの定義

画像上の数字は、正解ラベルから取得した数字です

出力内容




```
# ニューラルネットワークを使うためのモジュールを import
from tensorflow.keras import layers, models, initializers
# 乱数のシードをセット
np.random.seed(20190222)
tf.random.set_seed(20190222)
```

トレーニングデータを乱数で作成しますが、実行ごとに、
乱数が変化することを避けるため、乱数のシードを設定します。

```
# モデルを定義
model = models.Sequential()
model.add(layers.Reshape((28, 28, 1), input_shape=(28*28,), name='reshape'))
```

手書き文字を分類する モデル を作成します。

1 次元の画像データを 28 x 28 ピクセルの 2 次元形式に変更している。

```
model.add(layers.Conv2D(16, (5, 5), padding='same',
                        kernel_initializer=initializers.TruncatedNormal(),
                        use_bias=True, activation='relu',
                        name='conv_filter'))
```

layers.Conv2D で乱数で初期化した畳み込みフィルターを適用している、5x5 サイズの 16 個のフィルターを用いる。
padding='same' は、画像の端に値 0 のピクセルを追加するということ。
kernel_initializer=initializers.TruncatedNormal() は フィルターの値を乱数で初期化するということ
Use_bias=True, activation='relu' は、定数を加えて、Relu を適用するということ

```
model.add(layers.MaxPooling2D((2, 2), name='max_pooling'))
```

画像サイズを、14x14 に縮小する　　プーリング層

```
model.add(layers.Flatten(name='flatten'))
```

16 種類のフィルターから得られた 16 枚の画像データを、全て 1 列に並べた 1 次元リスト形式に変換する。

```
model.add(layers.Dense(1024, activation='relu',
                       kernel_initializer=initializers.TruncatedNormal(),
```

1024 個のノードを持つ中間層（隠れ層）での分類処理　活性化関数は、ReLU,
kernel_initializer=initializers.TruncatedNormal() は、パラメータを乱数で初期化

```
model.add(layers.Dense(10, activation='softmax', name='softmax'))
model.summary()
```

ソフトマックス関数による確率値変換

出力内容

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 28, 28, 1)         0

conv_filter (Conv2D)         (None, 28, 28, 16)        416

max_pooling (MaxPooling2D)   (None, 14, 14, 16)        0

flatten (Flatten)            (None, 3136)              0

hidden (Dense)               (None, 1024)              3212288

softmax (Dense)              (None, 10)                10250

=================================================================
Total params: 3,222,954
Trainable params: 3,222,954
Non-trainable params: 0
_________________________________________________________________
```

3.1 の図にある畳み込みフィルター

3.1 の図にあるプーリング層

3.1 の図にある全結合層
（隠れ層・中間層）

3.1 の図にあるソフトマックス関数

パラメータの数

モデルを作成しました。以下で、データを検証します。

訓練プロセスを設定します。

```
# モデルをコンパイル
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

最適化アルゴリズムは Adam Optimizer

損失関数は'categorical_crossentropy'

正解率は'acc'を指定する

```
# モデルを学習させる
history = model.fit(train_images, train_labels,
                    validation_data=(test_images, test_labels),
                    batch_size=128, epochs=10)
```

