はじめての Python と OpenCV による画像処理

Image Processing with Python and OpenCV for Beginners/Eiichiro MOMMA

日本大学 理工学部 電気工学科 門馬英一郎

1. はじめに

画像を入力とし、畳み込み (Convolution) 層を用いた Convolutional Neural Network (CNN) がベースとなった、いわゆる人工知能 (AI) 技術のコモディティ化が進んでいる。これらに関わる最新の技術は次々と発表されるとともに、Github¹⁾などでプログラミング言語 Python²⁾を利用した深層学習フレームワークの Tensorflow³⁾, Keras⁴⁾, Pytorch⁵⁾などを用いた実装が公開され、すぐに試せる状況である。一方で、これらの技術をいざ自身の研究領域に適用するときに、データセットへ落とし込むまでの技術として画像処理に関する知識が必要となる。画像処理については「はじめての精密工学」2006年5月 (Vol. 72, No.5), 2015年9月 (Vol. 81, No.9) に中央大学の梅田先生が基礎的な範囲を、2017年10月 (Vol. 83, No. 10) で香川大学の林先生が実用例を取り上げている。本稿では、読者が画像を扱うことに主眼を置き、Pythonによる画像処理として、主に OpenCV⁶⁾パッケージを使った道具としての画像処理について解説を行う。なお、この記事と連動した情報として筆者の Github⁸⁾で情報を公開したので併せて利用していただきたい (図1)。

Pythonの環境構築については Anaconda⁷⁾の Individual Edition を利用する。Anaconda はパッケージシステムも提供しており、Anaconda Navigator または conda コマンドでパッケージがインストールできる。標準で選択可能なパッケージは限られているので Anaconda Cloud (<https://anaconda.org/>) でさまざまな channel から独自のパッケージが配布され、これらの組み合わせにより希望するパッケージを追加する。それでも不十分な場合には、Python

の標準的なパッケージシステムの pip も利用可能である。本稿での環境を紹介するが、本件にかかわらず何か新しいパッケージセットを試す際には、Anaconda Navigator の Environments から新しい仮想環境を作成した方がよい。本稿では、チャンネルに conda-forge を指定して下記のパッケージを指定して環境構築したが、依存パッケージも同時にインストールされているので、パッケージ一覧にはこれら以外も多数並ぶ。

python=3.7, NumPy, pillow, SciPy, Notebook (Jupyter Notebook), Matplotlib, scikit-image, py-opencv (OpenCV の Python I/F で執筆時のバージョンは 4.2)

OpenCV は Intel が開発した Computer Vision 向けのライブラリで Willow Garage, Itseez へメンテナンスが移管されていたが、2016年に Intel が Itseez を買収している。OpenCV は商用利用可能な BSD ライセンスで公開されている画像処理ライブラリとして広く支持されている。プログラミング言語については C++, Java, Python が公式のチュートリアルなどのドキュメントに組み込まれ公式なサポート言語と考えられるが、Google Summer of Code のプロジェクトで開発された I/F のほか、サードパーティでもさまざまな言語へのラッパーが用意されている。OpenCV は画像の入出力、画像処理、機械学習 (深層学習含む)、3次元計測等、Computer Vision に関連する技術の片っ端から取り込んで肥大化の一途をたどっているが、本稿で取り上げるのは入出力、フィルタ処理などの基本的な部分なので、メジャーバージョンが変わっても多少の変更で対応可能と思われる。詳細は公式ドキュメント⁹⁾での modules に一覧が並んでおり、Python での OpenCV は cv2 という名前で利用でき

```
import cv2
```

でインポートし、「cv2.関数名」または「cv2.モジュール名.関数名」のような形式で呼び出す。Main modules が公式にメンテナンスされ、Extra modules が実験的に採用・廃止されるもので Extra modules から Main modules に昇格する場合もある。Modules はパッケージがビルドされた環境により異なるため

```
import cv2
```

```
print(cv2.getBuildInformation())
```

とすると To be built に一覧が表示される。次章では cv2 を使った画像データの扱いについて解説する。なお、紙数

図1 連動サイト (<https://github.com/eiichromomma/jjspe202010>)

の都合上 (1) Python の基本的文法, (2) Anaconda の操作, (3) Jupyter Notebook の操作, については省略する。

2. 画像データの扱い

OpenCV は 2.0 以降から I/O については MATLAB を意識したような関数名が多く、画像の読み込みは `imread` 関数を使う。画像形式は自動的に判断され、チャンネル数 (多くの場合グレースケールの 1 チャンネルかカラー画像の Red, Green, Blue の 3 チャンネル) については第 2 引数で指定しない限りは自動的に適切なチャンネル数が割り当てられる。

```
import cv2
gimg = cv2.imread('sample.png', cv2.IMREAD_
GRAYSCALE)
type(gimg)
```

