

深層実習 第5,6回

中田尚

レポートの書き方

- 卒業論文に向けて体裁を重視します
 - 内容6点＋体裁2点
- 原則としてWordかPDF
 - 文字サイズ等は常識の範囲で
 - TeX（で作成したPDF）でも良い
 - **それ以外のファイル（ソースコードなど）は見ないと思ってください**
- 手書きの文章は不可
- スクリーンショットを挿入するのはOK
- 手書きの絵を挿入するのはOK

体裁2点のためには、
章立て必須です
昨年の「しりとりレポート」
を参照

☆課題1

- 2つの正方行列の積を求めるpythonプログラムを作成して、行列サイズと実行時間の関係を調べる
 - 計算ライブラリを使用してはいけません
 - time や matplotlib 以外の import は禁止
 - ライブラリを使った実行は課題2で予定しています
- レポートにはグラフを含めること
 - 単に載せるのではなくグラフから読み取れる内容を説明すること
 - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」
と思えるところまで
 - とはいえ1分以上の実行は不要です
- 締切：4/17 23:59

課題1

- 行列（2次元配列）と行列（2次元配列）の積の実行時間の測定

```
import time
```

```
N = 300 # サイズ
```

```
x = [[(i+j) for i in range(N)] for j in range(N)]
```

```
y = [[(i+j+1) for i in range(N)] for j in range(N)]
```

```
z = [[0 for i in range(N)] for j in range(N)]
```

```
for l in [50,100,150,200,250,300]:
```

```
    t1=time.time()
```

```
    for i in range(l):
```

```
        for j in range(l):
```

```
            for k in range(l):
```

```
                z[i][k] = z[i][k] + x[i][j] * y[j][k]
```

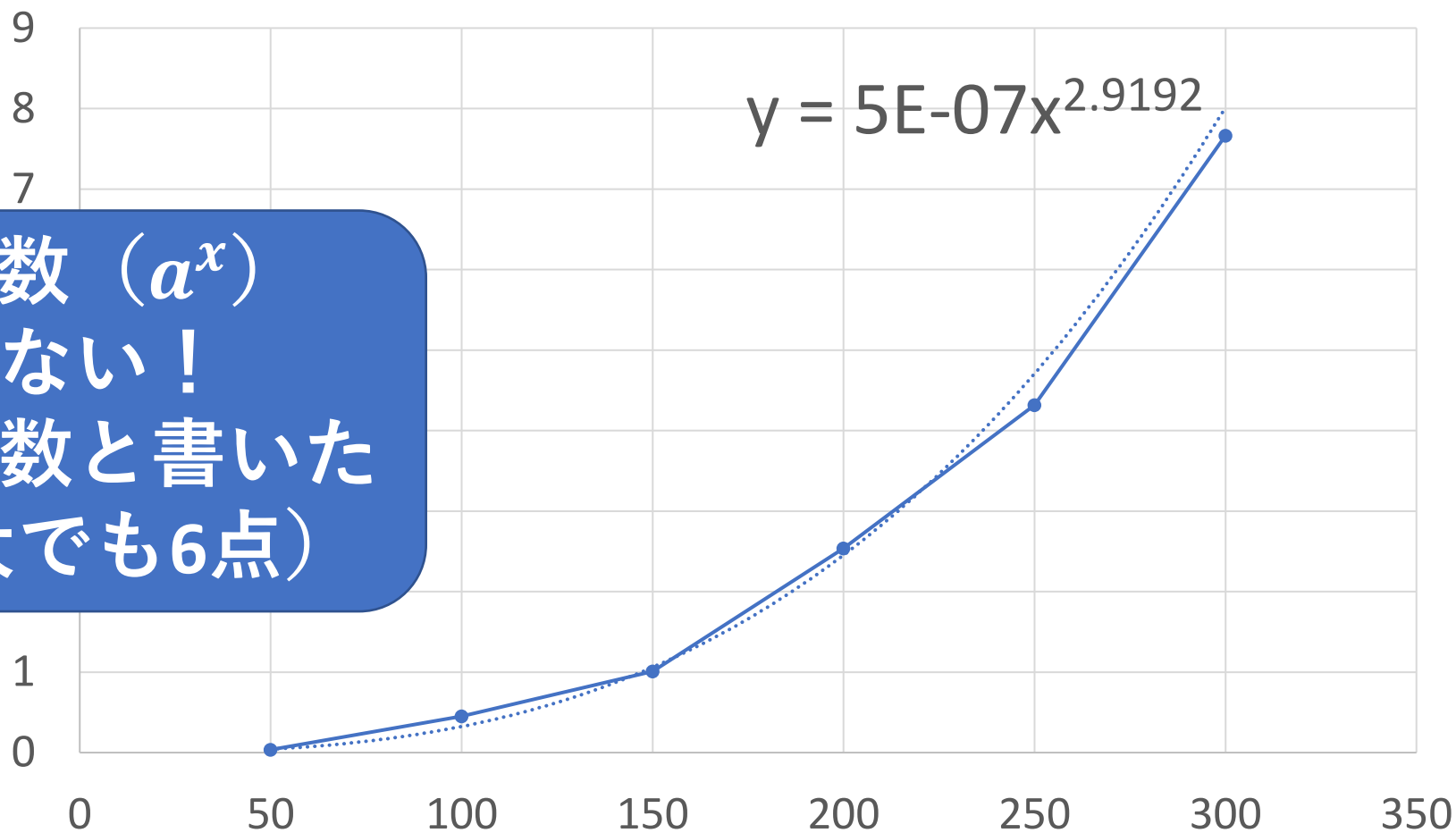
```
    t2=time.time()
```

```
    print(l,t2-t1)
```

