アプリケーション開発　Pytorch 基礎 1　　担当　村松

・ Pytorch のインストール

pytorch などのパッケージのインストール手法について示す

windows の場合、python3 については　https://www.python.org/downloads/　公式ページよりインストーラをダウンロードし、インストールを行う(2024 年 10 月現在の最新版は 3.13.0)

既にインストール済みである場合は上記は必要ない（また MacOS もインストール済みなので不要）

Python3 の IDLE シェルを立ち上げ下記のように sys モジュールをインポートし Path を確認する

```
>>>import sys
>>>sys.path
['', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312¥¥Lib¥¥idlelib', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312¥¥python312.zip', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312¥¥DLLs', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312¥¥Lib', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312', 'C:¥¥Users¥¥owner¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python312¥¥Lib¥¥site-packages']
>>>
```

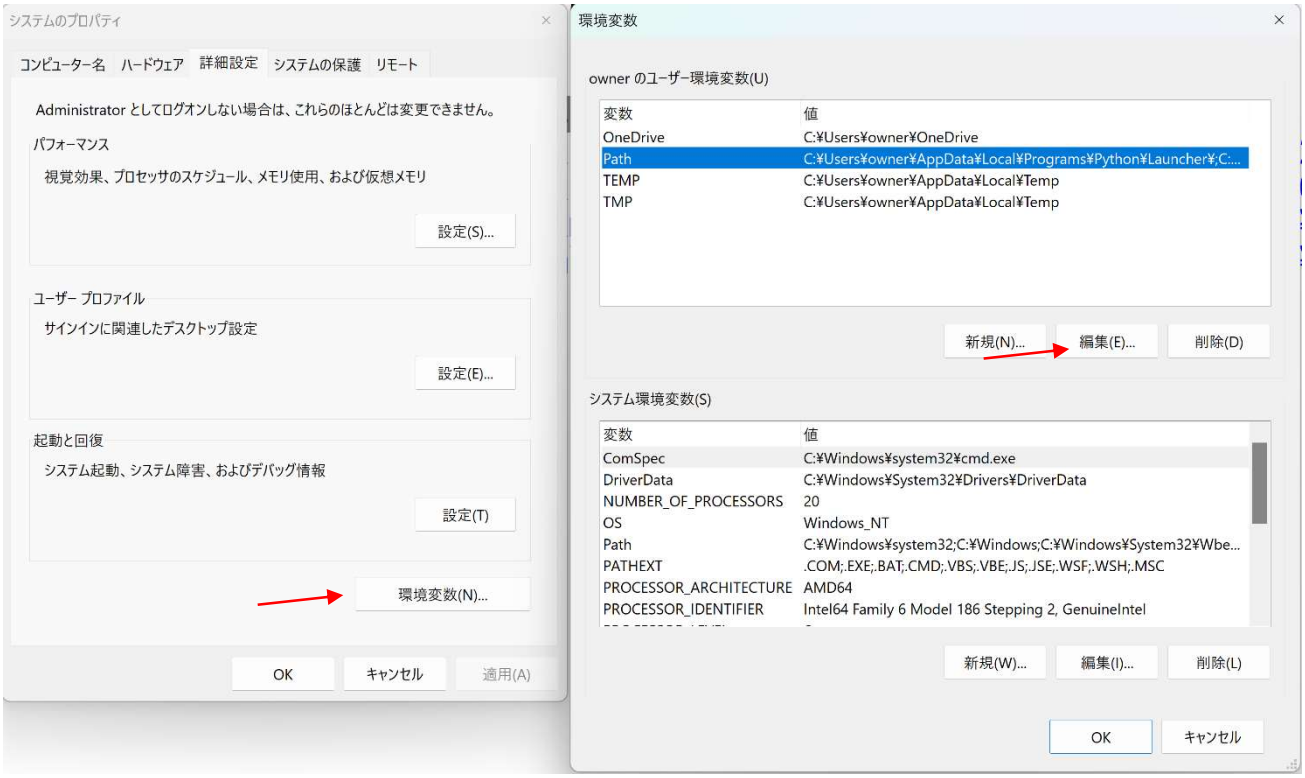
ここで出力結果より最初の行の

C:¥¥Users¥¥ユーザ名¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python バージョン¥¥Lib¥¥

