PyTorch講座　第三回

杉本　晃輔

前回のおさらい

- Pythonを使ってクラスを実装
  - 意味のある数値の集合を作ることが可能
  - 集合に対して一括で操作が可能

- PythonのライブラリであるNumpyの基礎を学習
  - 多次元配列を扱うことが可能
  - 多次元配列に対して一括・高速に処理を施すことが可能

Deep Learningの基礎

Deep Learningとは？

- 深層学習と呼ばれる、機械学習手法の一種
- Neural Networkと呼ばれる、人の脳神経細胞の仕組みを模したモノを利用

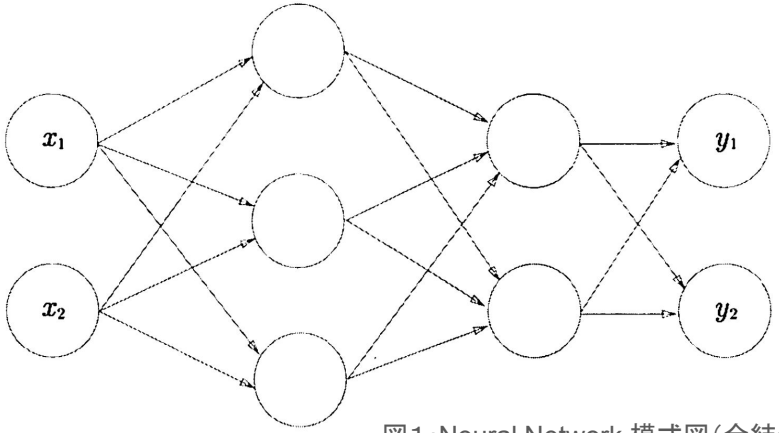

図1 :Neural Network 模式図（全結合）

NNの仕組み

- 学習アルゴリズムとして「Backpropagation（誤差逆伝播法）」を利用
- 誤差を伝播するにあたり、「勾配降下法」と呼ばれるアルゴリズムも利用

お気持ち的には、
誤差を重みで偏微分して得られる変化分だけ
重みを更新していくアルゴリズム

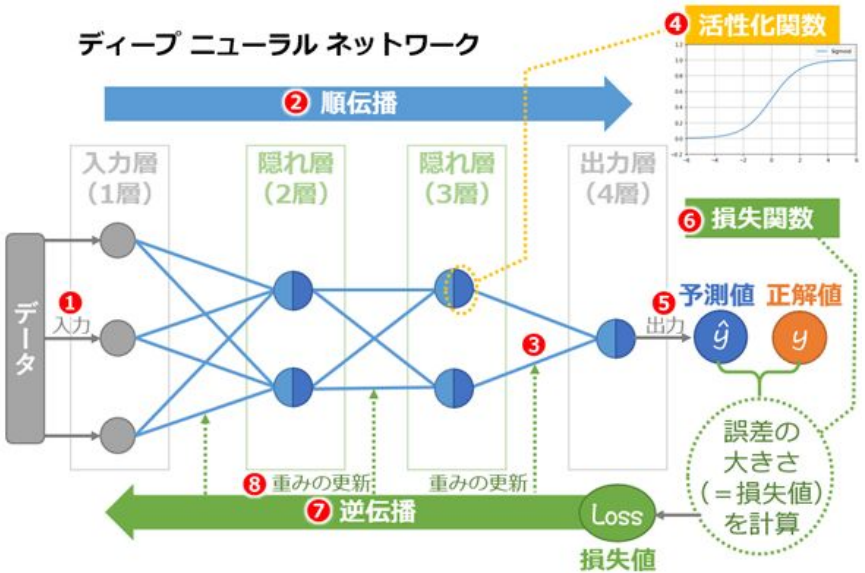

図2：誤差逆伝播法 模式図

実装方法

- Deep Learningを行う上で必要な機能が詰め込まれたフレームワークがPythonのライブラリとしていくつか提供されている
  - PyTorch：FacebookのAI Research labによって2016年に開発
  - TensorFlow：Googleによって2015年に開発
  - Keras：Googleの1エンジニアによって2015年に開発、現在はTensorFlowがサポート
    （お気持ち的にはTensorFlowを楽に使うためのライブラリ）

（少し偏見が入っているかもしれない）
フレームワーク比較表

\ Framework	PyTorch	TensorFlow	Keras
実行速度	速い	速い	遅い
カスタマイズ性	高い	高い	低い
可読性	低い	低い	高い
互換性	高い	低い	低い
このフレームワークでしかできないこと	ー	int8への量子化	TensorFlowと同様

（少し偏見が入っているかもしれない）
フレームワーク比較表

\ Framework	PyTorch	TensorFlow	Keras
実行速度	速い	速い	遅い
カスタマイズ性	高い	高い	皆無
可読性	低い	低い	高い
互換性	高い	低い	低い
このフレームワークでしかできないこと	ー	int8への量子化	TensorFlowと同様

⇒ 今回はバランスが良く、互換性の高いPyTorchを学習する

PyTorchの基礎

構成要素

- Deep Learningを実行するために必要な構成要素は大きく分けて、
    - モデル
    - データセット・データローダー
    - トレーニング