提出遅れは減点ですが
忘れずに提出してください

実行時間は行列サイズの約3乗に比例

指数関数 (a^x)
ではない！
(指数関数と書いた
人は最大でも6点)





指数関数



3次関数



指数関数



3次関数



指数関数



3次関数

☆課題2

畳み込みは padding='valid' (小さくなる) で良い
理由を説明すれば 'same' (同じサイズ) でも良い

- **2つの正方行列の積と正方行列と 3×3 行列の畳み込み**それぞれを求めるpythonプログラムを作成して、それぞれの行列サイズと実行時間の関係を調べる
 - それぞれのpython, numpy, cupy, pytorchの実行時間について考察する
 - cupyの畳み込みは省略しても良い
- レポートにはグラフを含めること
 - 単に載せるのではなくグラフから読み取れる内容を説明すること
 - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」
と思えるところまで
 - とはいえ1分以上の実行は不要です
- 締切： 4/24(木) 23:59

GPUのメモリサイズによっては
10秒程度でメモリ不足になるかも
共有メモリが使われると
一気に遅くなるので注意

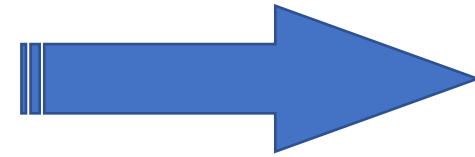
それぞれについて「行列サイズと実行時間の間にはXXという関係がある」
という考察が必要

課題2のヒント

- numpyはscipy.signal.correlate2dを使う
- scipy.signal.convolve2dはフィルタを反転させてから畳み込みを行うので、torchのconv2dと異なる結果になる
- **cupyx.scipy.signal.correlate2d が正しく動かない場合は省略して良い**
- pytorch はtorch.nn.functional.conv2d を使うが、入力は4次元である必要がある(次ページ参照)

pytorchのconv2d

- 入力データは[バッチサイズ][入力チャンネル][x][y]
フィルタは[出力チャンネル][入力チャンネル][x][y]
のデータを要求する
- 今回はバッチサイズ数も入出力チャンネル数も1とする
つまり、data[N][N]をdata[1][1][N][N]に変換する
 - data=torch.reshape(data,(1,1,N,N)) または
data=data.unsqueeze(0).unsqueeze(0)
で変換できる
- バッチサイズとチャンネルについては次回以降に説明



