

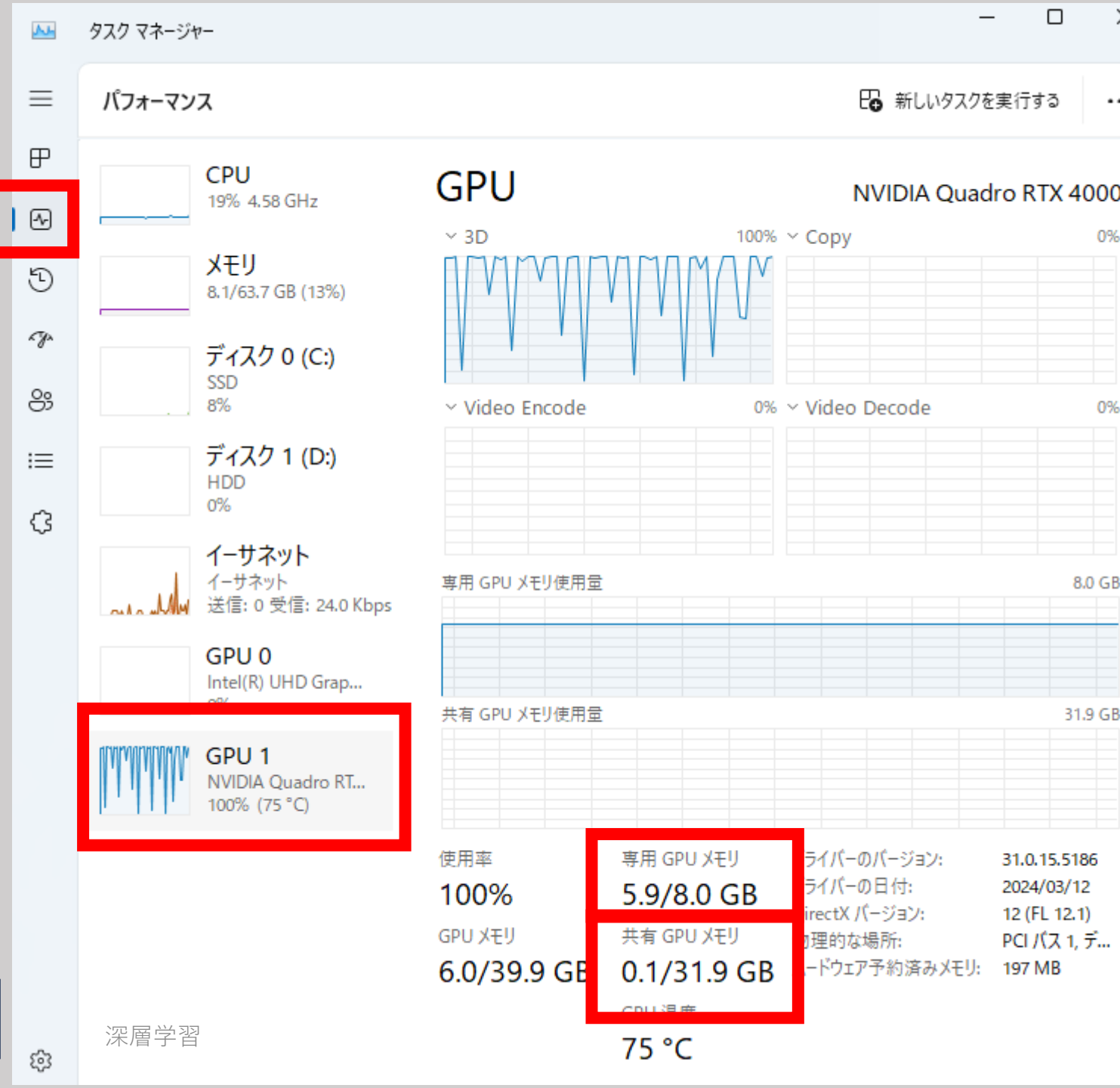
深層実習 第7,8回

中田尚

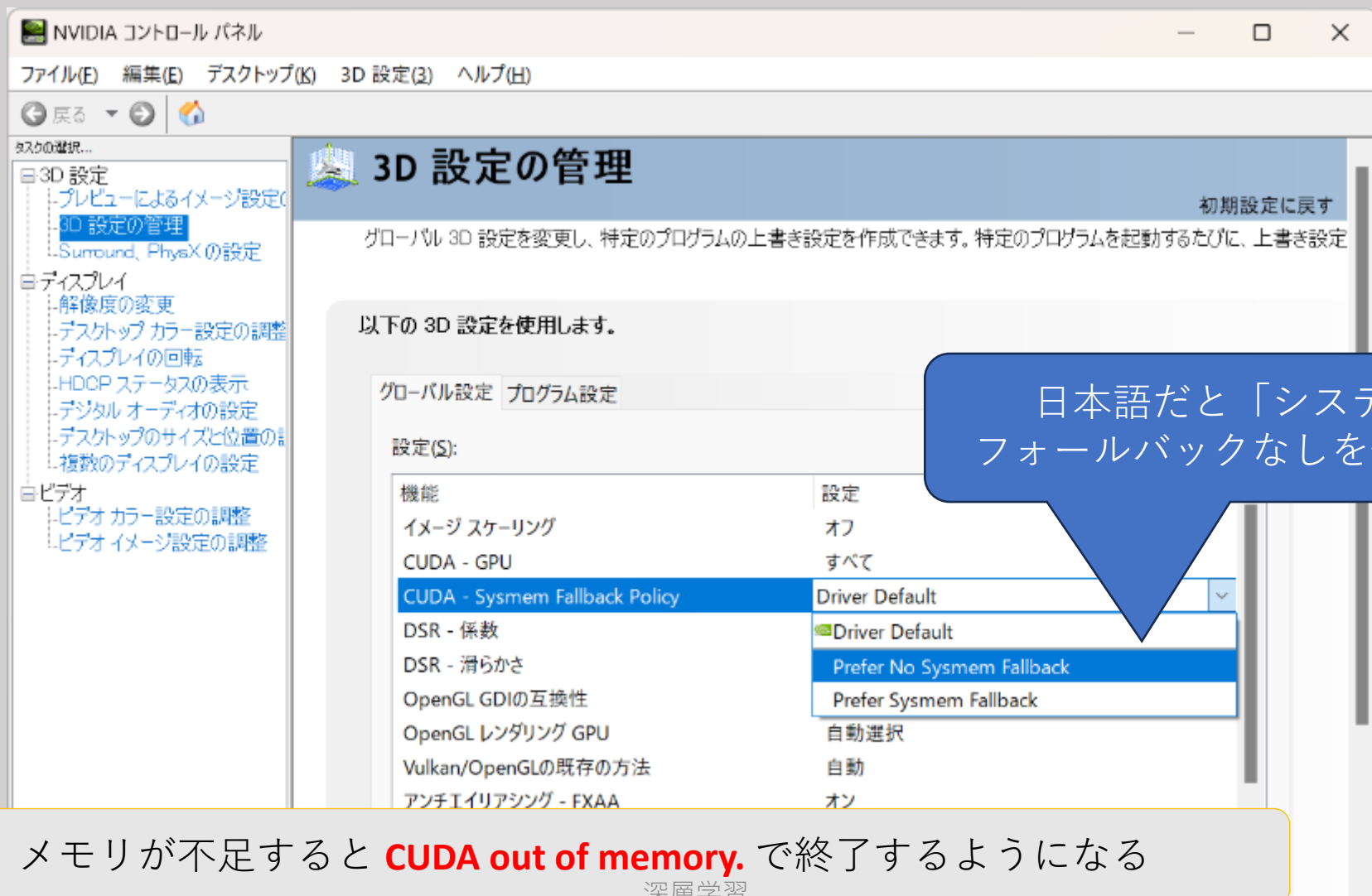
確認方法

- タスクマネージャを起動
- パフォーマンスを選択
- GPU1を選択
 - NVIDIAと書いてあるGPU
- 専用GPUメモリに余裕があり、共有GPUメモリがほぼ0であることを確認
- 共有メモリを使うと実行速度が**劇的に遅く**なります

昨年まではメモリ不足で終了していた



共有GPUメモリを無効にする



NVIDIA コントロール パネル

ファイル(E) 編集(E) デスクトップ(K) 3D 設定(3) ヘルプ(H)

戻る

3D 設定の管理

初期設定に戻す

グローバル 3D 設定を変更し、特定のプログラムの上書き設定を作成できます。特定のプログラムを起動するたびに、上書き設定

以下の 3D 設定を使用します。

グローバル設定 プログラム設定

設定(S):

機能	設定
イメージ スケーリング	オフ
CUDA - GPU	すべて
CUDA - Systemmem Fallback Policy	Driver Default
DSR - 係数	Driver Default
DSR - 滑らかさ	Prefer No Systemmem Fallback
OpenGL GDIの互換性	Prefer Systemmem Fallback
OpenGL レンダリング GPU	自動選択
Vulkan/OpenGLの既存の方法	自動
アンチエイリアシング - FXAA	オン

日本語だと「システムメモリ
フォールバックなしを優先」を選ぶ

メモリが不足すると **CUDA out of memory.** で終了するようになる

深層学習

成績評価

- 共通
 - 出席80%以上、課題認定80%以上
- 中間レポート60%、演習成果報告書40%
 - 必ずしも発表一発勝負ではない。準備段階や提出資料も含める。
 - レポート
 - 2週に1つ程度を予定（8点×8回程度①～⑧）
 - 任意課題
 - 5点×2回程度
 - 演習成果報告20点×2⑨⑩