訓練を実行します

validation_data に テストデータを与えることで、テストデータの評価も分かります

batch_size は小分けする塊です。epoch は学習の回数です。

出力内容

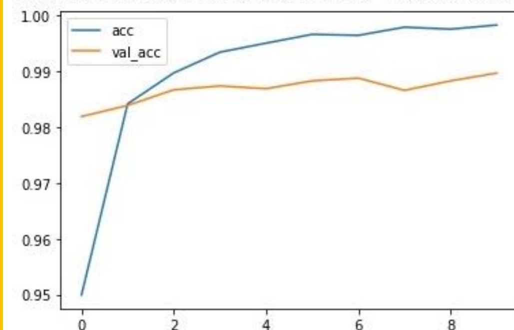
```
Epoch 1/10
469/469 [==============================] - 12s 7ms/step - loss: 0.1723 - acc: 0.9499 - val_loss: 0.0590 - val_acc: 0.9819
Epoch 2/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0510 - acc: 0.9842 - val_loss: 0.0483 - val_acc: 0.9839
Epoch 3/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0316 - acc: 0.9897 - val_loss: 0.0395 - val_acc: 0.9867
Epoch 4/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0211 - acc: 0.9934 - val_loss: 0.0382 - val_acc: 0.9874
Epoch 5/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0157 - acc: 0.9951 - val_loss: 0.0425 - val_acc: 0.9869
Epoch 6/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0101 - acc: 0.9966 - val_loss: 0.0395 - val_acc: 0.9883
Epoch 7/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0103 - acc: 0.9965 - val_loss: 0.0353 - val_acc: 0.9888
Epoch 8/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0067 - acc: 0.9979 - val_loss: 0.0473 - val_acc: 0.9866
Epoch 9/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0071 - acc: 0.9976 - val_loss: 0.0386 - val_acc: 0.9883
Epoch 10/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0055 - acc: 0.9983 - val_loss: 0.0348 - val_acc: 0.9897
```

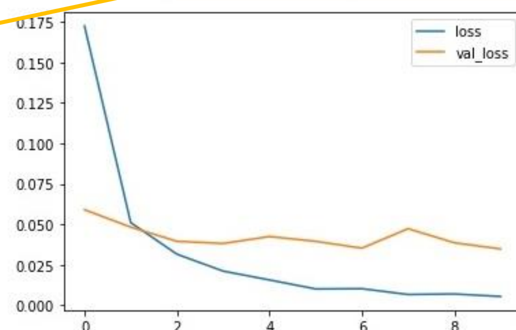
テストデータについて、10 エポック分学習すると、
正解率は 98%になりました。

```
# accuracy と loss の推移を確認
DataFrame({'acc': history.history['acc'],
           'val_acc': history.history['val_acc']}).plot()
DataFrame({'loss': history.history['loss'],
           'val_loss': history.history['val_loss']}).plot()
```

出力内容

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb15656a750>
```





正解率と損失関数の値の変化

畳み込み層適用後とプーリング層適用後の画像を確認します。

```
# 中間層(畳み込み層とプーリング層)の出力を取り出すモデルを定義
layer_outputs = [model.get_layer('conv_filter').output,
                 model.get_layer('max_pooling').output]
model2 = models.Model(inputs=model.input, outputs=layer_outputs)
```

中間層のモデルとして、model2 を定義します。

```
# テストデータの先頭 9 枚に対する中間層の出力
conv_output, pool_output = model2.predict(test_images[:9])
# 学習後の畳み込み層のフィルターの重み
filter_vals = model.get_layer('conv_filter').get_weights()[0]
print(filter_vals)
```

フィルターの重みです
以下は全部ではなく、一部のみの表示です。

出力内容

```
1/1 [==============================] - 0s 87ms/step
[[[[ 2.10731830e-02  2.39050426e-02  6.53755143e-02  5.73902614e-02
    -2.04500277e-02 -1.53147489e-01 -2.87418693e-01  1.88237187e-02
    -2.98456013e-01 -2.56794151e-02 -4.78448905e-02 -1.70669088e-03
     2.06493703e-03  4.09319401e-02  6.38662130e-02  1.16898805e-01]]

  [[ 1.37885436e-01  3.49433944e-02 -1.25139533e-02  4.26757000e-02
    -1.08488113e-01 -1.60996154e-01 -2.27820233e-01 -1.45325765e-01
    -1.89491376e-01 -8.43290240e-04  1.00511089e-01 -2.75202710e-02
    -1.48056582e-01 -1.02151006e-01 -5.05183777e-03 -3.26200165e-02]]

  [[ 1.48944005e-01  1.12050967e-02  1.17736004e-01  4.72931750e-02
    -1.54439270e-01 -1.43687770e-01 -2.55924881e-01  4.19733599e-02
     3.67865264e-02  6.21294379e-02  3.15535031e-02  9.21895448e-03
    -8.79997686e-02  6.19369373e-02 -1.87014028e-01 -2.39663031e-02]]

  [[ 1.80186912e-01  5.75842038e-02  5.64385839e-02 -9.19475108e-02
    -1.81308761e-01  9.31672305e-02 -2.03910023e-01  4.10782024e-02
     6.50881305e-02  5.05834781e-02  3.01272813e-02  1.75959878e-02
    -1.70265079e-01 -4.09144908e-02 -6.40406832e-02  4.34476277e-03]]