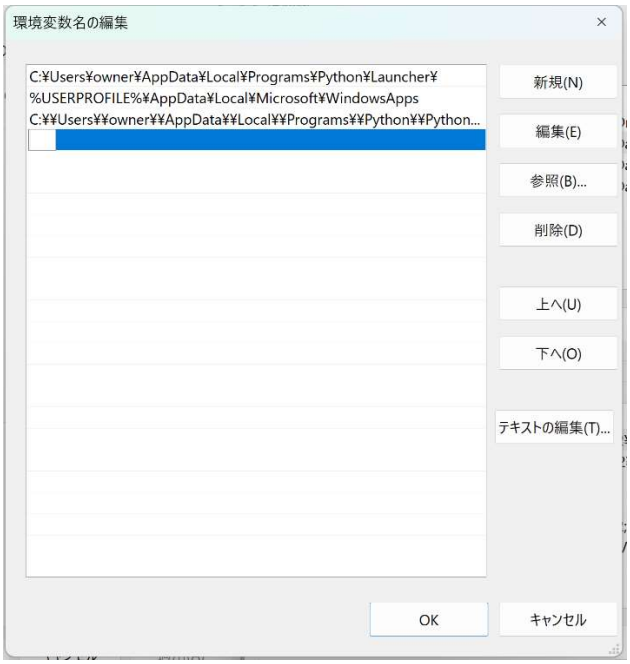
をコピーする（ユーザ名、Python バージョンは各自で異なるので注意）

システムの詳細設定より環境変数の項目を開く

環境変数のプロパティを開いたら Path を選択して「編集」を開く



環境変数名の編集にて「新規」を選択し、先ほどコピーしたパスを張り付ける
張り付けた後、下記のように¥¥の最後に「Scripts」を追記する
C:¥¥Users¥¥ユーザ名¥¥AppData¥¥Local¥¥Programs¥¥Python¥¥Python バージョン¥¥Lib¥¥Scripts
（ユーザ名、Python バージョンは各自で異なるので注意）
追加できたらＯＫで閉じる



以上は MacOS の場合は不要

次に Windows の場合は「windows PowerShell」を、MacOS の場合は「ターミナル」を開く
以下のコマンドを入力、実行してパッケージをインストールする（以下は windows での例）
pip3 install numpy scipy matplotlib pandas torch torchvision torchaudio

以上で必要な環境のインストールが完了

・tensor の生成

Pytorch では入力データは tensor と呼ばれる行列のような形で扱われる

torch.Tensor というクラスが用意されており、torch.tensor()を呼び出して値を定義することで生成する

pytorch のモジュールを組み込む場合は以下のように torch を import する

```
>>>import torch
```

数値がひとつの場合は「スカラー」もしくは「0 階テンソル」と呼ばれる

使い方：tensor(スカラーもしくはベクトル、行列)

```
>>>torch_tensor1 = torch.tensor(1)
>>>torch_tensor1
   tensor(1)
>>>type(torch_tensor1)
   <class 'torch.Tensor'>
```

数値が 1 次元配列の場合は「ベクトル」もしくは「1 階テンソル」と呼ばれる

```
>>>torch_tensor2 = torch.tensor([1,2,3,4])
>>>torch_tensor2
   tensor([1, 2, 3, 4])
>>>type(torch_tensor2)
   <class 'torch.Tensor'>
```

数値が 2 次元配列以上の場合は「行列」もしくは「2 階テンソル」と呼ばれ、3 次元の場合は「3 階テンソル」、4 次元の場合は「4 階テンソル」という具合に n 次元配列に対して「n 階テンソル」となる

```
>>>torch_tensor3 = torch.tensor([[1,2,3],[4,5,6]])
>>>torch_tensor3
   tensor([[1, 2, 3],
           [4, 5, 6]])
>>>type(torch_tensor3)
   <class 'torch.Tensor'>
```

Python では大きな行列を扱うためのモジュールとして numpy があり、下記のような形で import できる（ここでは as を用いて np という名前で import している）

