PyTorch 講座 4

杉本　晃輔

前回のおさらい

- DeepLearning について
  - 機械学習手法の一種
  - 誤差逆伝播、勾配降下法というアルゴリズムを使って学習
  - PyTorch, TensorFlow, Keras 等のフレームワークがある

- PyTorch について
  - モデル作成、データ準備、トレーニングの 3 つで構成される
  - モデルへの入力は torch.Tensor 型でないとダメ
    - Tensor 型はほとんど numpy と同じもの
  - モデルクラスは torch.nn.Module を継承させて定義
  - モデルクラスは __init__, forward を定義すれば ok
  - モデルクラスの __init__ では super().__init__() を呼び出すのが必須

Layer の基礎

レイヤーとは？

- モデルを成す1つ1つの構成部品のこと
  ⇒ 右図で言えば、Conv2D, BatchNormalization 等が該当

- レイヤーを多種かつ複数重ねることでモデルが構築される

- 本講座では、よく見かけることになるであろう、
  - Convolution
  - Linear
  - BatchNormalization
  - ReLU

  に絞って説明

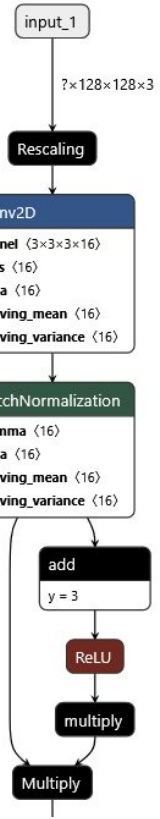


図1：MobileNetV3 より抜粋
netron にて描画

Convolution 層

- 畳み込み層とも呼ばれるレイヤー
- 近年のモデルの多くがこのレイヤーを使っている

- 画像の「特徴」を抽出する

カーネルと呼ばれる小さな多次元配列を入力全体に適用することで抽出

入力

カーネル