  [[ 1.38248414e-01  7.95020610e-02 -3.28928009e-02 -2.69635171e-01
     7.78428763e-02  3.04930005e-02 -1.44473776e-01  4.92738336e-02
    -2.52300389e-02 -9.82796624e-02 -1.52772889e-01 -1.73902418e-02
    -1.22927696e-01 -1.13826394e-01  1.27528936e-01  1.29093483e-01]]]


 [[[-6.34023920e-02  8.22957605e-02 -3.50568793e-03  1.69601990e-03
    -1.38512731e-01 -7.36057311e-02 -2.66189516e-01 -1.76440269e-01
     5.48242331e-02 -4.15914059e-02  4.31910902e-02  9.32806823e-03
     1.18628129e-01  1.15761675e-01  1.17963389e-01  2.17187434e-01]]
```

畳み込み層適用後の画像を確認します。

```
# 畳み込み層でフィルタリングした後の画像(畳み込み層の出力)を確認
num_filters = 16
fig = plt.figure(figsize=(10, num_filters+1))
v_max = np.max(conv_output)
##############
for i in range(num_filters):
    subplot = fig.add_subplot(num_filters+1, 10, 10*(i+1)+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(filter_vals[:,:,0,i], cmap=plt.cm.gray_r)
```

A 部の表示の設定です。

```
##############
for i in range(9):
    subplot = fig.add_subplot(num_filters+1, 10, i+2)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.set_title('%d' % np.argmax(test_labels[i]))
    subplot.imshow(test_images[i].reshape((28,28)),
                   vmin=0, vmax=1, cmap=plt.cm.gray_r)
```

B 部の表示の設定です。

```
##############
    for f in range(num_filters):
        subplot = fig.add_subplot(num_filters+1, 10, 10*(f+1)+i+2)
        subplot.set_xticks([])
        subplot.set_yticks([])
        subplot.imshow(conv_output[i,:,:,f],
                       vmin=0, vmax=v_max, cmap=plt.cm.gray_r)
```

C 部の表示の設定です。

次ページのフィルターで、特定方向のエッジを、抽出して、特徴を求めています。

A.16 種のフィルターを画像化したもの

B.オリジナルの画像データ

C.フィルターを適用した結果

出力内容

7 2 1 0 4 1 4 9 5

プーリング層適用後の画像を確認します。

```
# プーリング層を適用した後の画像を確認
num_filters = 16
fig = plt.figure(figsize=(10, num_filters+1))
v_max = np.max(pool_output)

for i in range(num_filters):
    subplot = fig.add_subplot(num_filters+1, 10, 10*(i+1)+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(filter_vals[:,:,0,i],
                   cmap=plt.cm.gray_r, interpolation='nearest')

for i in range(9):
    subplot = fig.add_subplot(num_filters+1, 10, i+2)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.set_title('%d' % np.argmax(test_labels[i]))
    subplot.imshow(test_images[i].reshape((28,28)),
                   vmin=0, vmax=1, cmap=plt.cm.gray_r)

    for f in range(num_filters):
        subplot = fig.add_subplot(num_filters+1, 10, 10*(f+1)+i+2)
        subplot.set_xticks([])
        subplot.set_yticks([])
        subplot.imshow(pool_output[i,:,:,f],
                       vmin=0, vmax=v_max, cmap=plt.cm.gray_r)
```

プログラムは畳み込み層のものと同等

画像をぼかしています。おおまかな画像で特徴を。顕著にしています。

プーリング層で画像を縮小した

出力内容

7 2 1 0 4 1 4 9 5

正しく分類できなかった 10 例について、確率値を検証します