☆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

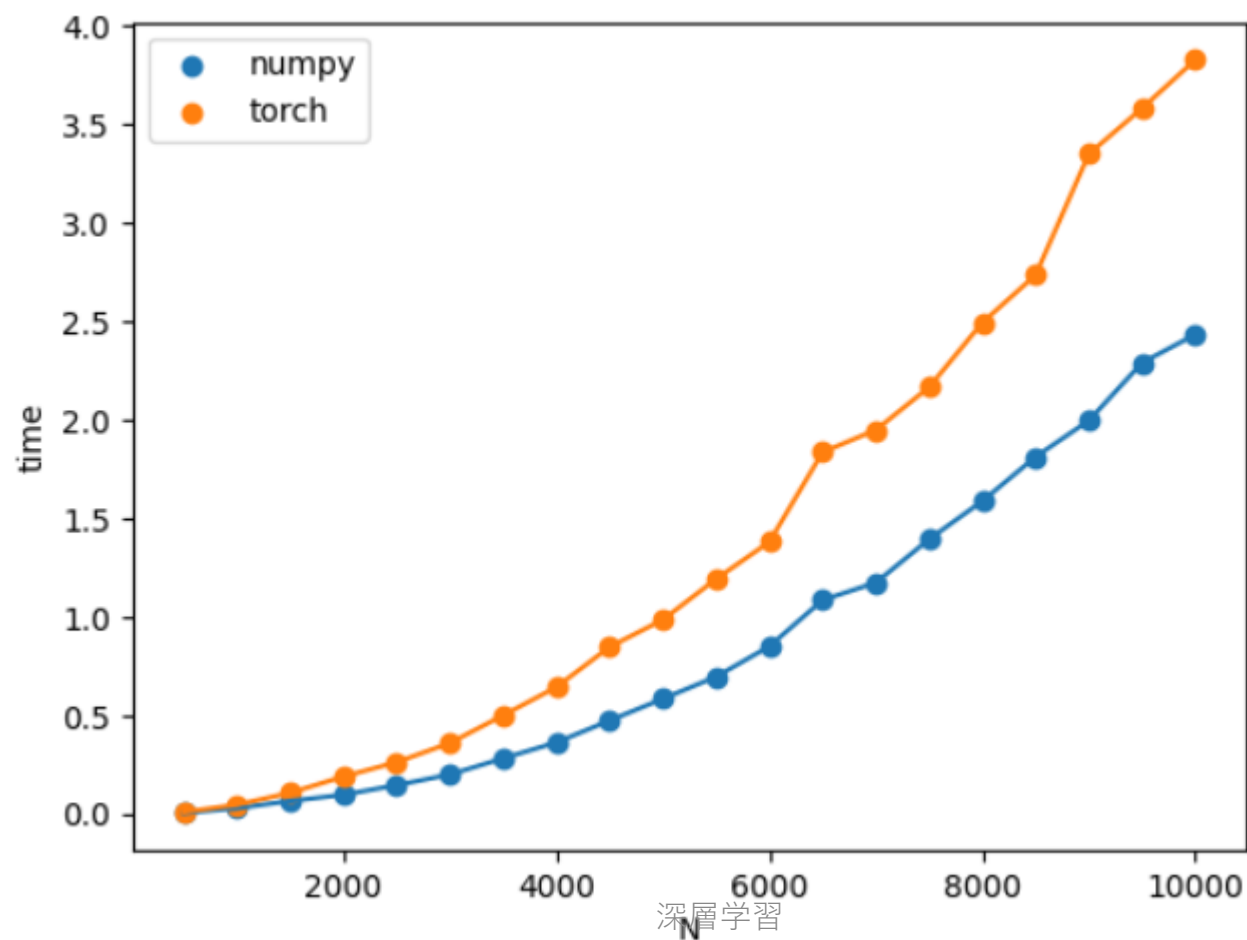
任意課題1 満点5点

- 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

課題2の考察例

- 演算回数を確認する
 - 行列積は $N \times N \times N$ の3重ループ
 - 畳み込みは $N \times N \times 3 \times 3$ の4重ループ
- オーダー記法
 - $O(N^3)$ とは N が十分大きいとき $M(N^3)$ となる定数 M が存在する
 - N^3 であれば $O(N^3)$ だし、 $9N^2$ であれば $O(N^2)$ となる
- つまり、畳み込みの実行時間は N^2 に比例するはず

実行してみる (CPUのみ)



表の書き方

- ダメな例

N	numpy	torch
5000	6.97910	4.441071
10000	24.545112	15.901942

数値の書き方

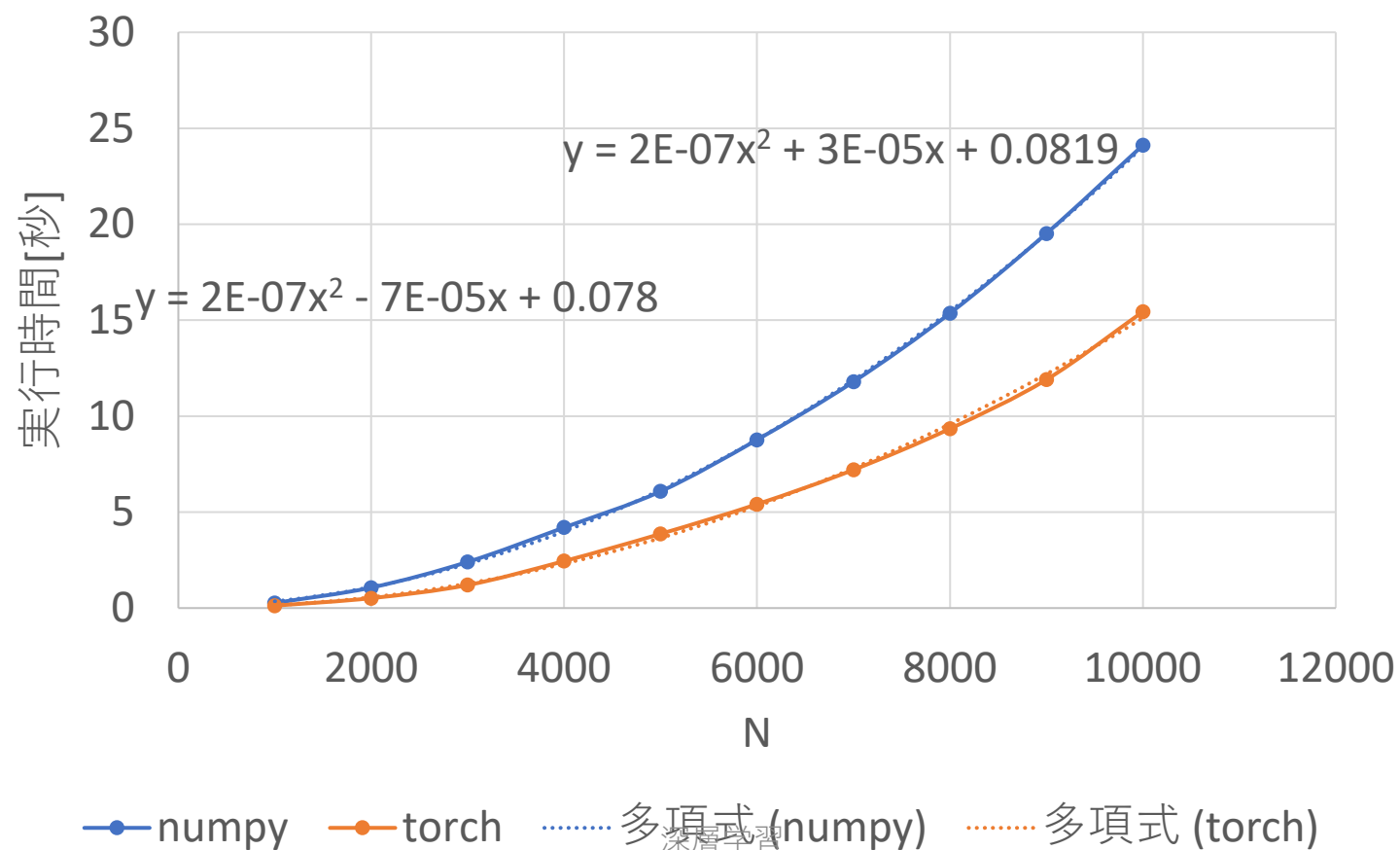
- ダメな例

N	numpy	torch
5000	6.97910	4.441071
10000	24.545112	15.901942

- 単位を付ける 有効数字を意識する

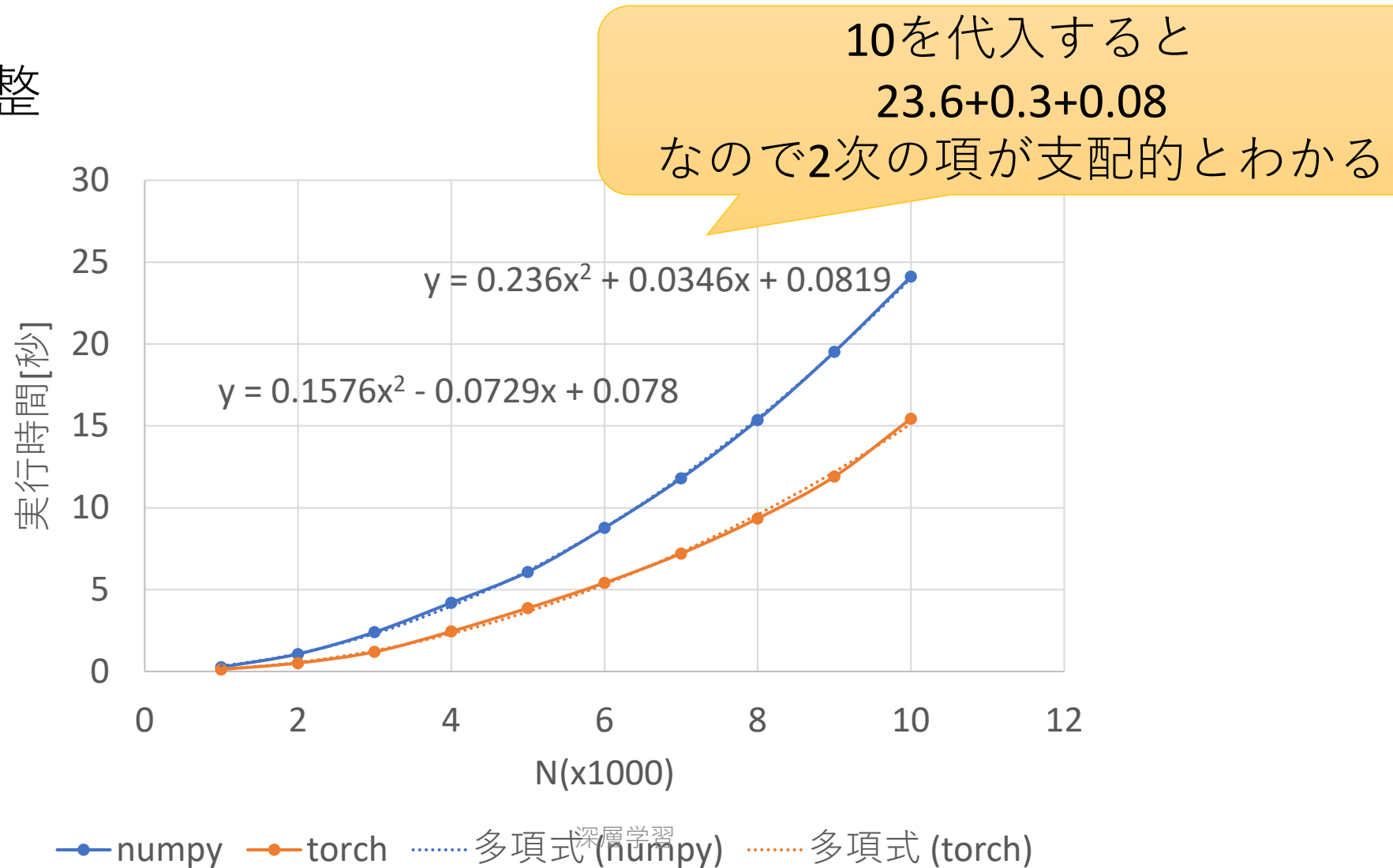
N	numpy[s]	torch[s]
5000	6.98	4.44
10000	24.55	15.90