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

Numpy の行列は以下のように array()を用いることで作成できる

使い方：array(ベクトルもしくは行列)

```
>>>np_array1 = np.array([1,2,3,4])
>>>np_array1
   array([1, 2, 3, 4])
>>>type(np_array1)
   <class 'numpy.ndarray'>
>>>np_array2 = np.array([[1,2,3],[4,5,6]])
>>>np_array2
   array([[1, 2, 3],
          [4, 5, 6]])
>>>type(np_array2)
   <class 'numpy.ndarray'>
```

このように numpy で大規模行列を作成した後に pytorch 用の tensor に変換する必要がある
Pytorch では from_numpy()を用いることで numpy の array を pytorch の tensor に変換できる
使い方：from_numpy(numpy のベクトルもしくは行列)

```
>>>np_array2 = np.array([[1,2,3],[4,5,6]])
>>>np_array2
   array([[1, 2, 3],
          [4, 5, 6]])
>>>type(np_array2)
   <class 'numpy.ndarray'>
>>>torch_tensor4 = torch.from_numpy(np_array2)
>>>torch_tensor4
   tensor([[1, 2, 3],
           [4, 5, 6]], dtype=torch.int32)
>>>type(torch_tensor4)
   <class 'torch.Tensor'>
```

Pytorch の numpy()を用いることで tensor を numpy の array に戻すこともできる
使い方：pytorch の tensor.numpy()

```
>>>np_array3 = torch_tensor4.numpy()
>>>np_array3
   array([[1, 2, 3],
          [4, 5, 6]])
>>>type(np_array3)
   <class 'numpy.ndarray'>
```

Pytorch には arange()があり、Python の renge()のような形で tensor を作成することができる
arrange()は()内の数値の範囲で数を生成し、テンソルに格納する
使い方：arange(生成したい数値の個数)もしくは arange(数値 1, 数値 2) *数値 1 以上数値 2 未満で生成

```
>>>torch_tensor5 = torch.arange(10)
>>>torch_tensor5
   tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>type(torch_tensor5)
   <class 'torch.Tensor'>

>>>torch_tensor6 = torch.arange(10,17)
>>>torch_tensor6
   tensor([10, 11, 12, 13, 14, 15, 16])
>>>type(torch_tensor6)
   <class 'torch.Tensor'>
```

linspace()を用いることである範囲の数を指定の数で等分する形で値を生成して tensor を作成することもできる
使い方：linspace(数値 1, 数値 2, 数値 3) *数値 1 以上数値 2 以下の範囲を数値 3 で等分して生成
例えば 0 から 1 までの範囲を 6 等分した数値を生成する場合は

```
>>>torch_tensor7 = torch.linspace(0,1,6)
>>>torch_tensor7
   tensor([0.0000, 0.2000, 0.4000, 0.6000, 0.8000, 1.0000])
```

ones()を用いることですべて 1 の要素の tensor を作成することができる

使い方：ones((行数, 列数))

```
>>>torch_tensor8 = torch.ones((3,3))
>>>torch_tensor8
   tensor([[1., 1., 1.],
           [1., 1., 1.],
           [1., 1., 1.]])
```

zeros()を用いることですべて 0 の要素の tensor を作成することができる

使い方：zeros((行数, 列数))

```
>>>torch_tensor9 = torch.zeros((3,3))
>>>torch_tensor9
   tensor([[0., 0., 0.],
           [0., 0., 0.],
           [0., 0., 0.]])
```

randn()を用いることで正規分布に従ったランダムな数値を生成してテンソルを生成できる

使い方：randn(要素数)もしくは randn(行数,列数)もしくは randn(次元数,行数,列数)

```
>>>torch_tensor10 = torch.randn(4)
>>>torch_tensor10
   tensor([ 0.1025,  1.7936,  0.0413, -0.9654])

>>>torch_tensor10 = torch.randn(2,3)
>>>torch_tensor10
   tensor([[-1.1211,  1.1862, -0.8247],
           [ 0.1244, -0.0269,  0.4385]])

>>>torch_tensor10 = torch.randn(2,3,4)
>>>torch_tensor10
   tensor([[[ 2.3911,  1.1161, -1.2445, -1.1102],
            [-0.6570,  1.3372, -0.3615,  0.3178],
            [ 0.3710, -1.1264, -0.5486,  0.2970]],

           [[-1.8103, -0.0196,  0.3495,  0.0326],
            [-0.0155, -2.2394, -1.1134,  1.6253],
            [ 1.7091, -0.4492, -1.4816,  0.2632]]])
```