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

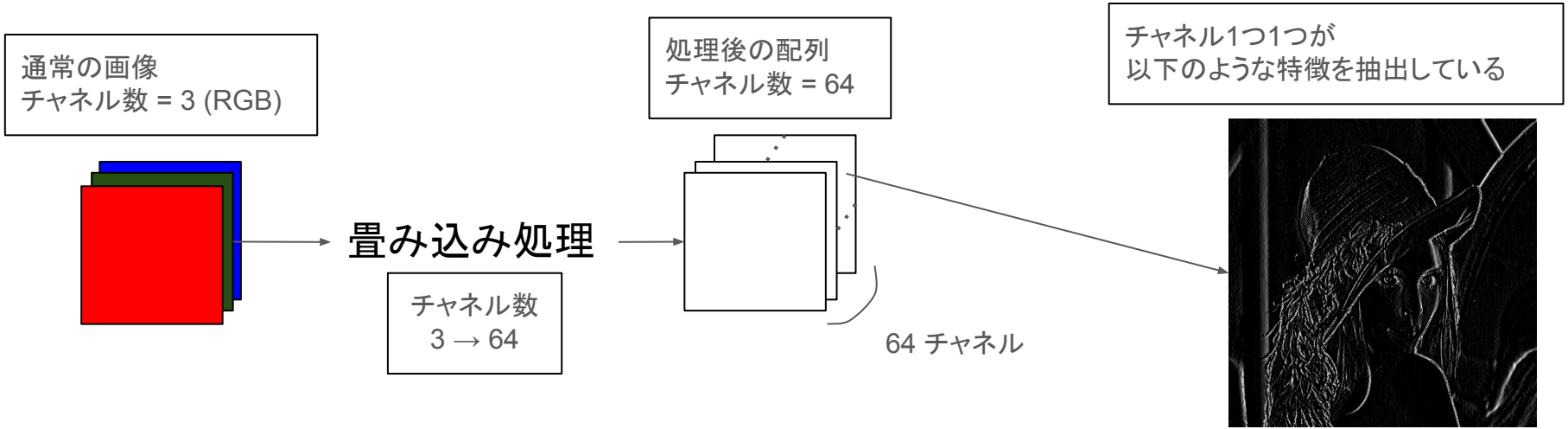
抽出された特徴

チャネルについて

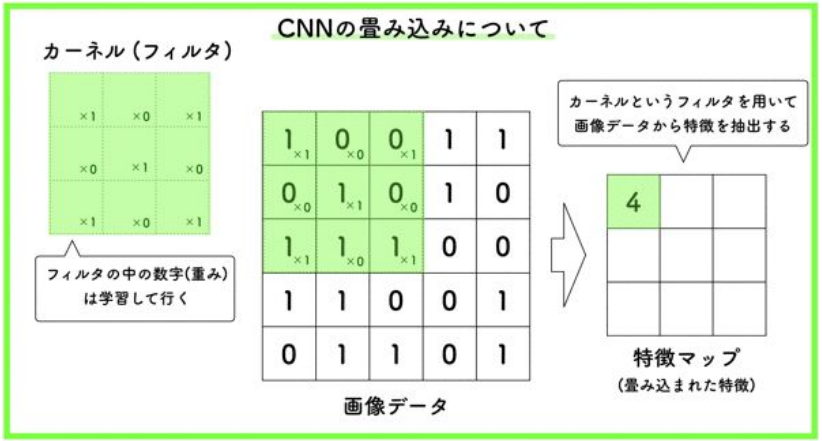
畳み込み層では「チャネル」と呼ばれる変数を指定する

⇒ チャネルとは、適用するカーネルの数

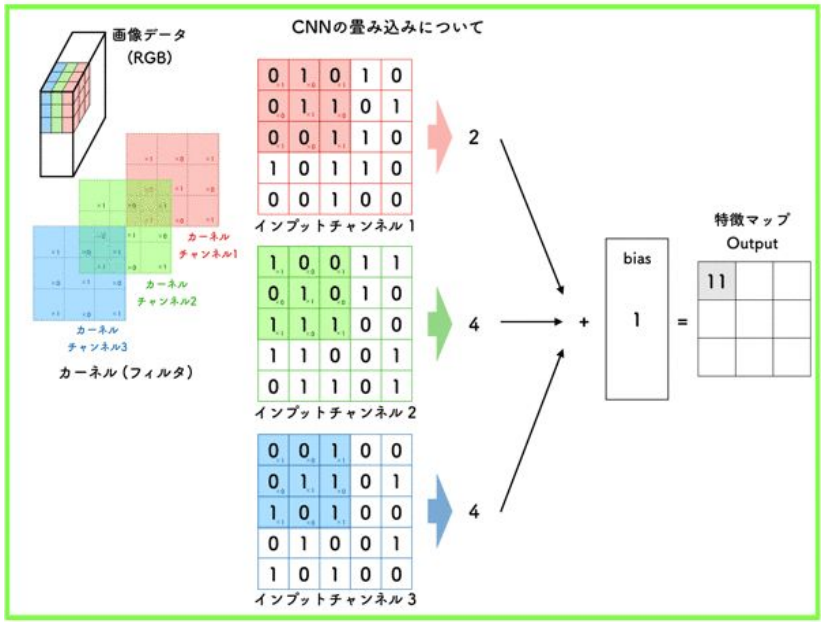
⇒ つまり、抽出する特徴の数を指している



実際の処理方法



チャネルが複数ある時は、
チャネル分のカーネルを用意しそれぞれ総和を取り、
1つの特徴として抽出する

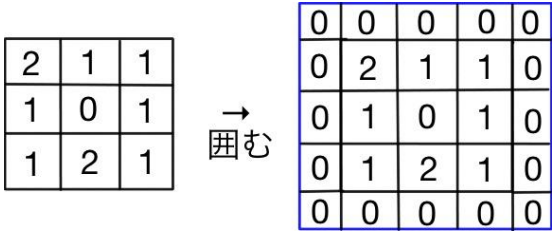


カーネルをズラす際の幅のことを stride という
上の例だと stride = 1

paddingをしないと得られる特徴マップは入力より縮小
⇒ 小さくなる分入力を padding すれば、
入力と同じ大きさの特徴が得られる

https://kenyu-life.com/2019/03/07/convolutional_neural_network/

PyTorch での実装



padding イメージ図

PyTorch では torch.nn.Conv2d で実装されている

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

主に使われる引数は

- in_channels (必須)　　：入力のチャネル数
- out_channels (必須)　　：出力される特徴のチャネル数
- kernel_size (必須)　　：処理に使うカーネルの大きさ
- stride　　：カーネルを適用させる際にズラす幅
- padding　　：入力サイズの補正幅