excelで近似曲線を求めてみる



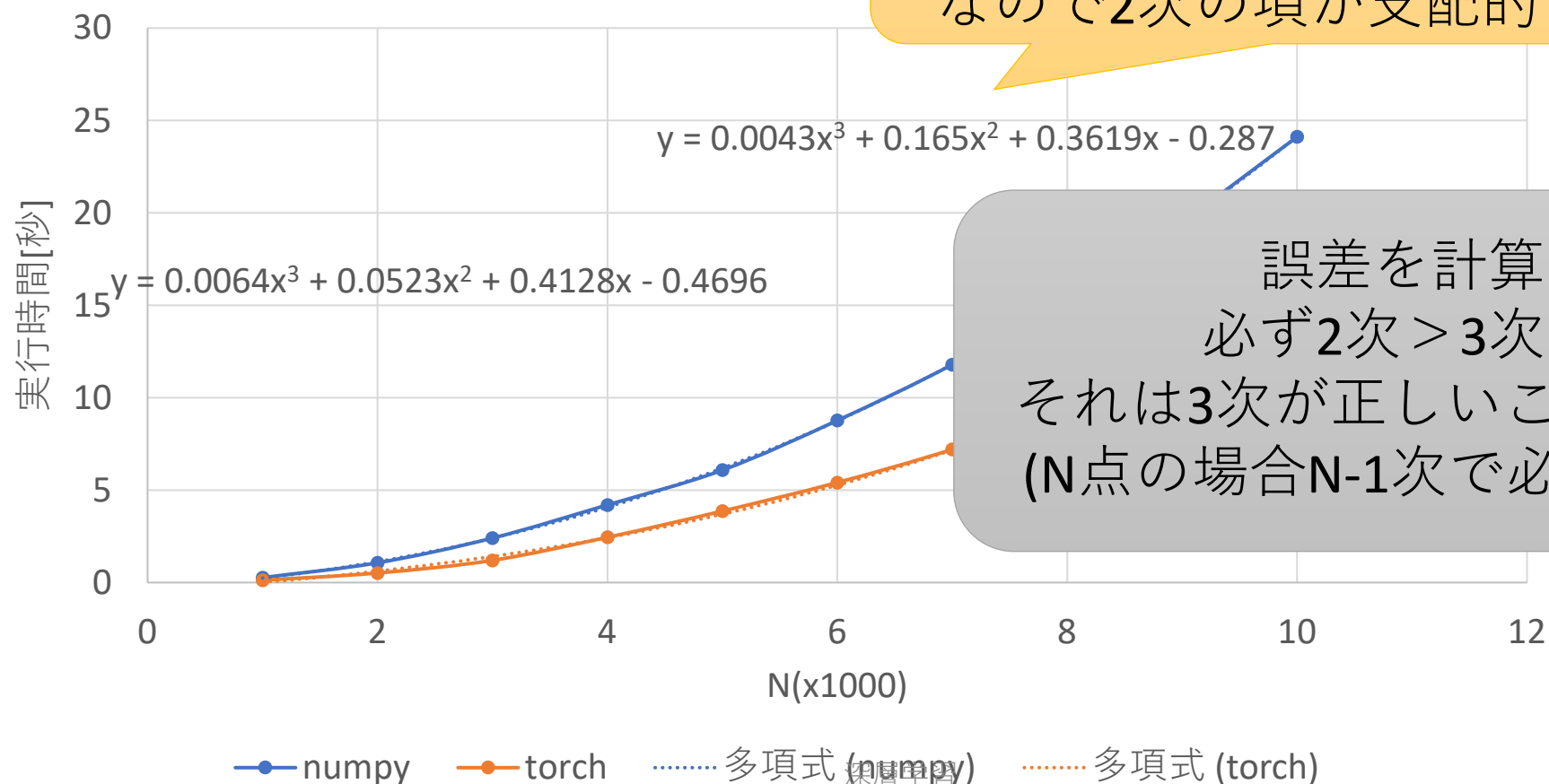
excelで近似曲線を求めてみる

- 軸を調整



excelで近似曲線を求めてみる

- 3次もやってみる



10を代入すると
 $4.3+16.5+0.3+0.28$
なので2次の項が支配的とわかる

誤差を計算すると
必ず2次>3次になるが
それは3次が正しいことにはならない
(N点の場合N-1次で必ず誤差0になる)

結論

- 行列積は略
- 畳み込みの実行時間は行列サイズの2乗に比例する
 - CPUであってもpytorchの方が1.56倍速い
 - CPU使用率を見る限りでは、pytorchはCPUでも並列化されている

深層実習 第7,8回

中田尚

ソースコード

- mnist.py
 - 全体（学習を含む）
- mnist_test.py
 - 前半の読み込み部分のみ
- 今回はPyCharm推奨
 - VS Codeではグラフがその場でない

簡単なパーセプトロン (3層の例)

input_size=1

hidden_size=30

output_size=10

```
class MyModel(nn.Module):
```

```
    def __init__(self):
```

```
        super(MyModel, self).__init__()
```

```
        self.layer1 = nn.Linear(input_size, hidden_size)
```

```
        self.layer2 = nn.ReLU()
```

```
        self.layer3 = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
```

```
        x = self.layer1(x)
```

```
        x = self.layer2(x)
```

```
        x = self.layer3(x)
```

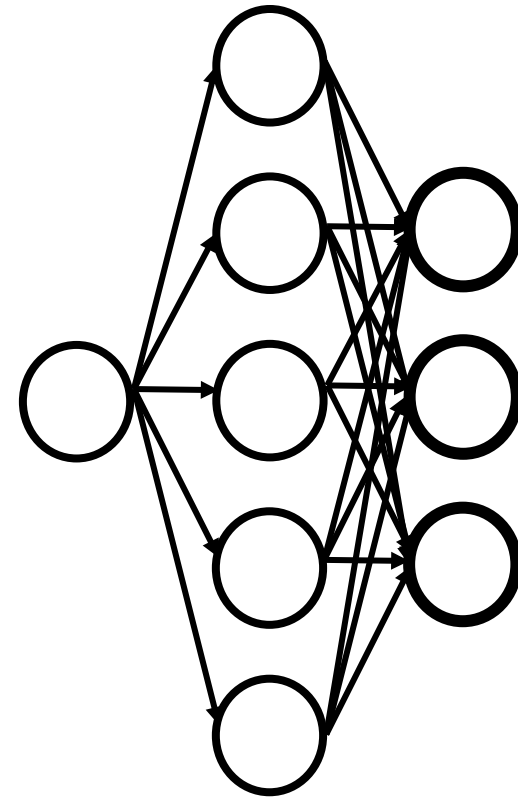
```
        return x
```

```
model = MyModel()
```

```
criterion = nn.CrossEntropyLoss()
```

```
#optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
optimizer = optim.Adam(model.parameters())
```



ReLUを__init__に書くかどうか



PyTorchを使用してニューラルネットワークを定義するには、
`torch.nn.Module`を継承したクラスを作成し、`__init__`メソッドで
ネットワークの構造を定義し、`forward`メソッドで順伝播を記述します。
以下に、3層の全結合ニューラルネットワークの定義例を示します。

python

Copy code

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.ReLU()
        self.layer3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
```

書いてある

深層学習

```
class MyMLP(nn.Module):
    def __init__(self, in_units, hidden_units, out_units):
        super(MyMLP, self).__init__()
        self.l1 = nn.Linear(in_units, hidden_units)
        self.l2 = nn.Linear(hidden_units, hidden_units)
        self.l3 = nn.Linear(hidden_units, out_units)

    def forward(self, x):
        h = F.relu(self.l1(x))
        h = F.relu(self.l2(x))
        y = self.l3(x)
        return y

model1 = MyMLP(20, 100, 2)

print(model1)

MyMLP (
  (l1): Linear (20 -> 100)
  (l2): Linear (100 -> 100)
  (l3): Linear (100 -> 2)
)
```

書いてない

ReLUを__init__に書くかどうか

- 結論
 - どちらでもよい
 - そもそもLayerに数えないことの方が多い（たぶん）
 - 活性化関数であり、層では無い
- 理由
 - 学習すべきパラメータがある場合は__init__に書かなければならない
 - さもなくば、パラメータが毎回初期化されてしまう
 - ReLUにはそんなパラメータは無い（純粋な関数）
 - シグモイドを使う場合も、同様の理由でどちらでも良い
- 違い
 - print(model)で表示されるかどうか
 - ただし、__init__ の定義順で表示されるので、forwardで同じReLUを複数回使うと結局1回しか表示されない
 - 複数回定義すれば複数回表示される

torchinfo.summary

- `pip install torchinfo` でインストールが必要
 - 実行時に警告が出るが無視して良い

```
class MyModel(nn.Module):  
    def __init__(self):  
        super(MyModel, self).__init__()  
        self.layer1 = nn.Linear(input_size, hidden_size)  
        self.layer2 = nn.ReLU()  
        self.layer3 = nn.Linear(hidden_size, output_size)  
  
    def forward(self, x):  
        x = self.layer1(x)  
        x = self.layer2(x)  
        x = self.layer3(x)  
        return x
```

`__init__` で
レイヤとして
定義

入力 [10]: `import torchinfo`

バッチサイズを含める

入力 [11]: `torchinfo.summary(model=model, input_size=(1, 1))`

出力[11]:

Layer (type:depth-idx)	Output Shape	Param #
MyModel	[1, 10]	--
├─Linear: 1-1	[1, 30]	60
├─ReLU: 1-2	[1, 30]	--
└─Linear: 1-3	[1, 10]	310
Total params: 370		
Trainable params: 370		
Non-trainable params: 0		

重み30(=1*30)
+ バイアス30

学習可能な
パラメータ無し

重み300(=30*10)
+ バイアス10

__init__に書かない場合

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer5 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.nn.functional.relu(self.layer1(x))
        x = torch.relu(self.layer3(x))
        x = self.layer5(x)
        return x
```