rand()を用いることでのランダムな数値によるテンソルが生成できる

使い方：rand(要素数)もしくは rand(行数, 列数)もしくは rand(次元数, 行数, 列数)

```
>>>torch_tensor11 = torch.rand(5,3)
>>>torch_tensor11
   tensor([[0.4016, 0.6218, 0.6237],
           [0.3737, 0.3849, 0.0913],
           [0.8604, 0.7339, 0.1996],
           [0.5583, 0.1022, 0.2521],
           [0.8513, 0.4350, 0.1334]])
```

・tensor の操作

view()を用いることで指定した tensor を任意の形状に変更できる

使い方：pytorch の tensor.view(行数, 列数)　＊行数もしくは列数のどちらかに-1 を指定すると自動調整して行数もしくは列数を相手の数に合わせて自動調整する

```
>>>torch_tensor12 = torch.arange(6)
>>>torch_tensor12
   tensor([0, 1, 2, 3, 4, 5])
>>>torch_tensor13 = torch_tensor12.view(3,2)
>>>torch_tensor13
   tensor([[0, 1],
           [2, 3],
           [4, 5]])
```

transpose()を用いることで指定した軸（行と列）を入れ替えることができる

使い方：transpose(転置したい tensor, 軸 1 , 軸 2) *軸の番号は index と同じく 0 から始まることに注意

```
>>>torch_tensor13
   tensor([[0, 1],
           [2, 3],
           [4, 5]])
>>>torch_tensor14 = torch.transpose(torch_tensor13, 0, 1)
>>>torch_tensor14
   tensor([[0, 2, 4],
           [1, 3, 5]])
```

単純な転置なら t()を用いることでも可能

```
>>>torch_tensor14 = torch_tensor13.t()
>>>torch_tensor14
   tensor([[0, 2, 4],
           [1, 3, 5]])
```

tensor はリストのように index 指定やスライスにより値を取り出すことができる

```
>>>torch_tensor15 = torch.arange(5)
>>>torch_tensor15
   tensor([0, 1, 2, 3, 4])
>>>torch_tensor15[1]
   tensor(1)
>>>torch_tensor15[1:3]
   tensor([1, 2])
>>>torch_tensor15[1] = 5
>>>torch_tensor15
   tensor([0, 5, 2, 3, 4])
>>>torch_tensor16 = torch.randn(2,3)
>>>torch_tensor16
   tensor([[-0.6767, -0.8791, -0.6625],
           [-1.6164, -1.0352,  0.2866]])
>>>torch_tensor16[0]
   tensor([-0.6767, -0.8791, -0.6625])
>>>torch_tensor16[0][1]
   tensor(-0.8791)
```

・tensor の計算

要素同士の足し算

add()を用いると tensor の要素同士の足し算ができる

使い方：add(テンソル 1, スカラー)もしくは add(テンソル 1, テンソル 2)

```
>>>torch_tensor17 = torch.arange(4)
>>>torch_tensor17
   tensor([0, 1, 2, 3])
>>>torch.add(torch_tensor17, 3)
   tensor([3, 4, 5, 6])
>>>torch_tensor18 = torch.arange(4)
>>>torch_tensor18
   tensor([0, 1, 2, 3])
>>>torch.add(torch_tensor17, torch_tensor18)
   tensor([0, 2, 4, 6])
```

要素同士の掛け算

mul()を用いると tensor の要素同士の積を計算することができる

使い方：mul(テンソル, スカラー)もしくは mul(テンソル１, テンソル２)

```
>>>torch_tensor17
   tensor([0, 1, 2, 3])
>>>torch.mul(torch_tensor17, 3)
   tensor([0, 3, 6, 9])

>>>torch_tensor17
   tensor([0, 1, 2, 3])
>>>torch_tensor18
   tensor([0, 1, 2, 3])
>>>torch.mul(torch_tensor17, torch_tensor18)
   tensor([0, 1, 4, 9])
```