とすると `numpy.ndarray` が表示される。つまり、画像データは NumPy はもちろん、ほとんどの深層学習のフレームワークでもそのまま利用可能である。なお、第 2 引数の `IMREAD_GRAYSCALE` はカラーであってもグレースケールで読み込む指定で、逆にグレースケール画像を読むときに `IMREAD_COLOR` とすると強制的に 3 チャンネルカラー画像として読める。次に

```
gimg.shape
```

を実行するとグレースケール画像として読んでいるので (184, 309) と 2 次元データとして表示される。NumPy の変数のため (rows, cols) になるが、OpenCV で座標を指定する際は (x,y) で間違いやすいので座標を扱う際はリファレンスを確認した方がよい。次にカラーで画像を `cimg` 変数に読み、`shape` を見ると

```
cimg = cv2.imread('sample.png')
cimg.shape
```

(184,309,3) のように (rows,cols,channels) という並びのデータになる。OpenCV ではカラー画像は Bitmap の仕様と同様に Blue, Green, Red の順でチャンネルが並ぶ。一方で Matplotlib の `imshow` 関数で可視化しようとする Red, Green, Blue の並びが必要になるため事前に入れ替える必要がある。OpenCV にも `imshow` 関数があるが、ウィンドウに画像を表示する機能で、文章とソースコードを実験ノートのように一緒に書ける Jupyter Notebook と相性が悪い。Notebook では文章内に inline 表示できる Matplotlib の `imshow` 関数を使うのが一般的で、`cvtColor` 関数で BGR から RGB へ変換した結果を渡す。また、グレースケール画像の場合には疑似カラー表示がデフォルトなのでカラーマップとして `cmap='gray'` を指定する必要がある。

```
%matplotlib inline
# カラー画像を読んで表示する
import cv2
import matplotlib.pyplot as plt
cimg = cv2.imread('sample.png')
```

```
plt.imshow (cv2.cvtColor (cimg,cv2.COLOR_
BGR2RGB))
# グレースケール画像を読んで表示する
gimg = cv2.imread ('sample.png',cv2.IMREAD_
GRAYSCALE)
plt.figure()
plt.imshow (gimg, cmap='gray')
```

画像の保存については `imwrite` 関数を使う。ファイル形式については拡張子で自動的に判断されるのでファイル名と保存したいデータを指定するだけである。

```
cv2.imwrite ('cdst.png',cdst)
```

この章の最後に動画の扱いについて紹介しておく。Matplotlib の animation 機能と `nbagg` (Notebook 上でのインタラクティブな描画機能) を利用するとカメラの映像を処理した結果を連続表示できる。ただし、ラグがあり筆者の環境 (Mac mini 2018, Intel Core i5) では 5 fps 程度となる。図 1 のサイトで公開している Notebook の 2_IO にカメラ画像のモザイク処理を、3_Kihon の 3.1 にはフレーム間差分処理の例を示した。ただしコールバック関数の `animate_func` 内のデバッグは難しいので、事前に静止画での動作確認をお勧めする。なお、`nbagg` での描画とセルの実行完了のタイミングは非同期なので、カメラを `release` するのは別のセルで描画処理が終わってからにする必要がある。

3. 基本的な画像処理

入出力の次に基本的な画像処理として、本誌で過去紹介された処理について解説する。なお、著者のサイトでは本章の 3_Kihon のほかに、本稿で紹介できなかった処理の例として、画像処理のトレーニングとして公開されている「画像処理 100 本ノック」^[10] の手短かな解法も紹介している。なお、以降のソースコードは事前に以下のコードが実行されている前提で記す。

```
%matplotlib inline
import cv2
import numpy as np
import matplotlib.pyplot as plt
gimg = cv2.imread ('sample.png',cv2.IMREAD_
GRAYSCALE)
cimg = cv2.imread ('sample.png')
```

また、処理結果 `dst` の表示については以下のいずれかを用いる。

```
plt.imshow (dst,cmap='gray') # グレースケール
plt.imshow (cv2.cvtColor (dst,cv2.COLOR_
BGR2RGB)) # カラー
```

3.1 画像の差分

画像の差分は、2 次元配列の差分で、NumPy においては単純に引き算の演算子で済む。用途としては動画でのフレーム間差分やアンシャープマスキングである。

```
blured = cv2.GaussianBlur (gimg, (5,5), 3)
```