関数として呼ぶ
torch.nn.functional.relu(x)
torch.relu(x)
torch.nn.ReLU()(x)
どれも同じ

torch.nn.ReLU(x)
ではない

```
print(model)
```

```
MyModel(
  (layer1): Linear(in_features=1, out_features=30, bias=True)
  (layer3): Linear(in_features=30, out_features=30, bias=True)
  (layer5): Linear(in_features=30, out_features=10, bias=True)
)
```

```
torchinfo.summary(model=model, input_size=(1, 1))
```

Layer (type:depth-idx)	Output Shape	Param #
MyModel	[1, 10]	--
├─Linear: 1-1	[1, 30]	60
├─Linear: 1-2	[1, 30]	930
├─Linear: 1-3	[1, 10]	310

深層学習

ReLUは表示されない

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer5 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer3(x)
        x = self.relu(x)
        x = self.layer5(x)
        return x
```

再利用

```
print(model)
```

```
MyModel(
  (layer1): Linear(in_features=1, out_features=30, bias=True)
  (relu): ReLU()
  (layer3): Linear(in_features=30, out_features=30, bias=True)
  (layer5): Linear(in_features=30, out_features=10, bias=True)
)
```

1回だけ表示

```
torchinfo.summary(model=model, input_size=(1, 1))
```

```
]:
```

Layer (type:depth-idx)	Output Shape	Param #
MyModel	[1, 10]	--
├─Linear: 1-1	[1, 30]	60
├─ReLU: 1-2	[1, 30]	--
├─Linear: 1-3	[1, 30]	930
├─ReLU: 1-4	[1, 30]	--
├─Linear: 1-5	[1, 10]	310

2回表示

2回表示

```
Total params: 1,300
Trainable params: 1,300
```

深層学習

損失関数

- **CrossEntropyLoss**
 - 分類問題において標準的
 - 正解ラベルの値を大きく、不正解を小さくする
- 解空間を定義する

最適化アルゴリズム

- `torch.optim`に複数準備されている
- 基本的には解空間の傾きを求めてより誤差が小さくなる方向にパラメータを更新する

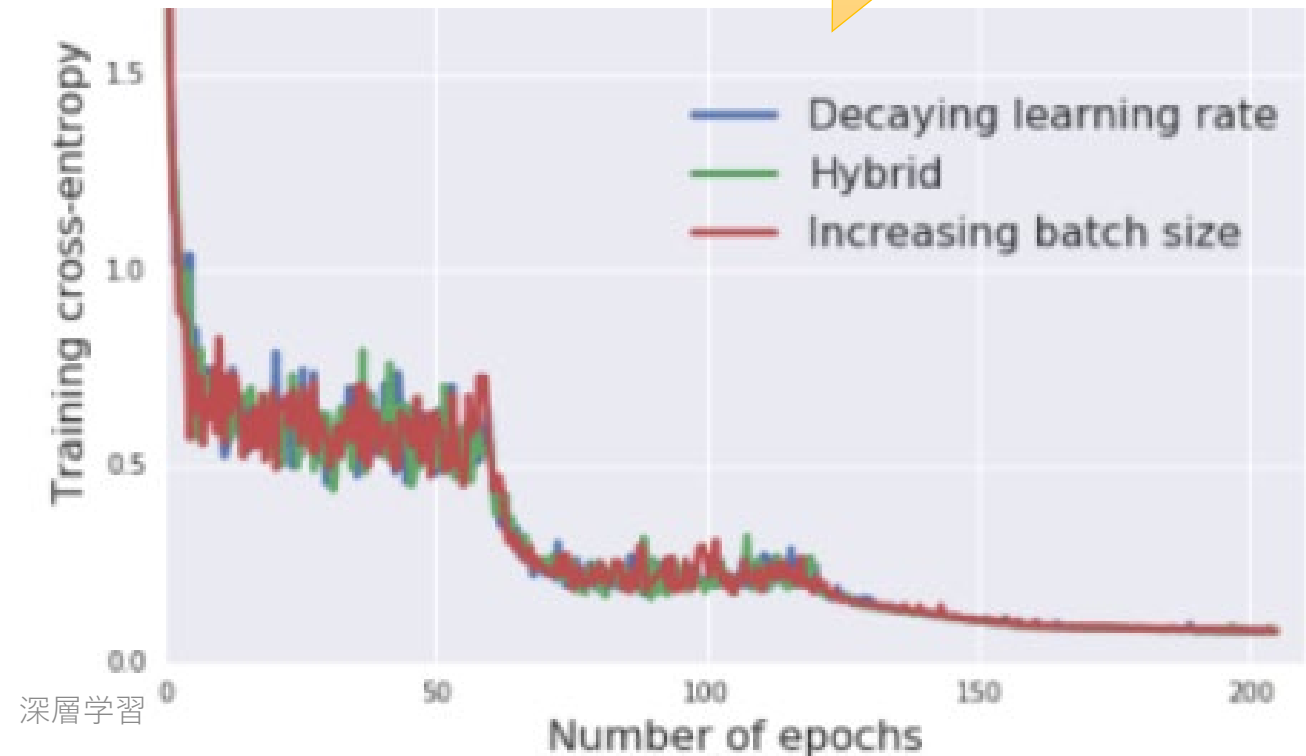
最適化アルゴリズムの例

- **SGD** 確率的勾配降下法
 - 基本的に忠実な方法だが、学習率の調整が難しい
 - 学習率が大きすぎると正解を通り過ぎてしまいいつまで経っても収束しない
 - 学習率が小さすぎるといつまで経っても正解にたどり着かない
- 学習率を自動で調整するアルゴリズム
 - RMSprop、Adagrad、Adadelata 等
- **Adam**
 - 加速と減速を組み合わせることで、「ボールが転がるように」最適化
 - 「とりあえずAdam」で良いことが多い
- Adamの改良
 - SparseAdam、AdamW 等

最適化パラメータ

- Learning Rate (LR)
 - 傾き * LR = 更新量
 - 100epoch毎に1/10にするというような時代もあった

SGDの典型的な
学習曲線



matplotlib

- レポートのためにグラフを描きたい
- PyCharmでは別画面に履歴付きで表示される（便利）
- Jupyter notebookではアニメーションにできる
（Jupyter labでのアニメーションのやり方は不明）
- 個人的にもいつもネットのサンプルを適当に使っていたので体系的に説明してみる

subplot/subplots

- `ax=plt.subplot()`
 - 場所を指定してサブグラフを1つ作成
- `fig, ax = plt.subplots()`
 - 複数の同じサイズのサブグラフを同時に作成

よくある省略記法

この書き方の説明が多い

- `plt.subplot(121)`
- `plt.plot(x,y1)`
- `plt.subplot(122)`
- `plt.plot(x,y2)`

- 上記を
- `plt.subplot(121)`
- `plt.subplot(122)`
- `plt.plot(x,y1)`
- `plt.plot(x,y2)`
- としたら動かない

- `ax1=plt.subplot(121)` #左半分
- `ax1.plot(x,y1)`
- `ax2=plt.subplot(122)` #右半分
- `ax2.plot(x,y2)`
- ↑こちらで説明
 - `ax1=plt.subplot(121)` #左半分
 - `ax2=plt.subplot(122)` #右半分
 - `ax1.plot(x,y1)`
 - `ax2.plot(x,y2)`
 - としても動く

グラフ全体(fig)とサブグラフ(ax)の取得

- `fig, ax = plt.subplots()`
 - 複数の同じサイズのサブグラフを同時に作成
 - 引数を省略するとサブグラフは一つ
 - `plt.subplots(1,2)`とすると左右に並んだ2つのサブグラフ
 - `ax`はリストで`ax[0]`と`ax[1]`
 - `plt.subplots(2,2)`とすると上下左右に並んだ4つのサブグラフ
 - `ax`は2次元リストで`ax[0][0]`, `ax[0][1]`, `ax[1][0]`, `ax[1][1]`
- `fig` は `fig=plt.figure()` と同じくグラフ全体を指す
- 以降ではサブグラフ1つで説明

折れ線オブジェクト

- `x`に`x`の値、`y[n]`に`y`の値が`n`本分格納されているとする
 - `x[100]`
 - `y[10][100]` ということ
- `line[i], =ax.plot(x,y[i], label=str(i))`
 - 返値はリスト
 - 要素数はグラフの本数だが、1本でもリストなので、`line[i]`, で1つ目の要素だけ保存して、残りを捨てている（1本の場合は何も捨てられずに、リストが解除されるだけ）
- 余談
 - `x[100]`
 - `y[100][10]` とすると
 - `line=ax.plot(x,y)`
 - で10本同時に描けるが、色やラベルを指定するのが面倒なので省略

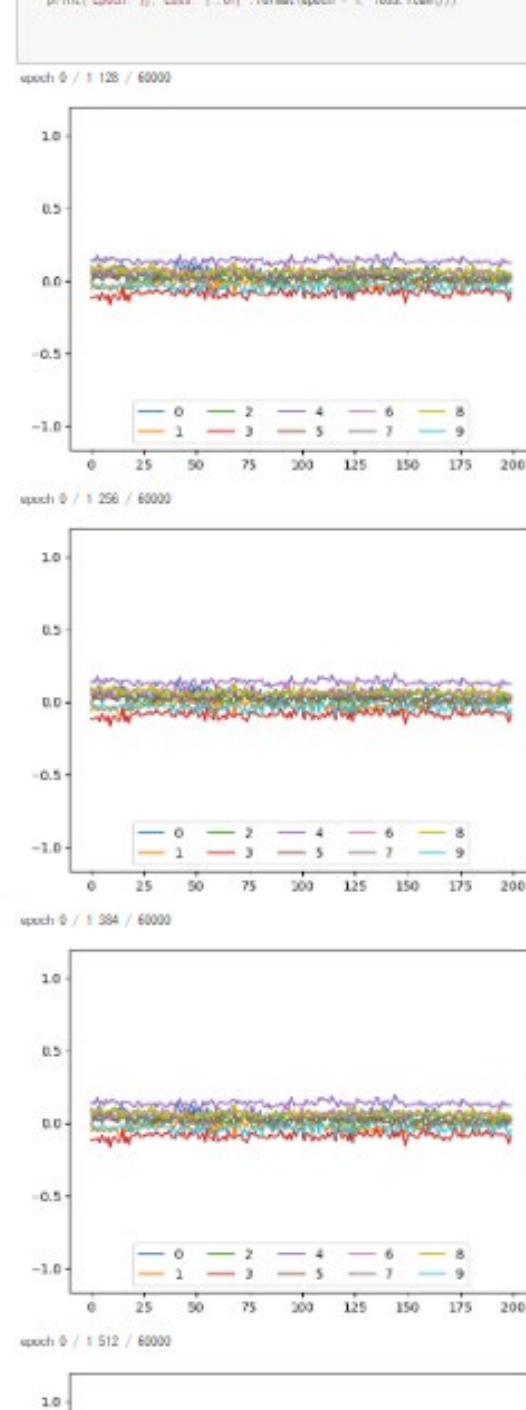
アニメーション表示

- `matplotlib.animation` が王道なのかもしれないが、プログラムの構造が大幅に変わるので機械学習の途中結果を表示するのには向いていない（気がする）
 - 動画ファイル（アニメーションgif等）の保存に対応
- 今回はそれ以外の手軽な方法で説明

パラパラアニメ

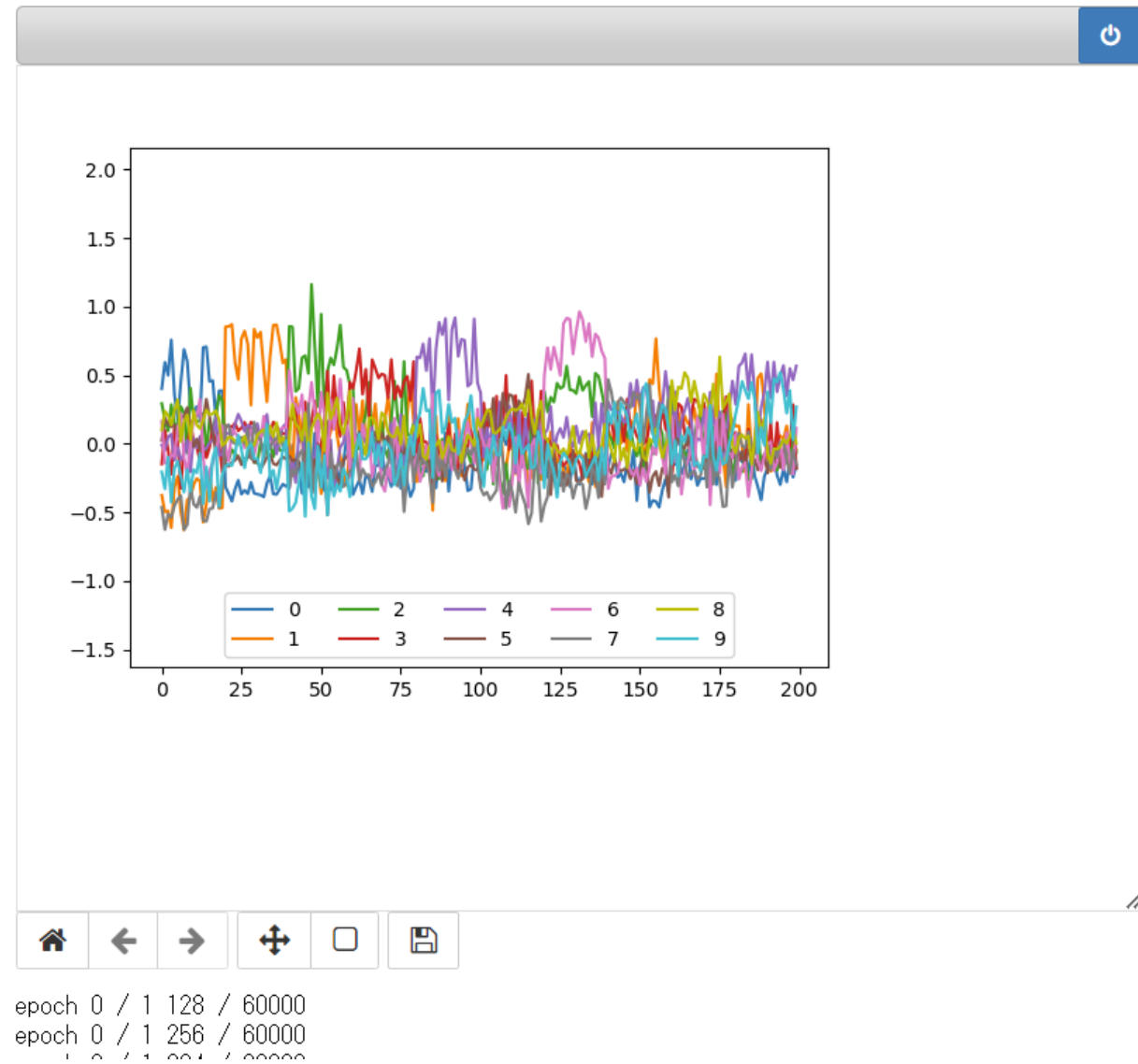
(%matplotlib notebook 等を書かずに)

- plt.show()
- を呼び出すと呼び出す度に新しい画像が表示される
 - **PyCharm**であれば別画面でスクロールしてくれるので気にならない
- 似たような画像が大量に表示されるので、ひたすら長くなってしまうが、
 - レポートに使うなら十分
 - スクロールで戻れるのは便利



notebookで動画

- %matplotlib notebook
- fig.canvas.draw() で更新される
 - ここでfigが必要になる
 - plt.show()では最後の1枚だけ表示される
- 右図はMNISTの学習途中
 - まだ学習中だが、10色の山が見えかかっている



説明用に省略
しているので
コピペして
はいけない

notebookで動画（初期化）