```
 #  正しく分類できなかった画像に対する予測結果を確認
fig = plt.figure(figsize=(12, 10))
c = 0
for (image, label) in zip(test_images, test_labels):
    image
    p_val = model.predict(np.array([image]))
    pred = p_val[0]
    prediction, actual = np.argmax(pred), np.argmax(label)
    if prediction == actual:
        continue
################
    subplot = fig.add_subplot(5, 4, c*2+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.set_title('%d / %d' % (prediction, actual))

    subplot.imshow(image.reshape((28, 28)),
                    vmin=0, vmax=1, cmap=plt.cm.gray_r)
#################
    subplot = fig.add_subplot(5, 4, c*2+2)
    subplot.set_xticks(range(10))
    subplot.set_xlim(-0.5, 9.5)
    subplot.set_ylim(0,1)
    subplot.bar(range(10), pred, align='center', edgecolor='b')
#################
    c += 1
    if c == 10:
        break
```

predict で確率値を取得します。

確率値の大きい値を、pred とします。

pred と label の値が同じであれば、そのまま継続され、
同じでない時（不正解の時） 以下の作業をします。

5 行 4 列で、1 番目、3 番目･･･の位置に表示されます

タイトルの表示は、予測/正解 とします。

5 行 4 列で、2 番目、4 番目･･･の位置に表示されます

目盛値の設定

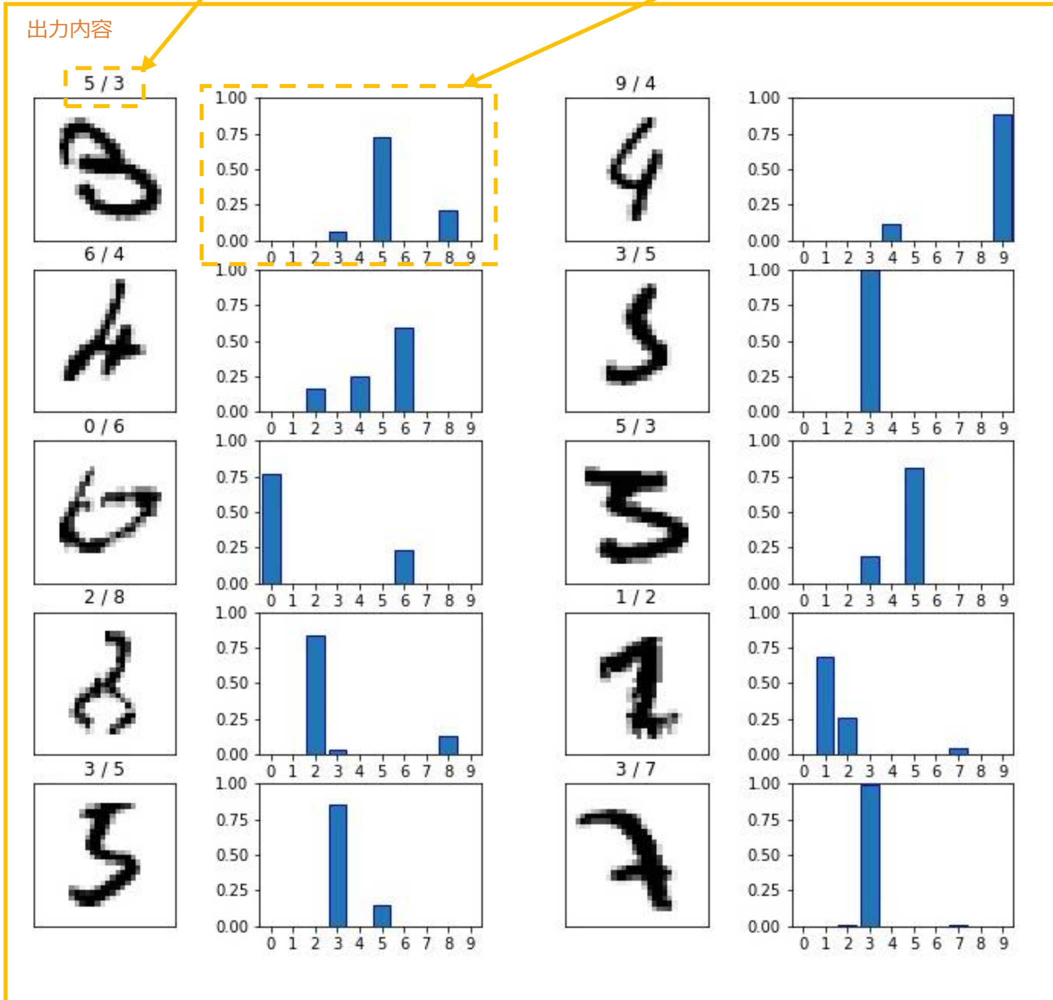
X 軸の数値設定

Y 軸の数値設定

10 例終わったら、やめます。

正解は３であるが、
５と分類される確率が高かった。

正解は３であるが、
５， ８　の確率が高かった。



これで　終わりですが、計算量が大きいので、実装されるときは、演算処理の能力の検証はしてください。