※ 入力データの端に付け足しを行う

PyTorch での実装（補足）

畳み込み処理を掛けた後の出力の大きさは

$$O = \frac{I + 2P - K}{S} + 1$$

と定義される（かなり簡素化してます）

K:カーネルサイズ(k,k)
P:パディング(p,p)
S:ストライド(s,s)
I:入力サイズ(i,i)
O:出力サイズ(o,o)

この式を元に出力が所望の大きさになるように前スライドの
引数からパラメーターを指定する

利用例

conv.py U

workspace > pytorch-training > codes > 04 > example > conv.py

```python
import torch
from torch import nn


if __name__=="__main__":

    # 画像代わりのテンソルを定義
    # 実際に学習に使われるデータは、
    #   (Batchsize, channel, width, height)
    # という形状であることが多い
    my_tensor = torch.full((16, 3, 256, 256), 2.718)

    # 畳み込み定義 & 適用
    # required のみ引数で指定
    conv = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3)
    out = conv(my_tensor)
    print(repr(out.size()))

    # optional も指定
    conv2 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=5, stride=2, padding=2)
    out2 = conv2(my_tensor)
    print(repr(out2.size()))

```

問題 出力 デバッグ コンソール ターミナル ポート bash - pytorch-training

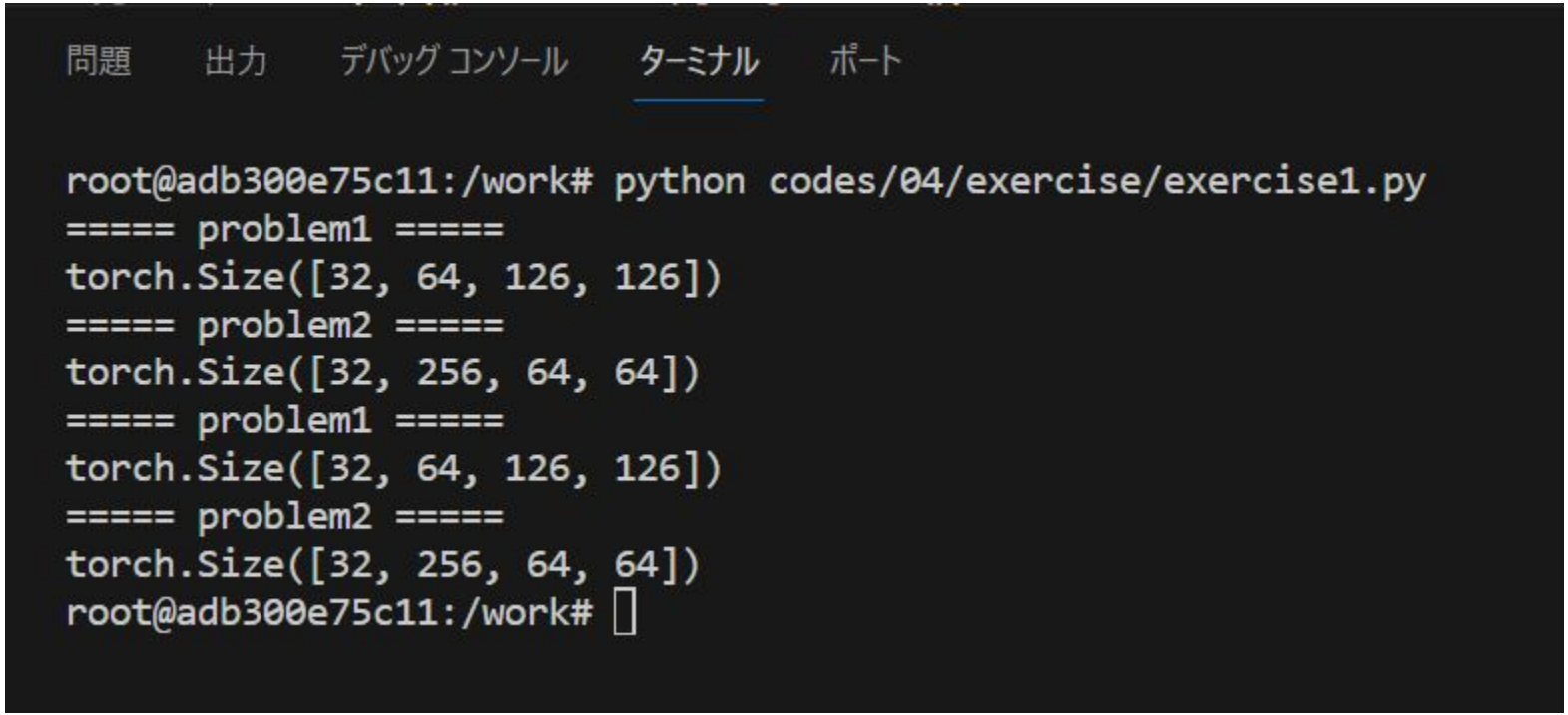
```
root@adb300e75c11:/work# python codes/04/example/conv.py
torch.Size([16, 64, 254, 254])
torch.Size([16, 128, 128, 128])
root@adb300e75c11:/work#
```