```
%matplotlib notebook
import numpy as np
from matplotlib import pyplot as plt
```

```
x=range(200) # x軸の配列
y=np.zeros((10,200)) # y軸の配列を10本分
fig,ax = plt.subplots() # 1つのサブグラフを設定しfigとaxを取得
ax2=ax.twinx() # y2軸（右の軸）を追加 ここでaxが必要
```

```
line=[0]*10 # 折れ線オブジェクトの保存用
for i in range(10):
```

```
    line[i],=ax.plot(x,y[i],label=str(i)) # ダミーを描いて折れ線オブジェクトlineの取得
line2,=ax2.plot(x,y[0],label="Acc",color="red") # 精度の折れ線オブジェクトline2の取得
```

説明用に省略
しているので
コピペして
はいけない

notebookで動画（学習と評価）

```
for epoch in range(num_epochs):  
    model.train()  
    for batch_idx, (data, target) in enumerate(train_loader):  
        #ここで学習  
        model.eval()  
        with torch.no_grad():  
            for n in range(10): # 0を20個、1を20個、、、の順で200個評価する  
                for data, target in test_loader:  
                    #ここで評価  
                    for i in range(10): # 0-9の評価値をすべて保存  
                        y[i][n*20+num]=output[0][i]  
cor=np.full(200,correct/total) # 正解率=精度
```

説明用に省略
しているので
コピペして
はいけない

notebookで動画（グラフ更新）

```
ymax=max(list(map(lambda x: max(x), y))) # y軸調整用に最大値を取得
ymin=min(list(map(lambda x: min(x), y))) # 最小値
for n in range(10):
    line[n].set_ydata(y[n]) # 10本の折れ線グラフを更新
ax.set_ylim(ymin-1,ymax+1) # y軸調整
line2.set_ydata(cor) # 正解率を水平線で表示
ax2.set_ylim(0,1) # 正解率は[0,1]
lines_1, labels_1 = ax.get_legend_handles_labels() # 左y軸
lines_2, labels_2 = ax2.get_legend_handles_labels() # 右y軸
lines = lines_1 + lines_2
labels = labels_1 + labels_2
ax.legend(lines, labels,loc="lower center",ncol=5) # 左右y軸を凡例に表示する
fig.canvas.draw() # グラフ更新
# plt.show() # PyCharmの場合はこちら
```

Jupyter Labの場合

- `%matplotlib notebook` はエラーになる
- `%matplotlib inline` はエラーにならないが
 - `fig.canvas.draw()` は表示されない
 - `plt.show()` や `display(fig)` はパラパラアニメ
- ChatGPTも一旦ファイルに保存しないと出来ないといていた

Jupyter Labでのアニメーション

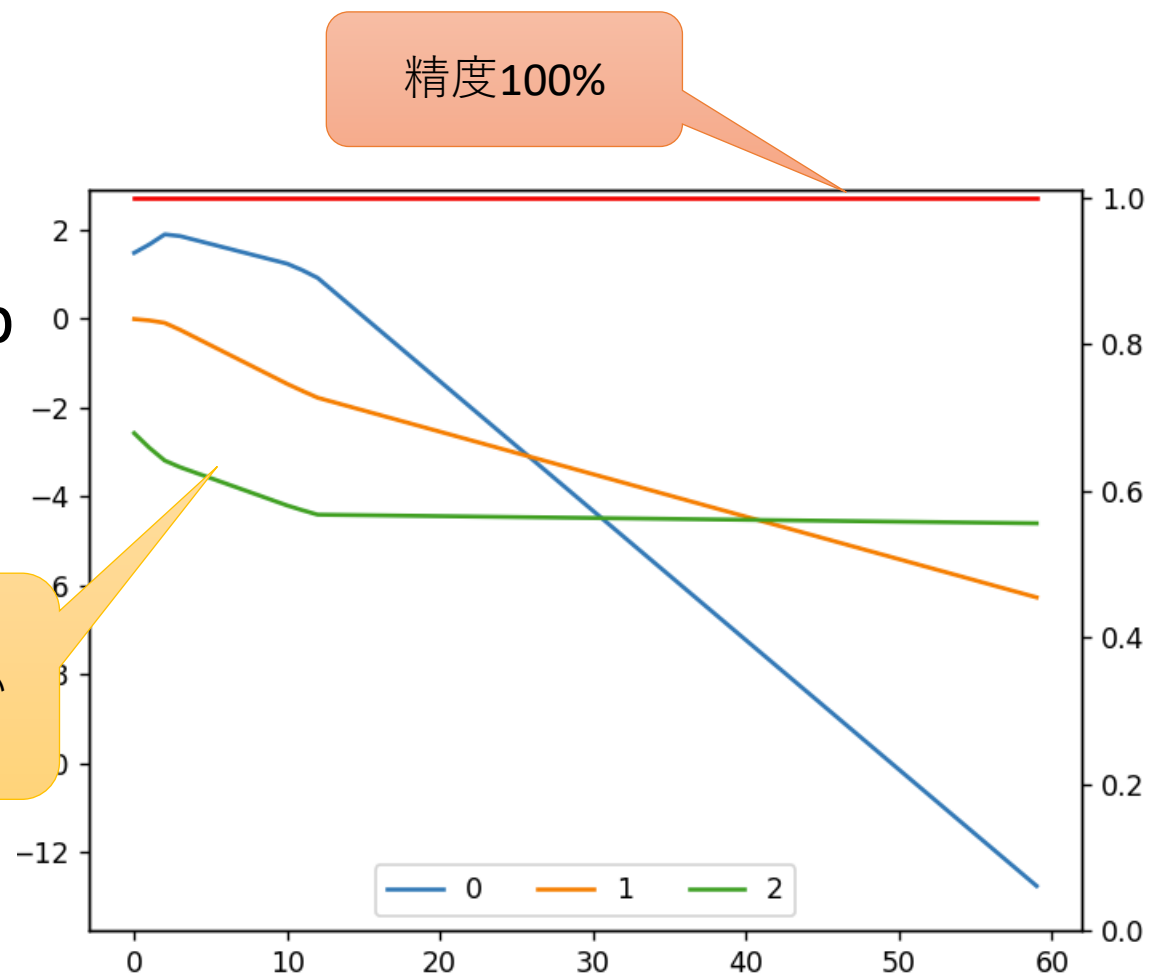
- 前ページから
 - `%matplotlib notebook` を削除して同じところに
 - `%matplotlib inline` を追加
 - `from IPython import display` を追加
 - `fig.canvas.draw()` を削除して同じところに
 - `display.clear_output(wait=True)` を追加（`wait=True`がないとチラつく）
 - `display.display(plt.gcf())` を追加
- `mnist-lab-ani.ipynb`を参照
 - 欠点としては`clear_output`でグラフだけでなく文字も消えてしまう

学習