[1,1,1],
[1,-8,1],
[1,1,1]
が
[[[1,1,1],
[1,-8,1],
[1,1,1]]]
となる

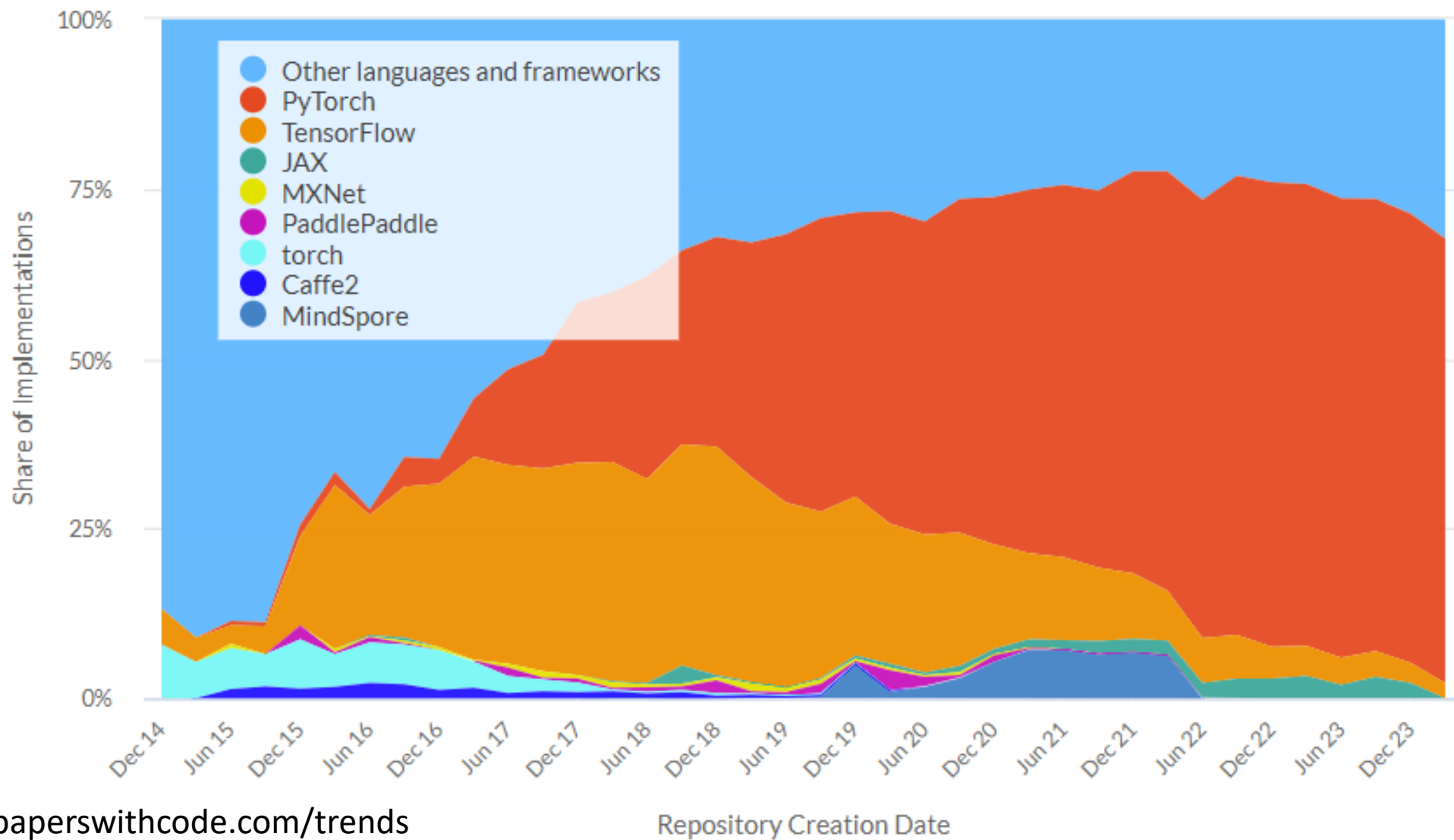
任意課題1 満点5点

- 課題1の実装は非常に遅く、numpyを使えば100倍以上の高速化が実現できるはずである
- numpyはC言語とFortranで実装されているので、C言語で課題1相当のコードを実装したところ、それでもnumpyの方が10倍程度早い（実行環境によってこの比率は大きく変わる）
- **C言語で実装しただけでは不十分な理由を考えよ**
 - 仮説であれば検証も行うこと（検証方法も自分で考える）
- **解消可能な理由であれば、自ら実装を行い実行速度がnumpyに近付くことを確かめよ**
 - numpyはGPUを使わないので、GPUは使用禁止
 - cupyとGPU実装で比較しても良い（さらに高難易度）
- 締切：4/24(木) 23:59

深層実習 第5,6回

中田尚

AIフレームワークのシェア



フレームワークの違い



項目	PyTorch	TensorFlow
開発者	Facebook AI	Google Brain
コア言語	Python	Python, C++, CUDA
デバッグの容易さ	よりデバッグしやすい (eager execution)	グラフベースの実装でデバッグが困難
ONNX対応	対応	対応
人気	成長中の人気、学術研究で好まれる	産業界で広く利用されている

データ相互変換可能

- 既存モデルの改良や新しいモデルを自作するにはPyTorchが良い
- すでにあるモデルを使うだけならTensorFlowが良い

一般的な機械学習

- データの読み込み
 - サンプルデータ
 - 自作のデータ
- モデルの作成
 - 何かを参考にするのがお手軽
 - 全く新しい構造を作り出すのはかなり大変
 - 規模さえ大きくすれば性能が良くなることがわかりつつある
 - 構造の工夫はあくまで最適化（同じ性能をより低コストで実現する手段）
- 学習
 - 試行錯誤が必要なことも多い

データの読み込み (DataLoder)

- サンプルデータの利用
 - `torchvision.datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)`
のようにするだけ
 - 自動でダウンロードしてくれる
- 自作学習データの準備
 - CSV
 - 画像
 - (その他?)