演習1

1. 32 × 3 × 128 × 128 のテンソルを作成してください
   ※ torch.ones を使って作成してみましょう（応用：別のメソッドでも）
2. 出力が 32 × 64 × 126 × 126 となるように畳み込みを定義し、
   1で作成したテンソルに適用してください (kernel_size = 3)
3. 出力が 32 × 256 × 64 × 64 となるように畳み込みを定義し、
   1で作成したテンソルに適用してください (kernel_size = 3)
4. 2, 3 で kernel_size = 5 として同様の結果が得られるように
   畳み込みを定義し、1 で作成したテンソルに適用してください

ヒント：
2. padding がいるかどうか考えてみましょう
3. ½ になっている ⇒ stride を調整すると…？

解答例

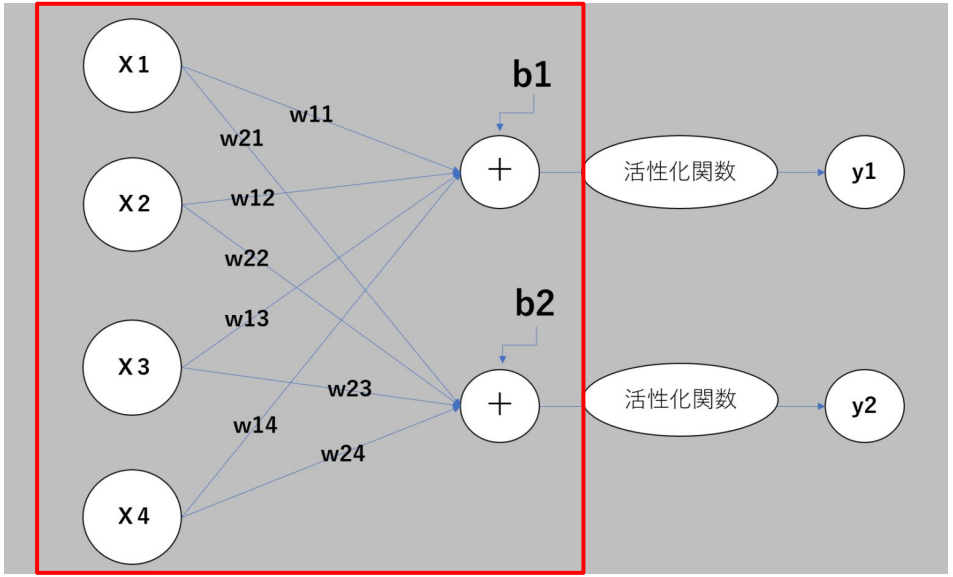

問題
出力
デバッグ コンソール
ターミナル
ポート
root@adb300e75c11:/work# python codes/04/exercise/exercise1.py
===== problem1 =====
torch.Size([32, 64, 126, 126])
===== problem2 =====
torch.Size([32, 256, 64, 64])
===== problem1 =====
torch.Size([32, 64, 126, 126])
===== problem2 =====
torch.Size([32, 256, 64, 64])
root@adb300e75c11:/work#

