第1章

NumPyで最低限知っておくこと

Python でデータ解析のプログラムを書く際には、NumPy というパッケージが利用されます。簡単に言えば、NumPy は配列[注1]を効率的に扱うためのデータ構造とその上での演算群を提供しています。NumPy の基本的な使い方を知らないと、Python でデータ解析のプログラム、もちろん Deep Learning のプログラムを書くことができません。ここでは NumPy について最低限知っておかなければならないことをまとめておきます。

NumPy は pip を使えば簡単にインストールできます。

```
> pip install numpy
```

Python の中で NumPy を利用するには、numpy をインポートします。

```
>>> import numpy as np
```

1.1 配列の生成

配列のデータ構造（型）は array です。要素が 1, 2, 3, 4, 5 となっている配列（大きさ 5 のベクトル）は、以下のように作成します。これが基本です。

```
>>> np.array([1,2,3,4,5])
array([1, 2, 3, 4, 5])
```

要素のリストを渡します。要素が 0, 1, 2, ⋯, 9 となっている配列は以下のように作成します。

```
>>> np.array(range(10))
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

注1　1 次元の配列がベクトル、2 次元の配列が行列です。

上記を略したものが以下です。

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

2 × 3 の配列（2 行 3 列の行列）は以下のように作成します。行のリストを要素としたリストを渡します。

```
>>> np.array([[0,1,2],[3,4,5]])
array([[0, 1, 2],
       [3, 4, 5]])
```

配列の形を変えたいときは reshape を使います。

```
>>> np.arange(6).reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

上記では 1 次元の配列を 2 × 3 の配列に変えました。変更前の元になる配列は 1 次元である必要はありません。

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
```

また、変更先の配列の形は 3 次元以上でもよいです。

```
>>> np.arange(27).reshape(3,3,3)
```

```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

配列の形を知りたいときは shape を、配列の要素数だけを知りたいときは size を、それぞれ使います。

```
>>> a = np.arange(60).reshape(10,6)
>>> a.shape  # 配列の形
(10, 6)
>>> a.size   # 要素数
60
```

行列の行数や列数は shape から取り出せます。

```
>>> nrow, ncol = a.shape   # 行数、列数の取り出し
```

以上のことだけ知っていれば配列の生成は可能ですが、もう少し効率的に書くために、以下の関数も知っておいたほうがよいでしょう。

```
# 0.0（実数）が5個ある配列
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# 1.0（実数）が5個ある配列
>>> np.ones(5)
```

```
array([ 1.,  1.,  1.,  1.,  1.])
```

0.0 や 1.0 で初期化しないで、単に指定の大きさの配列だけを作りたい場合は、empty を使います。

```
>>> np.empty(5)
array([4.28587719e-317, 2.50013916e-315, 4.28593647e-317,
       4.28565189e-317, 4.28588904e-317])
```

また、乱数の配列も必要なことがあります。以下は標準正規分布からの 5 つの乱数を要素とした配列を生成します。

```
>>> np.random.randn(5)
array([-0.89329388, -0.21069465,  0.2492592 ,
       -0.25109675, -0.34499488])
```

randn の部分が分布です。2 項分布なら binomial、ポアソン分布なら poisson といったように使います。通常は一様分布 uniform と正規分布 normal だけ知っていれば問題ありません。

```
# 区間 (0,1) の一様分布に従う乱数を 3 個生成
>>> np.random.uniform(0,1,3)
array([ 0.38991823,  0.78536285,  0.05736855])
>>> np.random.normal(1.5,2.0,3)
# 平均 1.5, 標準偏差 2 の正規分布に従う乱数を 3 個生成
array([ 4.33490939,  1.47686956,  2.79181854])
```

分布は色々あるので、必要に応じて調べてください。

また、配列の要素をシャッフルした配列を作ることもよくあります。

```
>>> np.random.permutation(range(6))
array([3, 2, 5, 0, 1, 4])
>>> np.random.permutation(6)  # 上記の省略形
```

```
array([5, 2, 3, 4, 1, 0])
```

同じようなメソッドとして shuffle もありますが、これは配列を破壊的に並べ替えるので、通常は permutation を使うほうが安全です。

また、行列関係としては単位行列の作り方も知っておいたほうがよいでしょう。

```
>>> np.identity(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

1.2 配列の加工

a と b を 2×3 の配列とします。a の右に b を連結させて 2×6 の配列を作るには hstack、a の下に b を連結させて 4×3 の配列を作るには vstack を使います。

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = np.arange(6,12).reshape(2,3)
>>> b
array([[ 6,  7,  8],
       [ 9, 10, 11]])
>>> np.hstack([a,b])
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
>>> np.vstack([a,b])
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
```

```
       [ 9, 10, 11]])
```