CSVファイルの読み込み

- Pandas

- 様々なデータの読み込みや処理に対応したライブラリ

- CSVファイルの例

- a ,b ,c ,d
 - 11,12,13,14
 - 21,22,23,24
 - 31,32,33,34



これは良くない例

失敗例

入力 [1]: ▶ `import pandas as pd`
`df = pd.read_csv("sample.csv")`

入力 [2]: ▶ `print(df.columns)`

よく見るとスペース
が入っている

`Index(['a ', 'b ', 'c ', 'd'], dtype='object')`

入力 [3]: ▶ `print(df["a"])`

`KeyError` Traceback (most recent call last):
 File "/opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py", line 3360, in
 try:
-> 3361 return self._engine.get_loc(casted_key)
 3362 except KeyError as err:

キー ("a") が無い

不要なスペースは入れてはいけない

- **a , b , c , d**

- 11,12,13,14

- 21,22,23,24

- 31,32,33,34

ではなく

- **a,b,c,d**

- 11,12,13,14

- 21,22,23,24

- 31,32,33,34

と詰めて書かなければならない

- Excelでcsv保存していれば大丈夫

やり直し

入力 [1]: ▶ `import pandas as pd`
`df = pd.read_csv("sample.csv")`

入力 [2]: ▶ `print(df.columns)` 余計なスペースが無い
`Index(['a', 'b', 'c', 'd'], dtype='object')`

入力 [3]: ▶ `print(df["a"])`

自動で連番が付く

```
0    11
1    21
2    31
Name: a, dtype: int64
```

Datasetの作成

- 目標
 - `dataset = csvdataset("test.csv")`
 - `dataloader = DataLoader(dataset, batch_size=8)`
としたい
- CSVファイルのローカルルール
 - 1列目が分類(label) (0から連続する整数)
 - 2列目以降がN次元データ
 - `dataloader`の返値は[8][N]

label,a,b,c

0,12,13,14

0,22,23,24

1,32,33,34

2,42,43,44

実行例

- `batchsize=2, shuffle=False`の例

```
▶ for label,data in dataloader:  
    print("label=",label)  
    print("label[0]=",label[0])  
    print("data=",data)  
    print("data0",data[0])
```

1ループ
=
1バッチ

```
label= tensor([0, 0])  
label[0]= tensor(0)  
data= tensor([[11, 12, 13],  
              [21, 22, 23]])  
data0 tensor([11, 12, 13])  
label= tensor([1, 1])  
label[0]= tensor(1)  
data= tensor([[31, 32, 33],  
              [41, 42, 43]])
```

2個セット

3次元データが
2セット

label,a,b,c
0,12,13,14
0,22,23,24
1,32,33,34
2,42,43,44

Datasetの自作

- Datasetを継承
 - `class myDataset(Dataset):`
 - どちらかが必須
 - `__iter__(self)`
 - ややこしいので省略 See IterableDataset
 - `__getitem__(self, index)`
 - `return self.data[index]`
 - `def __init__(self, csv_path):`
 - `self.data`に読み込んだデータを保存する
 - `def __len__(self):` #データを返す
 - `return len(self.data)`

csvdataset.py (N次元データ)

- #csvdataset.py
- import pandas as pd
- import torch
- from torch.utils.data import DataLoader, Dataset, random_split
- class csvdataset(Dataset):
 - def __init__(self, csv_path):
 - df = pd.read_csv(csv_path)
 - **self.labels = df['label']**
 - **self.data = df.iloc[:,1:].values**
 - def __getitem__(self, index):
 - **label=self.labels[index]**
 - **data=self.data[index]**
 - return label,data
 - def __len__(self): #データを返す
 - return len(self.data)

df =
Data
Frame
の略

pandasで読み込み

2つ並べたリスト

df = Data Frame

label, a, b, c

0	12	13	14
0	22	23	24
1	32	33	34
2	42	43	44

labels

data

画像データセット

- CSVファイルにファイル名を書く
- labelとpathの2列

```
label,path
0,./cifar10/airplane1.png
0,./cifar10/airplane4.png
1,./cifar10/automobile1.png
1,./cifar10/automobile10.png
```

```
#csvimgdataset.py
```

```
import pandas as pd
```

```
import torch
```

```
from torch.utils.data import DataLoader, Dataset, random_split
```

```
#from PIL import Image
```

```
from torchvision import transforms
```

```
from matplotlib import pyplot as plt
```

```
from torchvision.io import read_image
```