Linear 層

- 全結合層とも呼ばれるレイヤー
- 近年では出力層付近で使われることが多い

- 入力と重みのドット積を取り、
  新しい値を導出する

$$Linear(X) = W \cdot X(+bias)$$



Linear 層模式図

PyTorch での実装

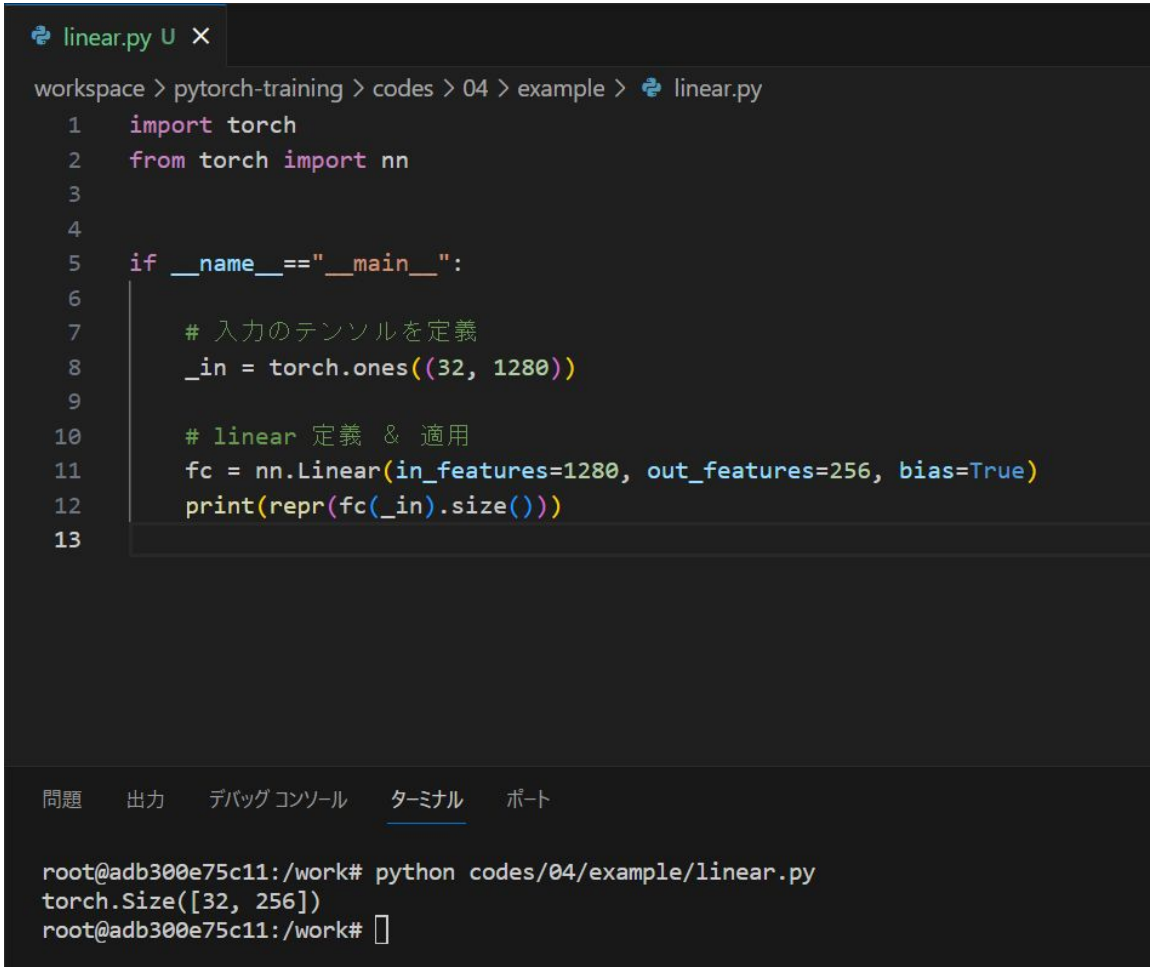
PyTorch では torch.nn.Linear で実装されている

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

主に使われる引数は
- in_features (必須)　　：入力の要素数（ 後述する要素数 N ）
- out_features (必須)　　：出力の要素数

畳み込みと異なり、期待する入力のサイズは
(B, N)
B = バッチサイズ, N = 要素数　の 1 次元データ

利用例


linear.py U
workspace > pytorch-training > codes > 04 > example > linear.py
1 import torch
2 from torch import nn
3
4
5 if __name__=="__main__":
6
7 # 入力のテンソルを定義
8 _in = torch.ones((32, 1280))
9
10 # linear 定義 & 適用
11 fc = nn.Linear(in_features=1280, out_features=256, bias=True)
12 print(repr(fc(_in).size()))
13
問題
出力
デバッグ コンソール
ターミナル
ポート
root@adb300e75c11:/work# python codes/04/example/linear.py
torch.Size([32, 256])
root@adb300e75c11:/work#