2次元の配列（行列）のある行や列を取り除いた配列を作ったり、逆に取り除く行や列からなる配列を作ったりする操作は重要です。リストのスライスの操作ができれば、これらは容易です。

例として、5×6 の配列から2行目と4行目を取り除いた 3×6 の配列を作ってみます。

```
>>> a = np.arange(30).reshape(5,6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29]])
>>> a[[0,2,4],:]
array([[ 0,  1,  2,  3,  4,  5],
       [12, 13, 14, 15, 16, 17],
       [24, 25, 26, 27, 28, 29]])
```

2列目と4列目を取り除いた 5×4 の配列を作るには、以下のようにします。

```
>>> a[:,[0,2,4,5]]
array([[ 0,  2,  4,  5],
       [ 6,  8, 10, 11],
       [12, 14, 16, 17],
       [18, 20, 22, 23],
       [24, 26, 28, 29]])
```

逆に、2列目と4列目からなる 5×2 の配列を作るには、`[0,2,4,5]` の部分が `[1,3]` になるだけです。

```
>>> a[:,[1,3]]
array([[ 1,  3],
```

```
       [ 7,  9],
       [13, 15],
       [19, 21],
       [25, 27]])
```

ある条件にあった値を別の値に置き換えるには以下のようにします。ここでは偶数の値を −1 に置き換えています。

```
>>> a[a % 2 == 0] = -1
>>> a
array([[-1,  1, -1,  3, -1,  5],
       [-1,  7, -1,  9, -1, 11],
       [-1, 13, -1, 15, -1, 17],
       [-1, 19, -1, 21, -1, 23],
       [-1, 25, -1, 27, -1, 29]])
```

配列のコピーは通常はポインターのコピーなので、コピー先の配列を変更すると、元の配列も変更されてしまいます。

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = a
>>> b[0,1] = 6   # コピー先を変更
>>> b
array([[0, 6, 2],
       [3, 4, 5]])
>>> a            # コピー元の配列も変更されている
array([[0, 6, 2],
       [3, 4, 5]])
```

ポインターのコピーではなく、実体をコピーするには copy を利用します。

```
>>> a = np.arange(6).reshape(2,3)
```

```
>>> b1 = a.copy()     # 実体をコピー
>>> b2 = np.copy(a)   # これも実体のコピー
>>> b1[0,1] = 6       # コピー先を変更
>>> b2[0,1] = 6       # コピー先を変更
>>> a                 # コピー元の配列は変更されていない
array([[0, 1, 2],
       [3, 4, 5]])
```

1.3 配列に対する演算

まず押さえておかなければならないことは、「配列に対して数値に関する演算を適用させると、配列内の全ての数値にその演算が適用される」ということです。これはベクトル演算の基本です。ただし、その演算は通常のパッケージ math で定義されているものではなく、NumPy で定義されている演算でなくてはいけません。

```
>>> a = np.arange(1,7).reshape(2,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a + 1       # 四則演算はそのままできる
array([[2, 3, 4],
       [5, 6, 7]])
>>> a**2        # 2乗もそのままできる
array([[ 1,  4,  9],
       [16, 25, 36]])
>>> np.log(a)   # logはmath.logではなく、np.log
array([[ 0.        ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947]])
```

数値の集合（ベクトルと見なせる）に対する演算は、配列の全要素に対するものになります。配列の形がどうであれ、いったん 1 次元の配列（ベクトル）になると考えておけばよいでしょう。

```
>>> np.sum(a)   # 全要素の和
21
>>> np.mean(a)  # 全要素の平均
3.5
```

軸を固定して演算することもできます。2 次元の場合、「軸を固定する」とは、行あるいは列ごとに演算することに対応します。axis=0 を付けると列ごとに、axis=1 を付けると行ごとに演算します。

```
>>> np.sum(a, axis=0)   # 列ごとの和
array([5, 7, 9])
>>> np.sum(a, axis=1)   # 行ごとの和
array([ 6, 15])
```

数値やその集合に対する演算もたくさんあるので、必要に応じて調べてください。

続いて行列に対する演算です。まず、サイズが同じ行列の四則演算は要素ごとに行われます。

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = np.arange(6,12).reshape(2,3)
>>> b
array([[ 6,  7,  8],
       [ 9, 10, 11]])
>>> a + b    # 足し算
array([[ 6,  8, 10],
       [12, 14, 16]])
>>> a - b    # 引き算
array([[-6, -6, -6],
       [-6, -6, -6]])
>>> a * b    # かけ算
array([[ 0,  7, 16],
```

```
       [27, 40, 55]])
>>> a / b    # 割り算（整数どうしの割り算に注意）
array([[0, 0, 0],
       [0, 0, 0]])
```