- #csvdataset.py
- import pandas as pd
- import torch
- from torch.utils.data import DataLoader, Dataset

```
class csvimgdataset(Dataset):  
    def __init__(self, csv_path, transform=None):  
        df = pd.read_csv(csv_path)  
        self.labels = df['label']  
  
        self.image_paths = df['path']  
        self.transform = transform
```

- class csvdataset(Dataset):
- def __init__(self, csv_path):
- df = pd.read_csv(csv_path)
- self.labels = df['label']
- self.data = df.iloc[:,1:-1].values

```
def __getitem__(self, index):  
    label=self.labels[index]  
    path = self.image_paths[index]
```

img = Image.open(path) #0-255の画像データとして読み込み

img = read_image(path)/255.0 # 0-1のTensorとして読み込み

if self.transform is not None:

img = self.transform(img)

return label, img, path

```
def __len__(self):  
    return len(self.image_paths)
```

- def __getitem__(self, index):
- label=self.labels[index]
- data=self.data[index]
- return label,data
- def __len__(self): #データを返す
- return len(self.data)

実行例

```
transform = torch.nn.Sequential(transforms.Resize((32,32)))
dataset = csvimgdataset("./imglist.csv",transform)
dataloader = DataLoader(dataset, batch_size=4)
for label ,img, path in dataloader:
    print('label=',label,' path=',path)
    print('img.shape=',img.shape)
    fig, ax = plt.subplots(1, len(img), figsize=(len(img),1))
    for i in range(len(img)):
        ax[i].xaxis.set_major_locator(plt.NullLocator())
        ax[i].yaxis.set_major_locator(plt.NullLocator())
        ax[i].imshow(img[i].permute(1,2,0),cmap="bone")
    plt.show()
```

確認用の
画像表示

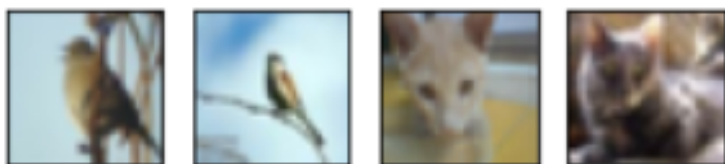
実行結果

- `transform = torch.nn.Sequential(transforms.Resize((32,32)))`

```
label= tensor([0, 0, 1, 1]) path= ('./cifar10/airplane1.png', './cifar10/airplane  
4.png', './cifar10/automobile1.png', './cifar10/automobile10.png')  
img.shape= torch.Size([4, 3, 32, 32])
```



```
label= tensor([2, 2, 3, 3]) path= ('./cifar10/bird4.png', './cifar10/bird6.png',  
'./cifar10/cat1.png', './cifar10/cat2.png')  
img.shape= torch.Size([4, 3, 32, 32])
```



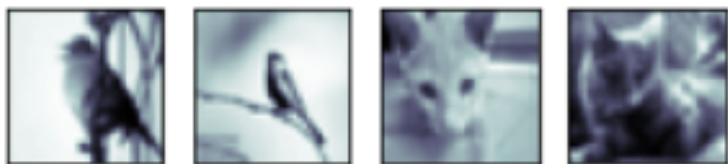
実行結果（グレースケール）

- `transform = torch.nn.Sequential(transforms.Resize((32,32)),transforms.Grayscale())`

```
label= tensor([0, 0, 1, 1]) path= ('./cifar10/airplane1.png', './cifar10/airplane4.png', './cifar10/automobile1.png', './cifar10/automobile10.png')  
img.shape= torch.Size([4, 1, 32, 32])
```



```
label= tensor([2, 2, 3, 3]) path= ('./cifar10/bird4.png', './cifar10/bird6.png', './cifar10/cat1.png', './cifar10/cat2.png')  
img.shape= torch.Size([4, 1, 32, 32])
```



torchvision.transforms

- Compose 複数の Transform を連続して行う
 - 今はtorch.nn.Sequentialが推奨されている
- Pad パディングを行う（画像を大きくする）
- CenterCrop 画像の中心を切り抜く
- Resize リサイズを行う
- Grayscale グレースケール変換を行う
- GaussianBlur ガウシアンフィルタを適用する
- RandomCrop ランダムに画像を切り抜く
- RandomResizedCrop ランダムに切り抜いたあとにリサイズを行う
- RandomHorizontalFlip ランダムに左右反転を行う
- RandomVerticalFlip ランダムに上下反転を行う
- RandomRotation ランダムに回転を行う