```
for epoch in range(num_epochs):  
    for labels,data in train_dataloader:  
        optimizer.zero_grad()    # 勾配をゼロにリセット  
        outputs = model(data)    # 順伝播  
        loss = criterion(outputs, labels)    # 損失計算  
        # すべての計算をlossという一つの変数に集約し最小化する  
        loss.backward()    # lossを起点に逆伝播（勾配の計算）  
        optimizer.step()    # 勾配に応じて重みの更新
```

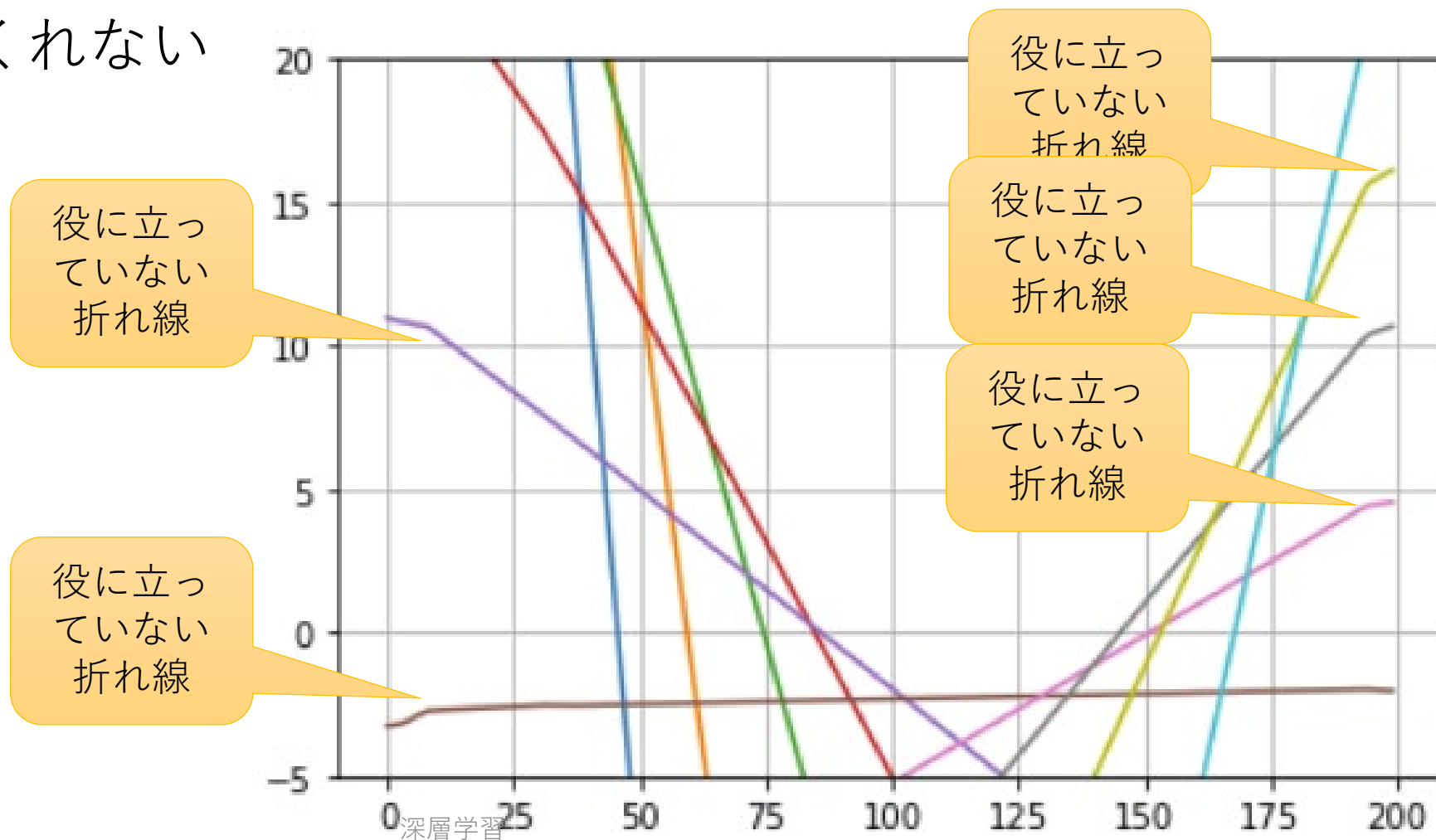
1次元3分類の例

- イヌ、サル、キジ
- csv-train-inusarukiji.ipynb
- sample-inusarukiji.csv



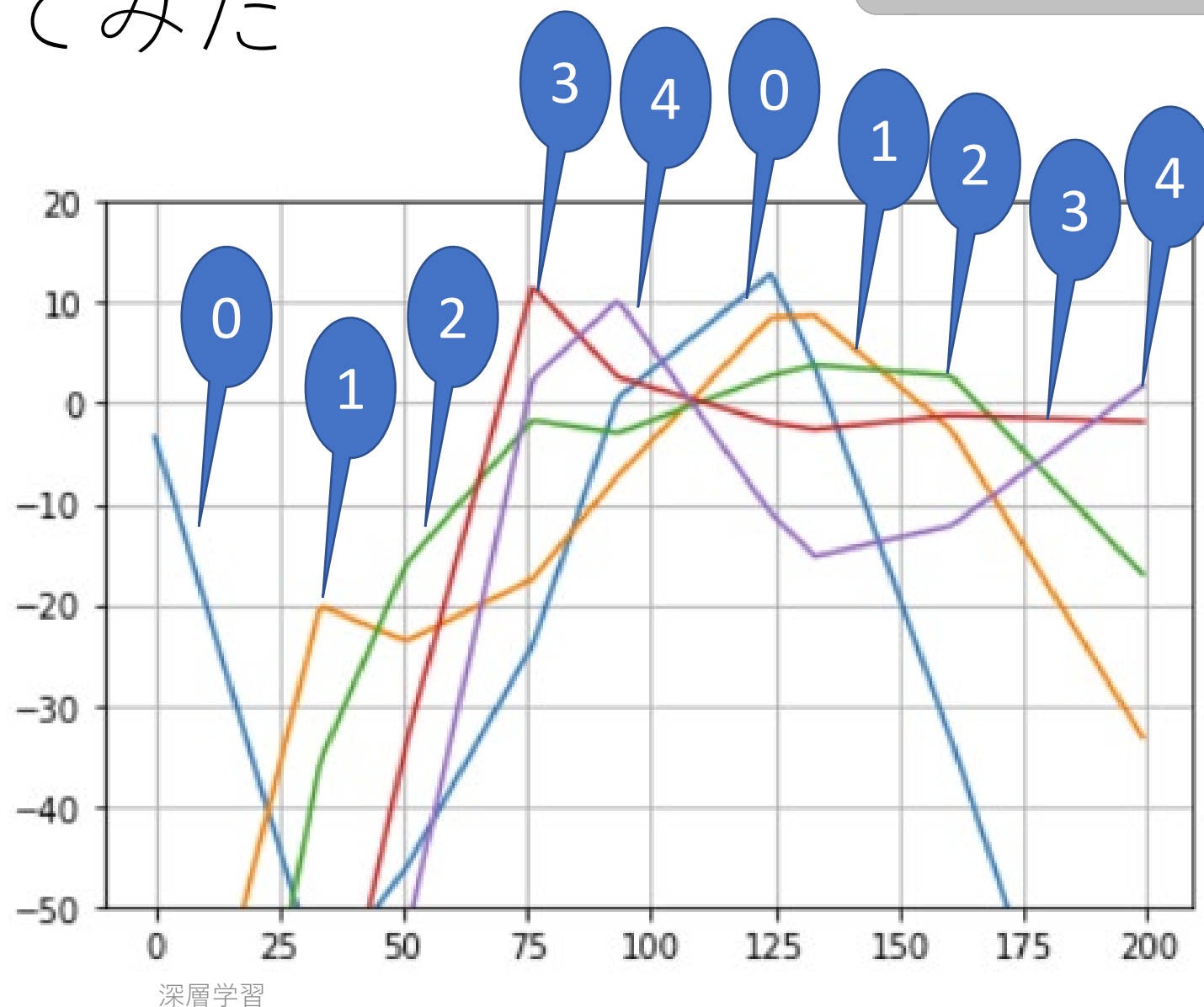
1次元10分類にしてみた

- 山型になってくれない
- 線形分離できるのが敗因



1次元5分類にしてみた

- 10分類の
- 0,1,2,3,4,5,6,7,8,9
を
- 0,1,2,3,4,0,1,2,3,4
とした
- 山型になった☺



☆課題3

- MNISTの学習において、全結合層と畳み込み層の違いが学習時間と精度に与える影響を調査する
 - 学習時間と精度のグラフは必須
 - アニメーションは遅いので時間計測時は消すこと
- つまり、
 - mnist.py をダウンロードして実行を確認してから
 - Conv2dを使わないように修正して比較する