次に行列の演算で重要なのは、行列の積です。ベクトルに対しては内積になります。

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(4,8)
>>> b
array([4, 5, 6, 7])
>>> a.dot(b)   # 配列が1次元（ベクトル）のときは内積
38
>>> a = np.arange(6).reshape(2,3)
>>> a          # 2行3列の行列
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = np.arange(6).reshape(3,2)
>>> b          # 3行2列の行列
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.dot(b)   # 行列のかけ算
array([[10, 13],
       [28, 40]])
```

また、行列の演算に関しては逆行列、転置行列、行列式、固有値が重要です。

```
>>> a = np.array([[0,6,3],[-2,7,2],[0,0,3]])
>>> a
array([[ 0,  6,  3],
       [-2,  7,  2],
       [ 0,  0,  3]])
```

```
>>> a.T                       # 転置行列
array([[ 0, -2,  0],
       [ 6,  7,  0],
       [ 3,  2,  3]])
>>> np.linalg.det(a)          # 行列式
36.0                          # 0 でないので逆行列がある
>>> np.linalg.inv(a)          # 逆行列
array([[ 0.58333333, -0.5       , -0.25      ],
       [ 0.16666667,  0.        , -0.16666667],
       [ 0.        ,  0.        ,  0.33333333]])
>>> la, v = np.linalg.eig(a)  # 固有値と固有ベクトル
>>> la
array([ 3.,  4.,  3.])        # 固有値
>>> v                         # 固有ベクトル
array([[-0.89442719, -0.83205029,  0.43643578],
       [-0.4472136 , -0.5547002 , -0.21821789],
       [ 0.        ,  0.        ,  0.87287156]])
```

1.4　配列の保存と読み出し

多大な計算コストをかけて作った配列も、プログラムが終了すればメモリから消えます。もう一度作るのに、再度多大な計算コストをかけるのは効率がよくありません。このような場合、配列のイメージをファイルに保存しておき、別プログラムでそのファイルから配列のイメージを読み出すようにします。

pickle を使えば、配列に限らずどのようなオブジェクトでも保存とその読み出しができます。

```
>>> a = np.random.randn(10000).reshape(100,100)
>>> a
array([[  7.44399780e-01,   2.33218034e+00,   ...,
          1.34792915e+00,  -7.81493561e-01],
       [  4.45931378e-01,   1.53038724e+00,   ...,
          7.05434461e-01,  -1.82279009e+00],
       [ -1.24952092e+00,   2.01394421e-01,   ...,
          2.08654508e+00,  -1.67717136e+00],
```

```
       ...,
       [  1.69618572e+00,   4.39548063e-02,   ...,
         -2.71672527e-02,  -8.68962312e-02],
       [  8.18006708e-01,   1.93145788e+00,   ...,
          3.21227532e-04,   5.14777810e-01],
       [  1.02043204e-02,  -4.97362556e-01,   ...,
         -9.42025754e-01,   4.16687820e-01]])
>>> import pickle
>>> f = open('a.pickle','w')
>>> pickle.dump(a,f)
>>> f.close()
```

上記で配列 a のイメージがファイル a.pickle に書き出されます。一度 python を終了し、再度立ち上げて、先のファイルから配列 a を読み出してみます。

```
>>> import numpy as np
>>> import pickle
>>> f = open('a.pickle','r')
>>> a = pickle.load(f)
>>> f.close()
>>> a
array([[  7.44399780e-01,   2.33218034e+00,   ...,
          1.34792915e+00,  -7.81493561e-01],
       [  4.45931378e-01,   1.53038724e+00,   ...,
          7.05434461e-01,  -1.82279009e+00],
       [ -1.24952092e+00,   2.01394421e-01,   ...,
          2.08654508e+00,  -1.67717136e+00],
       ...,
       [  1.69618572e+00,   4.39548063e-02,   ...,
         -2.71672527e-02,  -8.68962312e-02],
       [  8.18006708e-01,   1.93145788e+00,   ...,
          3.21227532e-04,   5.14777810e-01],
       [  1.02043204e-02,  -4.97362556e-01,   ...,
         -9.42025754e-01,   4.16687820e-01]])
```

pickle は汎用的に使えますが、NumPy の配列用には save と load、あるいは savetxt と loadtxt があります。

```
>>> np.save('a.npy',a)          # バイナリで保存
>>> b = np.load('a.npy')        # その読み出し
```

```
>>> np.savetxt('a.data',a)      # テキストで保存
>>> b = np.loadtxt('a.data')    # その読み出し
```

ファイル a.data 内には、行列の各行がスペース区切りで記載されています。つまり、このようなファイルに書かれているデータを読み込んで、プログラム内で配列を作りたい場合にも loadtxt が使えます。

また、Chainer では学習できたモデルの保存と読み込みのために、serializers が提供されています。基本的に保存は以下の形です。

```
serializers.save_npz(filename, model)
```

読み込みは以下の形です。

```
serializers.load_npz(filname, model)
```