DataLoader

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,  
            batch_sampler=None, num_workers=0, collate_fn=None,  
            pin_memory=False, drop_last=False, timeout=0,  
            worker_init_fn=None, *, prefetch_factor=2,  
            persistent_workers=False)
```

DataLoaderのオプション

- **dataset,**
 - データ本体（必須）
 - イテラブルであること
 - for i in data の dataの部分に書いてもエラーにならないこと
- **batch_size=1,**
 - バッチサイズ
 - 何個ずつ取り出すか
 - 最後の端数はここで指定した数字よりも小さくなることもある
- **shuffle=False,**
 - 順序をランダムにするか
- **sampler=None,**
 - 順序をランダムでは無く、自分で決めたいときに関数を与える

DataLoaderのオプション

- `batch_sampler=None`,
 - バッチ処理の時に組み合わせを自分で決めたいときに関数を与える
 - 例えば同一種類は同一バッチに1個ずつにしたいとき
- `num_workers=0`,
 - マルチプロセス化 読み込み処理が重いときに効果有り。画像を毎回拡大縮小する場合とか。
- `collate_fn=None`,
 - バッチ処理の動作を変更したいときに関数を与える
 - `batch_sampler`は順序のみ指定するのでそれ以上の細かいことをしたい場合
- `pin_memory=False`,
 - GPUのメモリ転送を最適化する 遅くなることもあるらしいので注意

DataLoaderのオプション

- `drop_last=False`,
 - バッチ処理で割り切れない最後の部分を捨てる
 - シャッフルしていれば捨てられるデータはランダム
 - 割り切れない部分の処理でバグがあるときの逃げの手段にもなる
- `timeout=0`,
 - 読み込み制限時間
 - ネットワークから読み込む場合に処理が止まらないようにできる
 - ただし、`timeout`したときの処理を考える必要がある
- `worker_init_fn=None`,
 - 独自の前処理
 - `DataLoader`起動時に何かしなければいけないとき

DataLoaderのオプション

- *,
 - 以降はキーワード専用パラメーター
- prefetch_factor=2,
 - 先読み
 - 暇(?)な時に実は次を読み込んでいるので、その量を指定する
- persistent_workers=False
 - 読み込みプロセスを再利用する
 - 読み込み時間が短くなることが期待出来るが、メモリ使用量も増える
 - 「読み込み→学習→読み込み」の「学習」中もメモリを使い続けるので

キーワード専用パラメーターとは

- DataLoader(`dataset`, `batch_size=1`, `shuffle=False`, `sampler=None`, `batch_sampler=None`, `num_workers=0`, `collate_fn=None`, `pin_memory=False`, `drop_last=False`, `timeout=0`, `worker_init_fn=None`, *, `prefetch_factor=2`, `persistent_workers=False`)

は

- DataLoader(`data`, `1`, `False`, `None`, `None`, `0`, `None`, `False`, `False`, `0`, `None`, `prefetch_factor=2`, `persistent_workers=False`)
- のように青文字の部分を省略できるが、最後の2つを指定する場合は必ず `prefetch_factor=` のように書かなければならない
- 一般的に
 - 前から順に定義順に並べる限り `XXX=` は省略できる (*の手前まで)
 - 途中を省略するには (書きたい部分だけ) `XXX=` と書く
 - 一度 `XXX=` と書いたらそれ以降は必ず `XXX=NNN` と書く必要がある
 - 省略した部分は定義の `=` の右に書かれた値になる (`0`, `1`, `None`, `False` など)
 - `=` の無い引数は省略できない (`dataset=data` のように書くことはできる)

Datasetの分割

- 学習データと評価データは明確に分離する必要がある
 - 最低でも2分割だが、訓練用（train）、評価用（val）、テスト用（test）の3分割にすることが多い
 - valは訓練中に進捗状況の確認用
 - testは最終確認にのみ使用

Datasetの分割

```
train_ratio = 0.7
```

```
val_ratio = 0.2
```

```
test_ratio = 0.1
```

```
train_size = int(train_ratio * len(dataset))
```

```
val_size = int(val_ratio * len(dataset))
```

```
test_size = len(dataset) - train_size - val_size
```

test_size = int(test_ratio * len(dataset))
としてはいけない