  の 3 つ

PyTorch - モデル -

- 情報を入力し、特定の形式で出力するまでの一連の処理を担当
  - 入力：画像、音声、自然言語 etc…
  - 出力：タスクによって異なる
    - 画像分類・認識 ： 確率値
    - 物体検出：Bounding Box
    - 画像合成・音声合成：画像、音声（合成対象）
- Neural Networkに相当

```python
from torch import nn


class MyModel(nn.Module):

    def __init__(self):
        super().__init__()

    def forward(self, x):
        return x

```

PyTorch - データセット・データローダー -

- 読み込み・前処理など、学習に用いるデータ関連全ての処理を担当
- データセット：データを読み込み、前処理を施して返すまでの流れを記述
- データローダー：データセットからデータを1つ受け取り、バッチ化する

```python
from PIL import Image
from pathlib import Path
from torch.utils.data import Dataset, DataLoader


class MyDataset(Dataset):

    def __init__(self, dataset_dir):
        self.dataset_dir = Path(dataset_dir).resolve()
        self.paths = list(self.dataset_dir.glob("*"))

    def __len__(self):
        return len(self.paths)

    def __getitem__(self, idx):
        path = self.paths[idx]
        return path

dataloader = DataLoader(dataset=MyDataset("."), batch_size=2)

```

PyTorch - トレーニング -

- 前述したモデル・データローダーを使って、モデルを学習させる処理を記述
  - データローダーからデータを受け取る
  - 受け取ったデータをモデルに入力
  - モデルからの出力を受け取る
  - 出力を使って誤差を算出
  - 誤差を使って重みを更新

Tensor型について

PyTorchのモデルへ入力する際には、torch.Tensor型である必要がある

- Numpyと同じく多次元配列を扱うためのデータ型
- モデルの学習に必要な「勾配」を保持できる
- GPUを利用した高速な計算をサポート

基本的にNumpyで可能な操作はTensorでも可能

Tensorに対する操作

- Tensor作成
  - torch.zeros = np.zeros
  - torch.ones = np.ones
  - torch.full　　= np.full
- NumpyからTensor作成
  - メモリ共有 : torch.from_numpy(numpy配列)
  - メモリ非共有 : torch.tensor(numpy配列)
- 形状を確認
  - torch.Tensor.size = np.ndarray.shape
- 軸入れ替え
  - torch.permute

tensor_script.py

workspace > pytorch-training > codes > 03 > example > tensor_script.py

```python
import torch
import numpy as np

# 0 埋めの Tensor 作成
# 形状指定はリスト、タプル、数値、のどれでも問題なし
zero_tensor1 = torch.zeros((2, 4))
zero_tensor2 = torch.zeros([2, 4])
zero_tensor3 = torch.zeros(2, 4)
print(zero_tensor1, "\n", zero_tensor2, "\n", zero_tensor3)

# 任意の値で埋めた Tensor 作成
# 形状指定はリスト、タプルのみ、＋埋めたい値
any_tensor1 = torch.full([2, 4], 3.14)
any_tensor2 = torch.full((2, 4), 3.14)
print(any_tensor1, "\n", any_tensor2)

# numpy からの Tensor に変換
arr = np.full((2, 4), 3.14)
shared_tensor = torch.from_numpy(arr)
private_tensor = torch.tensor(arr)
print("before")
print(shared_tensor, "\n", private_tensor)
arr[0] = -1
print("after")
print(shared_tensor, "\n", private_tensor)

# 形状確認 & 軸入れ替え
# permute( 対象のテンソル , 対象テンソルの並び替え後の軸(0-index) )
zero_tensor4 = torch.zeros((2, 4, 8))
print(zero_tensor4.size(), torch.permute(zero_tensor4, (2, 0, 1)).size())
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
 tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
 tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
tensor([[3.1400, 3.1400, 3.1400, 3.1400],
        [3.1400, 3.1400, 3.1400, 3.1400]])
 tensor([[3.1400, 3.1400, 3.1400, 3.1400],
        [3.1400, 3.1400, 3.1400, 3.1400]])
before
tensor([[3.1400, 3.1400, 3.1400, 3.1400],
        [3.1400, 3.1400, 3.1400, 3.1400]], dtype=torch.float64)
 tensor([[3.1400, 3.1400, 3.1400, 3.1400],
        [3.1400, 3.1400, 3.1400, 3.1400]], dtype=torch.float64)
after
tensor([[-1.0000, -1.0000, -1.0000, -1.0000],
        [ 3.1400,  3.1400,  3.1400,  3.1400]], dtype=torch.float64)
 tensor([[3.1400, 3.1400, 3.1400, 3.1400],
        [3.1400, 3.1400, 3.1400, 3.1400]], dtype=torch.float64)
torch.Size([2, 4, 8]) torch.Size([8, 2, 4])
```