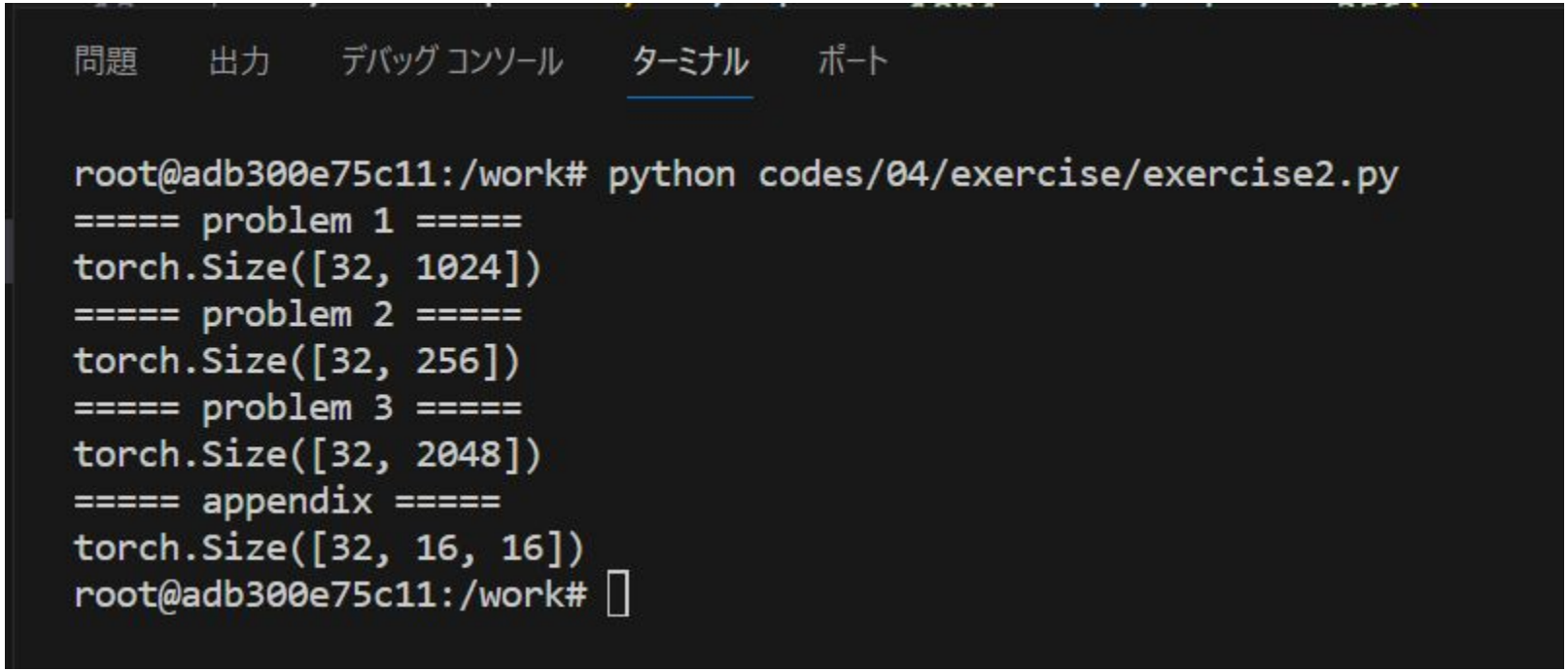
演習2

1. 入力用のテンソルとして、32 × 1024 のテンソルを定義してください
   ※ 埋める値は任意で問題ありません
2. 出力が 32 × 256 となるように全結合層を定義し、適用してください
3. 出力が 32 × 2048 となるように全結合層を定義し、適用してください

おまけ

2 で作成されたテンソルを 32 × 16 × 16 の形状のテンソルに直してください

解答例


問題
出力
デバッグ コンソール
ターミナル
ポート
root@adb300e75c11:/work# python codes/04/exercise/exercise2.py
===== problem 1 =====
torch.Size([32, 1024])
===== problem 2 =====
torch.Size([32, 256])
===== problem 3 =====
torch.Size([32, 2048])
===== appendix =====
torch.Size([32, 16, 16])
root@adb300e75c11:/work#