```
train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])
```

```
train_dataloader = DataLoader(train_dataset, batch_size=4)
```

```
val_dataloader = DataLoader(val_dataset, batch_size=4)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=4)
```

分割結果の確認

```
print("train")
for label,data in train_dataloader:
    print("label=",label)
    print("data=",data)
print("val")
for label,data in val_dataloader:
    print("label=",label)
    print("data=",data)
print("test")
for label,data in test_dataloader:
    print("label=",label)
    print("data=",data)
```

```
train
label= tensor([0, 1, 1, 1])
data= tensor([[11, 12, 13],
              [31, 32, 33],
              [51, 52, 53],
              [61, 62, 63]])
label= tensor([1])
data= tensor([[71, 72, 73]])
val
label= tensor([1])
data= tensor([[41, 42, 43]])
test
label= tensor([0, 2])
data= tensor([[21, 22, 23],
              [81, 82, 83]])

8=5+1+2に分割されている
```

演習：独自のDatasetを作成する

- 次週に学習を説明してから、**Dataset**作成を含めた課題とする予定なので、今のうちに使えるようになりましょう
 - 例：シャッフルの有無で学習効果が変化するかどうか
- CSVファイルを作成して、**Dataset**を作る
 - 1次元や3次元**以外**のデータで確認
 - バッチサイズやシャッフルの設定を変えて挙動を確認する
- ファイル名の**CSV**ファイルを作成して、画像の**Dataset**を作る
 - **transform**をいくつか試して画像が変わることを確認する

課題2

- 特に行列積演算で**GPU**があまり速くないことがある



GPUの行列積計算がCPUと比較して早くない場合に考えられる原因は



GPUの行列積計算がCPUと比較して早くない場合に考えられる原因はいくつかあります。

1. 問題サイズ: 行列のサイズが小さい場合、**GPU**の並列処理能力を十分に活用できず、**CPU**と比較して速度向上が見られないことがあります。

18. ハードウェア互換性: 使用しているハードウェアが、GPUやシステムとの互換性がない場合、パフォーマンスが低下することがあります。互換性を確認し、必要に応じてハードウェアのアップグレードを検討してください。

NA

他にありますか

 これまでに様々な要因を挙げてきましたが、さらにいくつか追加します。