内積

dot()を用いるとテンソル同士の内積を計算することができる

使い方：dot(テンソル 1, テンソル 2)

```
>>>torch_tensor19 = torch.tensor([1,2,3,4])
>>>torch_tensor19
   tensor([1, 2, 3, 4])
>>>torch_tensor20 = torch.tensor([5,6,7,8])
>>>torch_tensor20
   tensor([5, 6, 7, 8])

>>>torch.dot(torch_tensor19, torch_tensor20)
   tensor(70)
```

行列の積

matmul()を用いると行列の積を計算できる

使い方：mv(テンソル 1, テンソル 2)

```
>>>torch_tensor21 = torch.arange(8).view(2,4)
>>>torch_tensor21
   tensor([[0, 1, 2, 3],
           [4, 5, 6, 7]])
>>>torch_tensor22 = torch_tensor21.t()
>>>torch_tensor22
   tensor([[0, 4],
           [1, 5],
           [2, 6],
           [3, 7]])
>>>torch.matmul(torch_tensor21, torch_tensor22)
   tensor([[ 14,  38],
           [ 38, 126]])
```

・テンソルの形や要素の確認

足し算や積を計算する際は行列の要素数や形に注意する必要がある

要素数を確認するためには shape を用いることでその要素数や行列の形を参照できる

使い方：テンソル.shape

```
>>>torch_tensor2
   tensor([1, 2, 3, 4])
>>>torch_tensor2.shape
   torch.Size([4])
>>>torch_tensor3
   tensor([[1, 2, 3],
           [4, 5, 6]])
>>>torch_tensor3.shape
   torch.Size([2, 3])
```

max()を用いることでそのテンソル内の最大値を参照することができる

使い方：テンソル.max()

```
>>>torch_tensor3
   tensor([[1, 2, 3],
           [4, 5, 6]])
>>>torch_tensor3.max()
   tensor(6)
```

また、以下のような使い方も可能でこの場合、指定した軸の最大値とその index を参照することができる

使い方：max(テンソル, 軸)　＊軸＝ 0 の場合は列方向で最大値を参照、軸= 1 の場合は行方向で参照

```
>>>torch.max(torch_tensor3,0)
   torch.return_types.max(
   values=tensor([4, 5, 6]),
   indices=tensor([1, 1, 1]))
```

上記の場合は torch_tensor3 を列方向に参照し、各列毎の最大値とその index 番号が結果として得られる

一列目の最大値は 4 でその index（この場合は行数）は 1 となっているので 1 列目の最大値 4 の index は（0 列目の 1 行目）という形になる（index は 0 番からスタート）

```
>>>torch.max(torch_tensor3,1)
   torch.return_types.max(
   values=tensor([3, 6]),
   indices=tensor([2, 2]))
```

上記の場合は軸=1 としているので、行方向に最大値を参照し、1 行目の最大値が 3 で 2 行目の最大値が 6 という結果が返ってきており、その index がそれぞれ 2（この場合列の index）となっている
つまり 1 行目の最大値 3 の index は（2 列目、0 行目）ということになる（index は 0 番からスタート）
この他、以下のように末尾に index 参照 [1] を付けると各行の最大値が存在する列の index のみ抽出できる（予測結果のうち最も可能性が高いものを参照する場合によくこの書き方を利用する）

```
>>>torch.max(torch_tensor3,1)[1]
   tensor([2, 2])
```

スカラーの型変換取り出し
スカラーのテンソルの場合、item()を用いると int など標準なスカラーとして取り出すことができる
使い方：スカラーのテンソル.item()

```
>>>torch_tensor1
   tensor(1)
>>>a = torch_tensor1.item()
>>>a
   1
>>>type(a)
   <class 'int'>
```

tensor にはその要素において型を定めることができる
以下の表の型が存在する

データ型	dtype	CPU テンソル	GPU テンソル
8 ビット整数	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
16 ビット浮動小数点	torch.float16 torch.half	torch.HalfTensor	torch.cuda.HalfTensor
32 ビット浮動小数点	torch.float32 torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64 ビット浮動小数点	torch.float32 torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
8 ビット整数 符号付き	torch.uint16 torch.short	torch.ShortTensor	torch.cuda,ShortTensor
16 ビット整数 符号付き	torch.uint16 torch.short	torch.ShortTensor	torch.cuda,ShortTensor
32 ビット整数 符号付き	torch.float32 torch.int	torch.intTensor	torch.cuda.intTensor
64 ビット整数 符号付き	torch.uint64 torch.long	torch.LongTensor	torch.cuda.LongTensor