演習1

- PyTorch講座2で用いた以下のデータ（3クラス、各クラス5人、2科目）

```
data = np.array([
[[85, 78], [67, 82], [92, 88], [75, 70], [60, 64]],
[[70, 68], [77, 72], [85, 90], [60, 65], [78, 76]],
[[80, 84], [88, 87], [66, 68], [72, 73], [64, 60]]])
```

Tips:

torch.mean を使うときは中身のデータが
float でないとエラーが…

tensor メソッドで引数指定か、
⇒ torch.tensor(~, dtype=float)
float() メソッドの利用
⇒ torch.Tensor.float()

問題

1. Numpy 配列を Tensor に変換
2. 2科目、3クラス、各クラス5人に並び変える
3. 2で作成したデータからクラスごと、個々人の2科目合計点
4. 3で作成したデータからクラスごと、2科目合計点の平均
5. 3で作成したデータからtorch.meanを使わずに4と同様のものを導出
   ※ ハードコーディングはしないこと

解答例

右図のようになればok

```
root@f65ec892051c:/work# python codes/03/exercise/exercise1.py
===== problem 1 =====
torch.Size([3, 5, 2])
===== problem 2 =====
tensor([[[85., 67., 92., 75., 60.],
         [70., 77., 85., 60., 78.],
         [80., 88., 66., 72., 64.]],

        [[78., 82., 88., 70., 64.],
         [68., 72., 90., 65., 76.],
         [84., 87., 68., 73., 60.]]], dtype=torch.float64)
torch.Size([2, 3, 5])
===== problem 3 =====
tensor([[163., 149., 180., 145., 124.],
        [138., 149., 175., 125., 154.],
        [164., 175., 134., 145., 124.]], dtype=torch.float64)
===== problem 4 =====
tensor([152.2000, 148.2000, 148.4000], dtype=torch.float64)
===== problem 5 =====
tensor([152.2000, 148.2000, 148.4000], dtype=torch.float64)
root@f65ec892051c:/work#
```

モデルの基礎

モデルの作成（基礎）

モデルは nn.Module を必ず継承

- __init__ :
  モデルの構成要素を記述
  super().__init__() は継承元の関数を引き継ぐため必須

- forward :
  入力から出力までの処理を記述
  引数にモデルの入力を与える

workspace > pytorch-training > env > sample.py

```python
from torch import nn


class MyModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv = nn.Conv2d(in_channels=3, out_channels=64,
                              kernel_size=3, stride=1, padding=1)
        self.bn = nn.BatchNorm2d(num_features=64)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        return x

```

モデルの作成（補足）

- init で定義したものはインスタンス作成時に初期値として引数で与えることが可能
- forward は 2 通りの呼び出し方法が存在

make_model.py

workspace > pytorch-training > codes > 03 > example > make_model.py

```python
import torch
from torch.nn import Module

class MyModel(Module):

    def __init__(self, arg1: int, arg2: str):
        super().__init__()
        self.arg1 = arg1
        self.arg2 = arg2

    def forward(self, x):
        return x

if __name__=="__main__":
    # インスタンス作成
    mymodel = MyModel(arg1=1234, arg2="tus")

    # forward 呼び出し
    _input = torch.ones((2, 3))
    out1 = mymodel(_input)
    out2 = mymodel.forward(_input)
    print("===== method1 ======")
    print(repr(out1))
    print("===== method2 ======")
    print(repr(out2))

```

```
root@3bd28f05084e:/work/codes/03/example# python make_model.py
===== method1 ======
tensor([[1., 1., 1.],
        [1., 1., 1.]])
===== method2 ======
tensor([[1., 1., 1.],
        [1., 1., 1.]])
root@3bd28f05084e:/work/codes/03/example#
```

演習2

問題

1. クラス変数として、mytensor, elem_add, elem_multiply を持ち、__init__ で引数からそれらのクラス変数を初期化する
2. モデルの呼び出しの際に、入力と self.mytensor を足し合わせる
3. 2 の操作後のテンソル全体に self.elem_add を加算する
4. 3 の操作後のテンソル全体に self.elem_multiply を乗算する
5. 2, 3, 4 操作後のテンソルを出力する

以上の条件をすべて満たすモデルのクラスを作成してください

解答例

右図のように指定して、左図のようになればok

```python
if __name__=="__main__":
    mymodel = ExerciseModel(torch.ones((3, 3)), 4, 6)

    p2out, p3out, p4out = mymodel(torch.full((3, 3), 2))

    print("===== problem 2 =====")
    print(repr(p2out))
    print("===== problem 3 =====")
    print(repr(p3out))
    print("===== problem 4 =====")
    print(repr(p4out))
```

```
root@3bd28f05084e:/work/codes/03/exercise# python exercise2.py
===== problem 2 =====
tensor([[3., 3., 3.],
        [3., 3., 3.],
        [3., 3., 3.]])
===== problem 3 =====
tensor([[7., 7., 7.],
        [7., 7., 7.],
        [7., 7., 7.]])
===== problem 4 =====
tensor([[42., 42., 42.],
        [42., 42., 42.],
        [42., 42., 42.]])
root@3bd28f05084e:/work/codes/03/exercise#
```