43. ハードウェア互換性の問題: システムの他のハードウェアコンポーネントとGPUが完全に互換性がない場合、パフォーマンスが低下することがあります。ハードウェア互換性を確認し、必要であれば適切なハードウェアに変更してください。

例えば、NVIDIAのPascalアーキテクチャの一部のGeForce GTXシリーズGPUは、倍精度浮動小数点演算性能が単精度浮動小数点演算性能の1/32であり、約100倍遅いです。この性能差は、GPUのアーキテクチャやモデルによって異なる場合がありますが、一般的にGeForceシリーズでは倍精度浮動小数点演算性能が大幅に低い傾向にあります。

高精度演算が必要な場合は、NVIDIAのTeslaやQuadroシリーズのGPUなど、プロフェッショナル向けやデータセンター向けの製品を検討することをお勧めします。これらのGPUは、科学計算や機械学習などの高精度演算が求められるアプリケーションに対応しており、倍精度浮動小数点演算の性能も比較的高くなっています。

行列積の実行速度

- Quadro RTX 4000の性能⇒
(デスクトップPC)

実行時間 [s]						
N	numpy64	torch64	numpy32	torch32	torch16	
10000	5.823	7.622	2.890	0.337	0.092	

- 単精度 = float32
- 倍精度 = float64
- 倍精度は単精度の1/32の性能しかないのでCPUと同等の性能

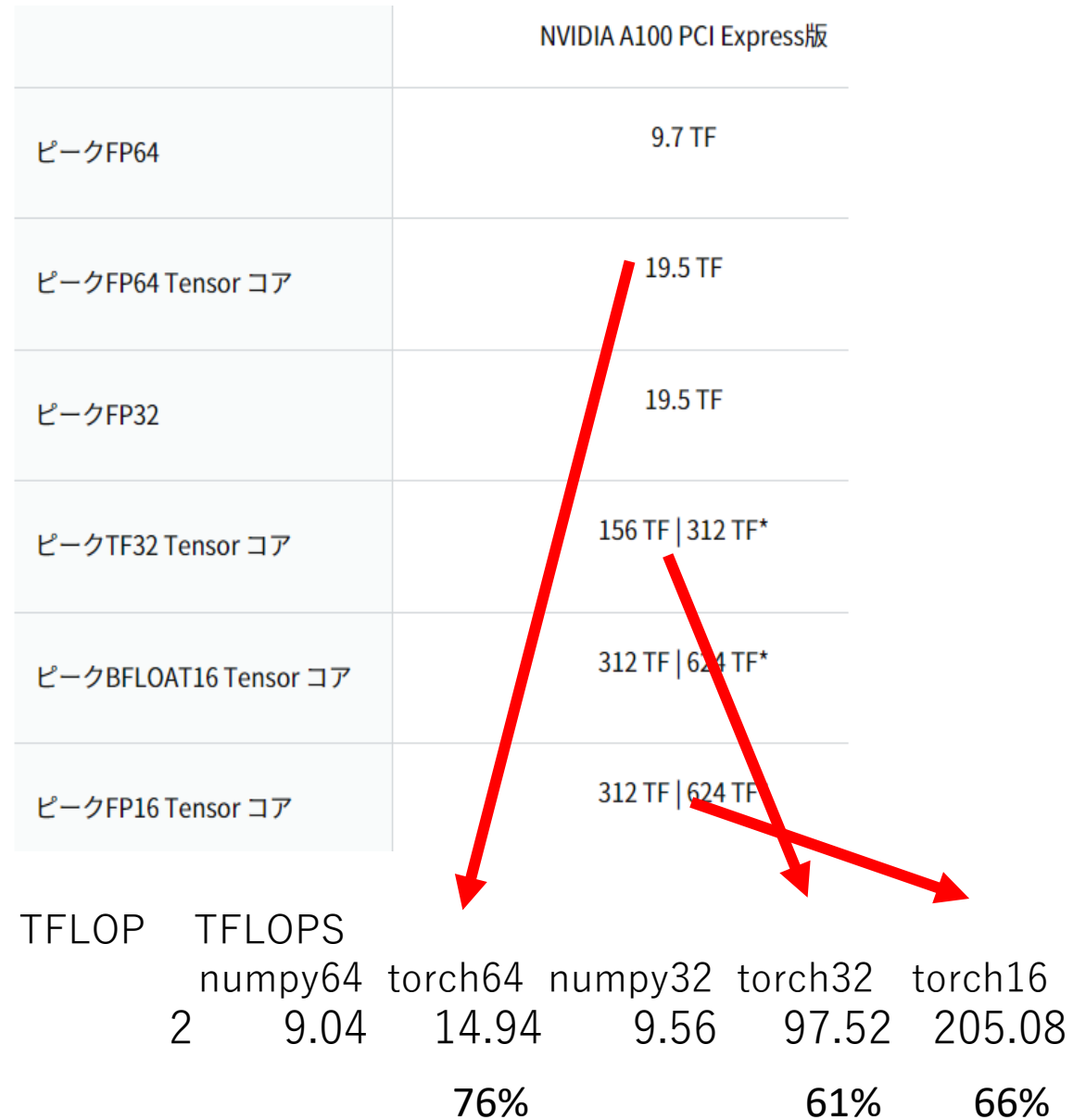
FLOPS (32:1:256)		
単精度 (TFLOPS)	倍精度 (GFLOPS)	Tensor (TFLOPS)
7.12	222.5	57.0

TFLOP	TFLOPS				
	numpy64	torch64	numpy32	torch32	torch16
2	0.172	0.262	0.346	5.934	21.816
		116%		84%	38%

行列積の実行速度

- NVIDIA A100の性能⇒
(AIサーバ)
- Tensorコア≡行列積専用

実行時間 [s]					
N	numpy64	torch64	numpy32	torch32	torch16
10000	0.2210	0.1338	0.2092	0.0205	0.0097




変数の型

- `tensor(1488.8413, device='cuda:0', dtype=torch.float64)`
- のように表示される
- `numpy`の型指定
(defaultはfloat64)
 - `npx=np.random.rand(N, N).astype(np.float32)`
- `torch`の型指定
(defaultはfloat32、`from_numpy`のときは元のまま)
 - `tox = torch.from_numpy(npx).to(torch.float64).to("cuda")`

課題2の追加考察（必須では無い）

`tox = torch.from_numpy(np_x).to("cuda")` # numpyからコピー

`toy = torch.from_numpy(np_y).to("cuda")`  float64

の部分を以下のように変更する

`tox = torch.rand(n,n).to("cuda")` # pytorchで新規作成

`toy = torch.rand(n,n).to("cuda")`  float32

乱数の中身は変化するのは当然として、実行時間も大きく変わる可能性がある

（デスクトップPCと大学推奨ノートPCでは変わるはず）

実行時間が変わった原因を考察する