Batch Normalization 層

- 名称通り、正規化 (Normalization) を行うレイヤー
- 一般に ReLU の前に配置されることが多い

内部共変量シフト

- 層を重ねることで生じる可能性のある、
  「各層の出力の分布が大きく変化する」
  という現象を抑制する働きがある
  - 他にも、様々な利点が
    参考 : Qiita1, Qiita2

⇒ 抑制することで学習結果の向上に繋がる

一般に、Batch Normalization は以下のように定義

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

E[x] : 入力 x のバッチ単位での平均値
Var[x] : 入力 x のバッチ単位での分散
γ, β : affine パラメータ

※affine 変換を施し、所望の分布での安定を図る

PyTorch での実装

PyTorch では torch.nn.BatchNorm2d で実装されている

```
CLASS  torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True,
       track_running_stats=True, device=None, dtype=None) [SOURCE]
```

主に使われる引数は

- num_features (必須)　：入力のチャネル数

利用例

batchnorm.py U

workspace > pytorch-training > codes > 04 > example > batchnorm.py

```python
import torch
from torch import nn


if __name__=="__main__":

    # 入力用のテンソル定義
    _in = torch.ones((32, 256, 64, 64))

    # batchnorm 定義 & 適用
    norm = nn.BatchNorm2d(num_features=256)
    print(repr(norm(_in).size()))

    # 確認用
    _in2 = torch.tensor([
        [3., 2., 5.],
        [16., 43., 1.],
        [18., 3.1, 56.]
    ]).unsqueeze(dim=0).unsqueeze(dim=0)
    norm2 = nn.BatchNorm2d(num_features=1, affine=False)
    print("===== before =====")
    print(repr(_in2))
    print("===== after =====")
    print(repr(norm2(_in2)))

```

問題　出力　デバッグ コンソール　ターミナル　ポート

```
root@adb300e75c11:/work# python codes/04/example/batchnorm.py
torch.Size([32, 256, 64, 64])
===== before =====
tensor([[[[ 3.0000,  2.0000,  5.0000],
          [16.0000, 43.0000,  1.0000],
          [18.0000,  3.1000, 56.0000]]]])
===== after =====
tensor([[[[-0.7067, -0.7596, -0.6008],
          [-0.0182,  1.4116, -0.8126],
          [ 0.0877, -0.7014,  2.1000]]]])
root@adb300e75c11:/work#
```

ReLU 層

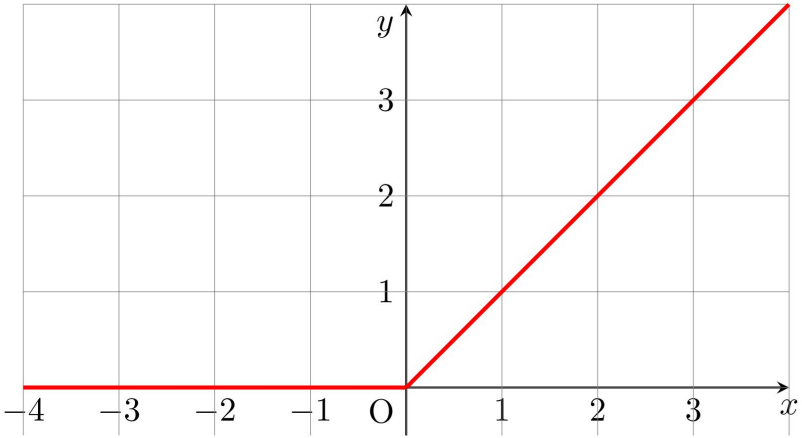
- Rectified Linear Unit の略称
- 活性化関数としての役割を持つ Layer

モデルに非線形性をもたらす関数のこと

これがないと、モデルの操作が線形結合で表せることがある
⇒ モデルの表現力の著しい低下

$$f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

ReLUの数式での定義



ReLU模式図

PyTorch での実装

PyTorch では torch.nn.ReLU で実装されている

```
CLASS  torch.nn.ReLU(inplace=False) [SOURCE]
```

今まで紹介した Layer とは異なり、
ReLU は基本的に引数を必要としない

利用例

relu.py U

workspace > pytorch-training > codes > 04 > example > relu.py

```python
import torch
from torch import nn


if __name__=="__main__":

    # 入力用のテンソル定義
    _in = torch.tensor([
        [-1., -2., 3.],
        [43., 21., -21.],
        [-0.5, 0, 0.1]
    ]).unsqueeze(dim=0).unsqueeze(dim=0)

    # ReLU 定義 & 適用
    relu = nn.ReLU()
    print(repr(relu(_in)))