型の確認

dtype を用いることでテンソルや numpy の行列の型を参照することができる

使い方：テンソル.dtype

```
>>>np_array = np.array([1,2,3])
>>>np_array
   array([1, 2, 3])
>>>np_array.dtype
   dtype('int32')
>>>np_array2 = np.array([1.0, 2.0, 3.0])
>>>np_array2
   array([1., 2., 3.])
>>>np_array2.dtype
   dtype('float64')

>>>torch_tensor2
   tensor([1, 2, 3, 4])
>>>torch_tensor2.dtype
   torch.int64
```

型の指定

tensor もしくは numpy の array を生成する際に dtype に型を明示することで指定の型に変更できる

```
>>>np_array = np.array([1,2,3])
>>>np_array.dtype
   dtype('int32')
>>>np_array = np.array([1,2,3],dtype=float)
>>>np_array
   array([1., 2., 3.])
>>>np_array.dtype
   dtype('float64')
>>>torch_tensor = torch.tensor([1,2,3])
>>>torch_tensor.dtype
   torch.int64
>>>torch_tensor = torch.tensor([1,2,3],dtype=float)
>>>torch_tensor
   tensor([1., 2., 3.], dtype=torch.float64)
>>>torch_tensor.dtype
   torch.float64
```

・補足　GPU へのテンソルの配置

機械学習は大量のデータ（テンソル）を処理する必要があるため、そのデータ量に比例して学習時間が増大する問題がある

GPU（グラフィックカード）はテンソルを高速並列演算することが可能であるため CPU の場合に比較して数百倍から数千倍（GPU クラスタ数によっては数万倍以上）の速度で学習することが可能であるため予算や電力などのリソースが許すならば積極的に活用するべきである

GPU は Nvidia および AMD の両社からチップセットが販売されているが、現状では Nvidia の Geforce シリーズがデファクトスタンダードになっており、Pytorch においても Nvidia の GPU ライブラリである「CUDA」を用いて処理されることが非常に多い

データ型としても GPU は cuda を前提として型が定義されている

GPU 上の VRAM にテンソルを配置する場合は、tensor の定義時に

「device = “cuda:id”」もしくは 「to(“cuda:id”)」 ＊id はデバイス番号

として CUDA を介して GPU に配置することを明示する

```
>>>torch_tensor = torch.tensor([[1,2],[3,4]],device = "cuda:0")
```

もしくは

```
>>>torch_tensor = torch.tensor([[1,2],[3,4]]).to("cuda:0")
```

Nvidia 製の GPU が計算機に搭載されておらず CUDA ドライバが有効化されていない場合は下記のようにエラーメッセージが表示される

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    torch_tensor = torch.tensor([[1,2],[3,4]],device = "cuda:0")
  File "C:¥Users¥owner¥AppData¥Local¥Programs¥Python¥Python311¥Lib¥site-packages
¥torch¥cuda¥__init__.py", line 310, in _lazy_init
    raise AssertionError("Torch not compiled with CUDA enabled")
AssertionError: Torch not compiled with CUDA enabled
```

自身の開発環境で CUDA が有効化されているかどうかは「cuda.is_available()」で確認することができる

有効化されていれば「True」が返ってくる

以下は有効化されていないので「False」が返っている

```
>>>torch.cuda.is_available()
   False
```

GPU の VRAM に配置されている tensor を numpy の array に変換する場合、GPU 上の tensor を直接操作することができない

この場合、「to(“cpu”)」を用いることで GPU からＣＰＵのメモリに移動して変換する

以下は GPU 上の torch_tensor という tensor 変数を np_array という numpy array 変数へ変換する例

```
>>>np_array = torch_tensor.to("cpu").numpy()
```