図2 差分処理の結果

```
dst = np.float32(gimg) - np.float32(blured)
```

差分は負値にもなるので signed な変数に変換してから処理を施した方が結果が見やすい (図2)。

Notebook ではカメラ映像を用いたフレーム間差分も掲載しているが、前の状態を保持・利用する処理の場合については、コールバック関数 `animate_func` の外であらかじめ宣言しておき、関数内で global 宣言をして関数外の変数であることを明示する必要がある。

3.2 二値化処理

画像の濃度値について、ある値(閾値)に対する大小により白、黒で置き換える処理である。OpenCV の機能としても用意されているが、閾値を自分で決定する場合には

```
ndarray[白とする条件式] = 255
```

```
ndarray[黒とする条件式] = 0
```

で済む。閾値の自動決定法を使う場合には `threshold` 関数を用いる。下記では大津の法 (THRESH_OTSU) で、閾値より明るい画素を黒、暗い画素を白としている (THRESH_BINARY_INV)。

```
th, dst = cv2.threshold(gimg, thresh = -1, maxval = 255,
                        type = cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```

また、Red, Green, Blue 各チャネルでの画素値の関係を考えると特定色の画素の抽出も可能である。

3.3 モルフォロジー処理

主に2値画像に適用される処理群で、代表的なものとしては膨張処理 (`dilate`) と収縮処理 (`erode`) やこれらを組み合わせたオープニングとクロージングがある。モルフォロジー処理には Structuring Element (SE, 構造化要素) が必要で、SE と処理対象画像との組み合わせで処理を行う。SE は 3×3 程度の大きさの正方形、十字形、円形などがよく用いられ OpenCV では `getStructuringElement` 関数で取得できる。例えば膨張処理については2値画像に対して SE を走査し、画像の白成分について1画素でも SE が重なれば白 (hit)、それ以外は黒とする処理である。収縮処理については同様な走査で画像の白成分について SE がすべて重なった場合に白 (fit) とする処理である。ま

図3 LoG フィルタの Zero crossing

た、これらはそれぞれ SE の形の最大値フィルタ、最小値フィルタと考えることもできる。

```
th, bimg = cv2.threshold(gimg, thresh = -1, maxval = 255,
                        type = cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU) # 2値化処理
```

```
SE = cv2.getStructuringElement(cv2.MORPH_RECT, (3,3)) # 正方形 (3×3) の SE 取得
```

```
dst = cv2.dilate(bimg, kernel = SE) # 膨張処理
```

膨張と収縮の処理は同回数繰り返せば対象の大きさは変わらないと考えられるので、膨張を N 回の後に収縮を N 回繰り返すクロージング処理は2値化処理で欠損が発生した場所の穴埋めに使われ、収縮を N 回の後に膨張を N 回繰り返すオープニング処理は2値化によって意図せず連結した箇所を離したり、細かいノイズの除去に使われる。OpenCV では `morphologyEx` 関数で利用可能で引数 `op` に処理を指定する。

```
dst = cv2.morphologyEx(bimg, op = cv2.MORPH_OPEN, kernel = SE, iterations = 4) # オープニング
```

3.4 エッジ抽出

線形フィルタとしてのエッジ抽出は微分フィルタが使われる。画像のような離散データは、隣接画素の濃度値の差分が微分値となるものの、そのままではノイズに対して過剰に反応するため平均化フィルタを畳み込んだ Sobel フィルタがよく使われる。ただし、濃度値を抽出するだけで、連結性は考慮されていない。これに対しエッジの方向を考慮した縦方向と横方向の微分値の二乗和より求めた勾配強度に対して、2段階の閾値を設定して連結したエッジを抽出するのが Canny のエッジ抽出アルゴリズムである。OpenCV では `Canny` 関数があり、以下のように使用する。

```
dst = cv2.Canny(gimg, threshold1 = 100, threshold2 = 200, apertureSize = 3)
```

`threshold2` を超える場合はエッジ、`threshold1` 以下では非エッジ、中間の場合には近傍にエッジがあればエッジと判断する。また、エッジの中心を求める方法としては2次微分をボケさせた LoG (Laplacian of Gaussian) フィルタの結果からエッジ中心となる Zero crossing を求める方法もある (図3)。

3.5 テンプレートマッチング

テンプレート画像 (見つけたい箇所の画像) で対象画像を走査し、類似度が高い箇所を一致した場所と見なす処理である。類似度の計算方法は差分を用いる方法 (SSD, SAD) や、相関を用いる方法 (NCC, ZNCC) がある。

図4 テンプレートマッチングの失敗例

ユニークな場所については比較的一致しやすいが、類似した画像の場合には苦戦する。OpenCV では原画像とテンプレート画像と共に類似度の算出方法を指定する。