```

問題　出力　デバッグ コンソール　ターミナル　ポート

```
root@adb300e75c11:/work# python codes/04/example/relu.py
tensor([[[[ 0.0000,  0.0000,  3.0000],
          [43.0000, 21.0000,  0.0000],
          [ 0.0000,  0.0000,  0.1000]]]])
root@adb300e75c11:/work#
```

モデル作成の復習

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#
```

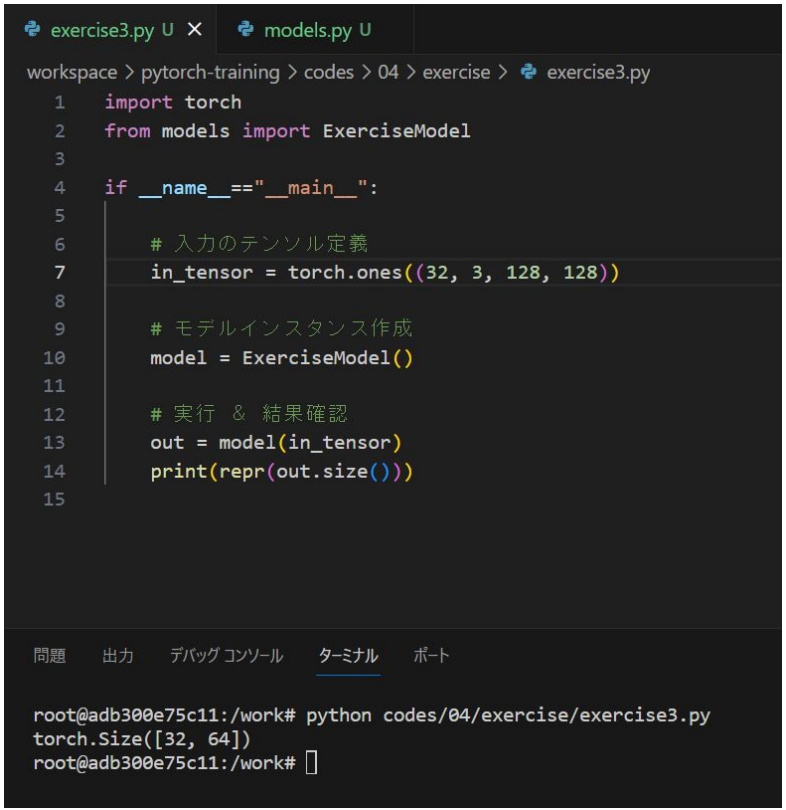
演習３

以下の条件に従うモデルのクラスを作成して下さい
入力のテンソルは次ページの画像を参照してください

1. forward で Convolution → Batch Normalization → ReLU → Linear の順に適用して出力する
2. Convolution は in_channels = 3, out_channels = 256, kernel_size = 5, stride = 8 であるとする
   ※ それ以外のパラメーターは自分で調整して下さい
3. 出力されるテンソルの形状は 32 × 64 であるとする

余裕のある人は同一ファイルにモデルを作るのではなく、
models.pyというファイルを別に用意し、
そこで定義したモノをimportしてみてください

解答例 & ヒント



Linear 層の入力の形状は
(B, N)
である必要があるが、

Convolution 層からの出力の形状は
(B, C, W, H)
のため、形状を変更する必要アリ

⇒ torch.Tensor.view() を利用する
※ view の利用例は次ページへ

利用例

view.py U

workspace > pytorch-training > codes > 04 > example > view.py

```python
import torch


if __name__=="__main__":

    # 入力のテンソルを定義
    _in = torch.ones((32, 8, 16, 16))

    # 形状を変更
    # (B, C, W, H) -> (B, N)
    out = _in.view(32, 8*16*16)

    # 確認
    print("===== before =====")
    print(repr(_in.size()))
    print("===== after =====")
    print(repr(out.size()))

```

問題　出力　デバッグ コンソール　ターミナル　ポート

```
root@adb300e75c11:/work# python codes/04/example/view.py
===== before =====
torch.Size([32, 8, 16, 16])
===== after =====
torch.Size([32, 2048])
root@adb300e75c11:/work#
```