5 おまけ　（パラメータの求め方）

5.1 勾配降下法

　2.1 項で、誤差関数の最小値を探して、パラメータを設定する話がありました。

これが、演算処理の主なところですので、説明を追加します。

　まずは、1 次元で説明します。

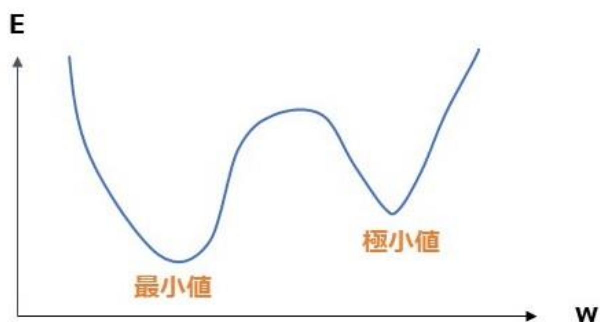
E = f(ｗ)として、

ｗがパラメータ、E は誤差関数です。

最小値は、勾配の傾きが 0 の点で

微分して求めます。

　放物線であれば、簡単ですが



右図のような、2 個の放物線の構成の場合、最小値でなく、極小値が選ばれる可能性があります。

特に、多次元の場合、その可能性は大きいです。

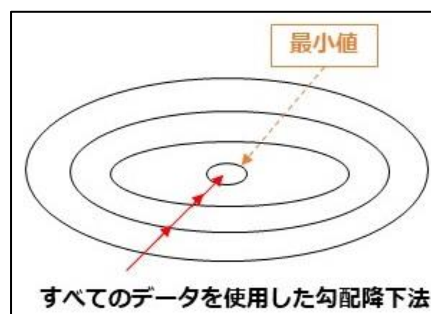
　その回避法として、確率的勾配降下法（SGD）があります。これは誤差関数の計算に、全てのデータを使うのではなく、データをミニバッチに分けて、1 回分のバッチについて、計算していきます。

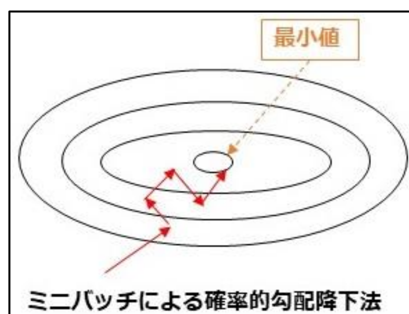
上述のプログラム（14 ページ目）で　‘ batch_size=128 ’ と入力しました。

イメージを示すと（楕円は 3 次元曲面とみてください。最小値を底にする円錐形状です）

右の確率的勾配降下法は、矢印が 1 回のバッチで、都度計算するので、直線にならず、

ジグザグに動くので、極小値の谷に入っても、抜け出るだろうという考え方です。





　誤差逆伝搬法　という、勾配の求め方を、効率アップした手法があります。

数値微分は、計算に時間がかかるので、出力値から、正解値と差を把握して、パラメータを調整するやり方です。（詳細は　よく理解してません。）

5.2 adam について

　パラメータの設定について、‘ optimizer='adam', ‘ と入力してます。（14 ページ）

パラメータを変化させて、最小の誤差関数のパラメータ値を求める最適化の手法の事です。

　上述の確率的勾配降下法（SGD）は、ジグザクな動きをするので、非効率という欠点があります。

　momentum という手法があります。パラメータの変化させ方を、速度をあげていく という

手法です。正負の動きの場合、交互に打消しあって、滑らかな動きとなります。

（慣性という意味で、ボールが傾斜面を転がるイメージです）

　adagrad という手法があります。学習の深度により、学習率を小さくするこで、効率を上げます。

学習率とは、パラメータの変化度合いを決める値です。小さいと時間がかかりますし、大きいとうまく最適化

できません。最初は大きく、次第に小さくするという調整を行います。

　adam というのは、上記の momentum と adagrad を融合したもです。