```
def __init__(self):  
    super(Net, self).__init__()  
    self.conv1 = nn.Conv2d(1, 4, 3)  
    self.conv2 = nn.Conv2d(4, 4, 3)  
    self.flatten = nn.Flatten()  
    self.fc1 = nn.Linear(4*24*24, 128)  
    self.fc2 = nn.Linear(128, 10)
```

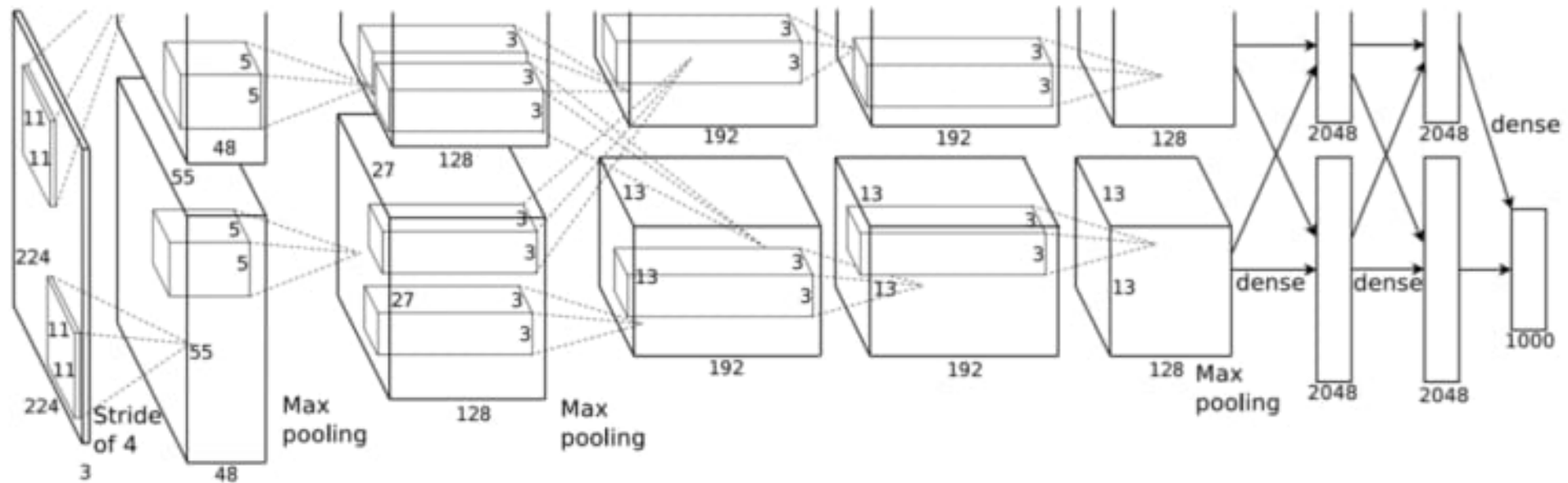
```
def forward(self, x):  
    x = self.conv1(x) #1x28x28->4x26x26  
    x = nn.ReLU()(x)  
    x = self.conv2(x) #4x26x26->4x24x24  
    x = nn.ReLU()(x)  
    x = self.flatten(x) #4x24x24->2304  
    x = self.fc1(x)  
    x = nn.ReLU()(x)  
    x = self.fc2(x)  
    return x
```

ファイル全体は
LMS参照

MNIST付属サンプル
を簡略化

畳み込み(Conv2d)とチャンネル

- これまでの説明では3x3のような畳み込みフィルタで説明していたが入出力チャンネルを考える必要がある

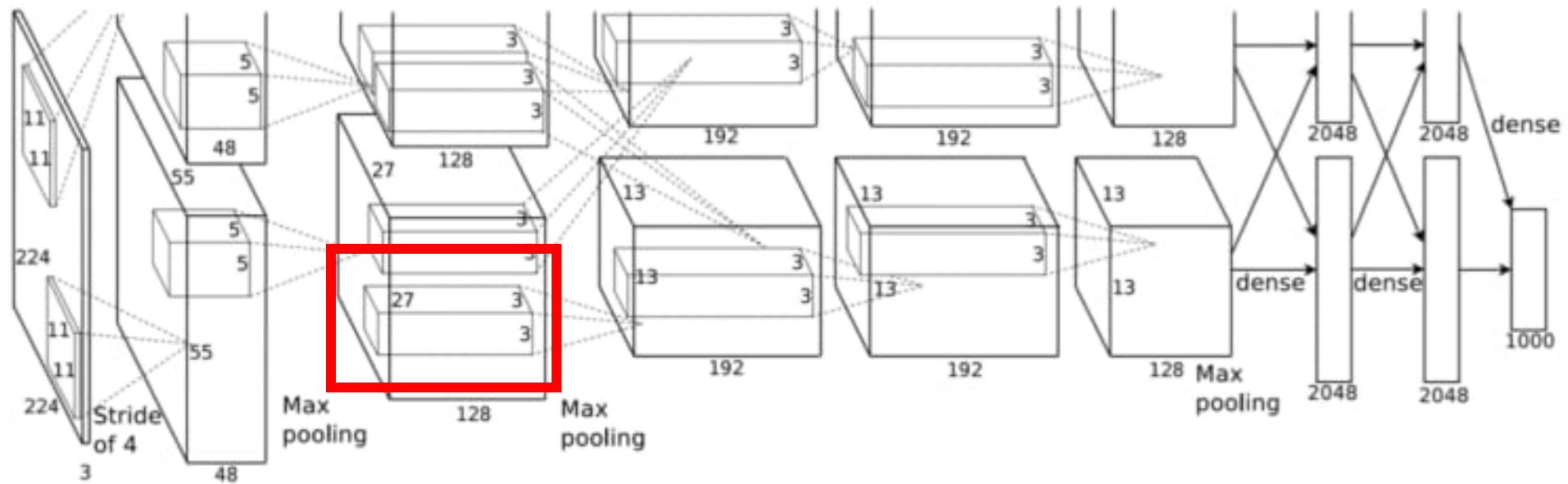


参考：Krizhevskyら "ImageNet Classification with Deep Convolutional Neural Networks"

深層学習

畳み込み(Conv2d)とチャンネル

- この例では128x3x3の畳み込みフィルタが192個
 - チャンネル方向には全結合となっている



参考：Krizhevsky's "ImageNet Classification with Deep Convolutional Neural Networks"

深層学習


```
self.conv1 = nn.Conv2d(1, 4, 3)
```

- `nn.Conv2d`(入力チャンネル数、出力チャンネル数、畳み込みフィルタのサイズ)
- 入力チャンネル数
 - 初段は白黒画像なら1、カラー画像なら3
 - 2段目以降は前段に従う
- 出力チャンネル数
 - 入力より大きくすることが多い
- 畳み込みフィルタのサイズ
 - 正方形のフィルタの1辺のサイズ

```
self.conv1 = nn.Conv2d(1, 4, 3)
```

- stride (省略)
 - 省略すると1
- padding (省略)
 - デフォルトはValidなので、出力サイズは「フィルタサイズ-1」だけ縦横とも小さくなる
- 入出力サイズ (CxHxW) やバッチサイズは明示的に指定しない点に注意
- MNISTの入力は1x28x28なので、
出力は $4 \times (28 - (3 - 1)) \times (28 - (3 - 1)) = 4 \times 26 \times 26$ となる

```
self.conv2 = nn.Conv2d(4, 4, 3)
```

- conv1の出力が入力となるので4x26x26
- 出力は $4 \times (26 - (3 - 1)) \times (26 - (3 - 1)) = 4 \times 24 \times 24$

self.flatten = nn.Flatten()

- 多次元の入力を1次元にする
 - ただし、バッチ処理の次元はそのまま
 - 結局は2次元になる
- `torch.flatten(x)` はバッチ処理部分も含めて完全に1次元にする
 - `torch.flatten(x,1)`は2次元目以降を1次元化するので、`nn.Flatten()`と同じ
- 今回の入力は4x24x24なので、出力は 2304

```
self.fc1 = nn.Linear(4*24*24, 128)
```

- 入力 $4*24*24=2304$ 、出力128の全結合層

```
self.fc2 = nn.Linear(128, 10)
```

- 入力128、出力10の全結合層
- 出力の10はMNISTの「0」から「9」の10種類に対応
- この後ろにsoftmaxを付けている例もあるが、CrossEntropyLossにsoftmaxが含まれるので不要
 - 2回softmaxしても動作はするので、無駄に気がつきにくい

Conv2dのパラメータ数

- Conv2d: 1-1
 - $1 \times 3 \times 3 \times 4 + 4 = 40$
- Conv2d: 1-2
 - $4 \times 3 \times 3 \times 4 + 4 = 148$
- 入力ch x フィルタ1辺 x フィルタ1辺 x 出力ch + バイアス
 - バイアスは出力チャンネル当たり1つ

```
self.conv1 = nn.Conv2d(1, 4, 3)
```

```
self.conv2 = nn.Conv2d(4, 4, 3)
```

```
x = self.conv1(x) #1x28x28->4x26x26
```

```
x = self.conv2(x) #4x26x26->4x24x24
```

Layer (type:depth-idx)	Output Shape	Param #
=====	=====	=====
Net	[1, 10]	--
└─Conv2d: 1-1	[1, 4, 26, 26]	40
└─Conv2d: 1-2	[1, 4, 24, 24]	148
└─Flatten: 1-3	[1, 2304]	--
└─Linear: 1-4	[1, 128]	295,040
└─Linear: 1-5	[1, 10]	1,290

Linearのパラメータ数

- Linear: 1-4
 - $2304 \times 128 + 128 = 295040$
- Linear: 1-5
 - $128 \times 10 + 10 = 1290$
- 入力ch x 出力ch + バイアス
 - バイアスは出力チャンネル当たり1つ