```
template = cimg[16:35, 16:35, :].copy()
dst = cv2.matchTemplate(cimg, template, cv2.TM_
    CCOEFF_NORMED) # テンプレートマッチング
```

のように原画像から切り抜いた箇所のマッチングは容易だが、電卓のキーのような似たパターンが多く並び、かつ別途テンプレート画像を撮影した場合には抽出対象とテンプレートの大きさが数 pixel 異なるだけで結果が変わる。Notebook 上では `img` と `template` の箇所のコメントを書き換えると実感できる (図4)。

```
cimg = cv2.imread('Calc_s.jpg') # 原画像
template = cv2.imread('CalcO5_s.jpg') # template 画像
dst = cv2.matchTemplate(cimg, template, cv2.TM_
    CCOEFF_NORMED) # ZNCC (図4左下)
```

3.6 特徴点によるマッチング

テンプレートマッチングは演算コストが高い上に、撮影のワークの厳格化が必要となる。特定物体を検出する方法として、特徴量 (feature descriptor) がある。これは画像から特徴的な場所 (特徴点) について、アルゴリズムに基づいて近傍の濃度値から求めるベクトルである。原画像とテンプレートとで特徴点を検出し、各特徴量を比較して最も距離が近い点を同一点と見なす方法である。ロボティクスの分野でも有名な ORB SLAM の ORB は特徴量の名前である。Notebook の 3_kihon では KAZE と ORB の例を示した。

3.7 カメラキャリブレーション

カメラを用いた計測にはカメラキャリブレーションによるカメラパラメータの推定が必要となる。単眼のカメラの場合にはひずみの除去として Undistortion が必要だが OpenCV の公式チュートリアル¹¹⁾に手順が載っている。重要な点はキャリブレーションに使用するチェッカーボードの剛性で、アームにより稼働領域が広いモニターに表示させる、あるいは iPad などのタブレットを使うのが確実な方法である。また、ステレオ視に関しても平行化が必要で、2台のカメラにチェッカーボードが映る状態で複数枚

撮影する必要がある。OpenCV のやや古いドキュメントや、OpenCV のサンプルの `cpp/stereo_match.cpp` を参照するとよい。

3.8 Hough 変換

直線に関する Hough 変換はエッジ画素の座標について Hough 空間への投票を行い、投票数の多い直線の係数を求める方法である。OpenCV では `HoughLines` 関数と、ランダムにピックアップしたエッジ画素を Hough 空間へ投票し、閾値を超えたパラメータの中で最長のものをピックアップし投票に使った画素を除去して繰り返す確率的 Hough 変換の `HoughLinesP` 関数がある。画像全体にわたる長い線成分を抽出したい場合には前者を、部分的な線成分を抽出したい場合には後者を用いる。また、後者は前者よりも速く線の端点が求まり、パラメータで最低限必要な長さや、無視するギャップ長が設定可能である。具体的には

```
edge = cv2.Canny(gimg, threshold1 = 100, threshold2 = 200, apertureSize = 3)
lines = cv2.HoughLinesP(edge, 1.0, np.pi/180.0, threshold = 10, minLineLength = 15, maxLineGap = 5)
```

のようにして端点のリストの `lines` を求め、ループで描画する。

4. おわりに

駆け足ではあったが、基本的な画像処理の紹介と Python による使用法について解説した。画像処理を道具として使いこなすことを主眼としたため、アルゴリズムや実装を理解することは二の次の内容となっているが、これを足掛かりにさまざまな問題解決に役立ててもらえれば幸いである。前述のように、OpenCV は 3 次元の可視化や深層学習の取り込みにも熱心であるが、Qt、VTK、CUDA や、Intel の OpenVINO などを含めた“完全体”の配布はなされておらず、利用できないケースが多い。OpenCV “完全体”の Python 用パッケージの生成はソースコードからコンパイルする必要があるが、ネット上でこれらの情報は手に入るので興味のある方は挑戦してもらいたい。

参考文献

- 1) Github 公式サイト: <https://github.com/>
- 2) Python 公式サイト: <https://www.python.org/>
- 3) Tensorflow 公式サイト: <https://www.tensorflow.org/>
- 4) Keras 公式サイト: <https://keras.io/>
- 5) PyTorch 公式サイト: <https://pytorch.org/>
- 6) OpenCV 公式サイト: <https://opencv.org/>
- 7) Anaconda 公式サイト: <https://www.anaconda.com>
- 8) 本稿との Github での連動サイト: <https://github.com/eiichiro momma/jjspe202010>
- 9) OpenCV 公式ドキュメント: <https://docs.opencv.org/4.2.0/>
- 10) 画像処理 100 本ノック: <https://yoyoyo-yo.github.io/Gasyori100knock/>
- 11) OpenCV 公式チュートリアル: https://docs.opencv.org/4.2.0/dc/dbb/tutorial_py_calibration.html