```
self.fc1 = nn.Linear(4*24*24, 128)
```

```
self.fc2 = nn.Linear(128, 10)
```

```
x = self.fc1(x)
```

```
x = self.fc2(x)
```

Layer (type:depth-idx)	Output Shape	Param #
=====		
Net	[1, 10]	--
└─Conv2d: 1-1	[1, 4, 26, 26]	40
└─Conv2d: 1-2	[1, 4, 24, 24]	148
└─Flatten: 1-3	[1, 2304]	--
└─Linear: 1-4	[1, 128]	295,040
└─Linear: 1-5	[1, 10]	1,290

深層学習

☆課題3

締切

2025/5/15 23:59

- MNISTの学習において、全結合層と畳み込み層の違いが学習時間と精度に与える影響を調査する
 - まず何をどう変更したのかを書く
 - 学習時間と精度のグラフは必須
 - アニメーションは遅いので時間計測時は消すこと
- つまり、
 - mnist.py をダウンロードして実行を確認してから
 - Conv2dを使わないように修正して比較する

MNISTは簡単なので、「どこまでモデルを小さくして精度が維持できるか、そのためには全結合層と畳み込み層のどちらが良いか」と考えると良い

```
def __init__(self):
```

```
    super(MyModel, self).__init__()
```

```
    self.conv1 = nn.Conv2d(1, 4, 3, 1)
```

```
    self.conv2 = nn.Conv2d(4, 4, 3, 1)
```

```
    self.flatten = nn.Flatten()
```

```
    self.fc1 = nn.Linear(ここを計算, 128)
```

```
    self.fc2 = nn.Linear(128, 10)
```

convを消す

flattenが1番目になる

fcを増やした方が良い
要素数も？

```
def forward(self, x):
```

```
    x = self.conv1(x) #1x28x28->4x26x26
```

```
    x = nn.ReLU()(x)
```

```
    x = self.conv2(x) #4x26x26->4x24x24
```

```
    x = nn.ReLU()(x)
```

```
    x = self.flatten(x)
```

```
    x = self.fc1(x)
```

```
    x = nn.ReLU()(x)
```

```
    x = self.fc2(x)
```

```
    return x
```

想定されるレポートの例

- 畳み込み層を単純に消しただけではXXXであるため、fc1のLinearの入力要素数をXXXに修正した
- 修正後に実行したところ、実行時間はXXXとなり、精度はXXXとなった
- さらに実行時間（または精度）を短くする（または高くする）ために、XXXをXXXと修正した
- その結果、実行時間はXXXとなり、精度はXXXとなったため、今回の課題では、XXXであることがわかった

（去年の）課題3のレポート総評

- 箇条書きのような文章は卒業しましょう
- （特にグラフ内の）文字化け
- 口語
 - 畳み込みは1層でやった。
- 唐突に文末が変わる
 - 「である」と「です・ます」の混在
 - たぶん生成AIやネットのコピペ
- モデルを書かない
 - 変更した場所すら書かない
 - 「1層を2層にした」ではわからない

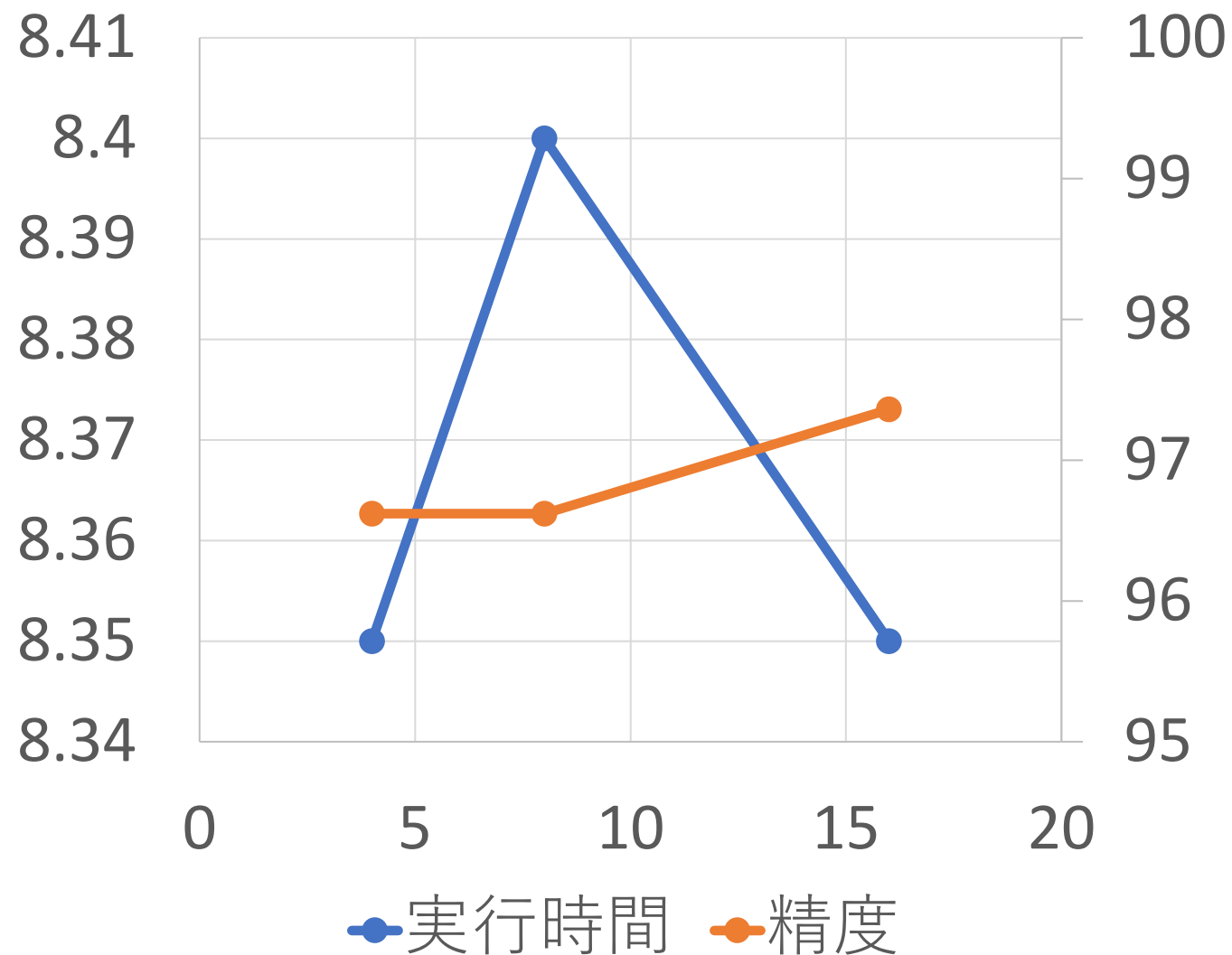
(去年の) 課題3のレポート総評

- グラフの軸にラベルがない
 - どっちが実行時間なのかわからない
 - 「数字見ればわかる」はナシ
 - 単位も必須
 - 精度や行列サイズのように無次元もあるが
 - 「グラフタイトル」のままなのは論外
- 「比較的よい」
 - 何と比較したのかが書かれていない
- 複数の結果の比較
 - 比べることを考えてグラフを作ること
 - もっと丁寧に書いてくれないと何を意図しているのか読み取れない
 - 「図XのN=10と図YのN=10の実行時間を比べると・・・がわかる」

良くない考察例

- チャンネル数を増やすと精度は徐々に向上した
- 一方実行時間は8から16で減少した。理由は今後の課題である

畳み込み層のチャンネル数



☆課題4（予告）

- 簡単なニューラルネットワークを使って、パラメータの違いが学習時間と精度に与える影響を調査する
 - パラメータの例
 - ノード数、層数、活性化関数、epoch数、損失関数
 - 基本的には全結合層と畳み込み層が良いが、それ以外を利用しても良い
 - パラメータと学習時間と精度のグラフは必須
 - `sklearn.datasets`や`torchvision.datasets`のような「ダウンロードするだけのデータセット」以外を使うと+1点（つまり9点満点の上限8点）

- OMP: Error #15: Initializing libiomp5md.dll, but found libiomp5md.dll already initialized.

というエラーが出たら以下をファイルの一番上に追加

```
import os
```

```
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```


(学習データの参考) json

- JavaScript Object Notation

- 元はJavaScript用だが、現在では言語にかかわらず使われる

- 例

- <https://www.jma.go.jp/bosai/forecast/data/forecast/270000.json>
 - 大阪の天気

```
1 [{"publishingOffice": "大阪管区气象台",
2   "reportDatetime": "2023-05-01T11:00:00+09:00",
3   "timeSeries": [
4     {
5 >       "timeDefines": [...
9     ],
10    "areas": [
11      {
12 >        "area": {...
15      },
16 >        "weatherCodes": [...
20      ],
21      "weathers": [
22        "晴れ 夕方 から く墨属学所により 昼過ぎ から 夜のはじめ頃 雨 で 雷を伴う",
23        "晴れ",
```

jsonファイルの読み込み例

入力 [1]: `import json`

入力 [2]: `fp = open("270000.json", "r")
data = json.load(fp)`

¥u3000は
全角空白

入力 [3]: `data[0]['timeSeries'][0]['areas'][0]`

出力[3]: `{'area': {'name': '大阪府', 'code': '270000'},
'weatherCodes': ['111', '100', '101'],
'weathers': ['晴れ¥u3000夕方¥u3000から¥u3000くもり¥u3000所により¥u3000昼過ぎ¥u3000から¥u3000夜のはじめ頃¥u3000雨¥u3000で¥u3000雷を伴う',
'晴れ',
'晴れ¥u3000時々¥u3000くもり'],
'winds': ['南西の風¥u3000後¥u3000西の風¥u3000やや強く', '北の風', '西の風'],
'waves': ['0. 5メートル¥u3000後¥u30001メートル', '0. 5メートル', '0. 5メートル']}`

3つずつあるのはたぶん
今日、明日、明後日

入力 [4]: `data[0]['timeSeries'][0]['areas'][0]['weathers'][0]`

出力[4]: `'晴れ¥u3000夕方¥u3000から¥u3000くもり¥u3000所により¥u3000昼過ぎ¥u3000から¥u3000夜のはじめ頃¥u3000雨¥u3000で¥u3000雷を伴う'`