

深層実習 第1,2回

中田尚

- 席は自由ですが、後ろの席には
コンセントがありません
- 飲食禁止

本講義の狙い

- 実際の**AI**システムを理解する
- 深層学習を利用出来るようになる
- 深層学習を自分で作れるようになる
- 自分のアイデアを実現できるようになる
- 自分の夢実現の手段として**AI**を活用できるようになる

他の科目との関係

- 1年次の科目

- 情報数学、コンピュータシステム、英語
- 線形代数、解析学
- ソフトウェア開発基礎、組込みプログラミング
- Pythonプログラミング

- 2年次

- 知的システム、人工知能数学
- 機械学習、画像・音声認識
- 臨地実務実習 I

他の科目との関係

- 3年次

- **深層学習**、自然言語処理
- 人工知能システム社会応用、人工知能サービスビジネス応用
- 地域共生ソリューション開発
- 臨地実務実習Ⅱ

- 4年次

- 地域共生ソリューション開発
- 臨地実務実習Ⅲ
- 卒業研究制作

授業の進め方

- 合計30コマ 金曜日4,5限2コマ
 - 191室
- ノートPC、デスクトップPC
- AIサーバも使うかも

自習の参考用

月	1	3年AI 林越	水	1	3年口術 中村	金	1	
	2	3年AI 林越		2	3年口術 中村		2	
	3			3			3	
	4	2年AI 中田吉田		4			4	3年AI 中田
	5	2年AI 中田吉田		5			5	3年AI 中田
	6			6			6	
火	1		木	1		1限 9:30～11:00 2限 11:10～12:40 3限 12:50～14:20 4限 14:30～16:00 5限 16:10～17:40 6限 17:50～19:20		
	2			2				
	3			3	3年AI 林越			
	4			4	2年AI 中田吉田			
	5			5	3年口術 中村			
	6			6	3年口術 中村			

シラバス

- 第1回：オリエンテーション
- 第2～3回：人工知能、機械学習、深層学習の技術
- 第4～5回：深層学習のフレームワーク
- 第6～7回：深層学習の例題について
- 第8～9回：深層学習の例題の実行準備（1）
- 第10～11回：（2）データの収集 第12～13回：（3）クレンジング
- 第14～15回：（4）保存形式 第16～17回：（5）データ構造
- 第18～20回：深層学習の例題の実行（1）
- 第21～24回：深層学習の例題の実行（2）ハイパーパラメータ
- 第25～26回：結果のまとめ 第27～28回：結果の考察
- 第29～30回：まとめの発表

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

レポートの書き方

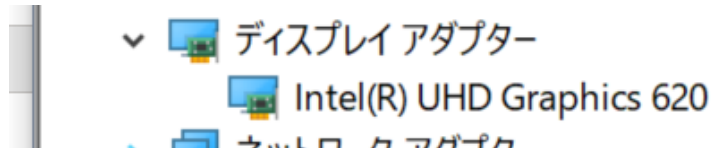
- 卒業論文に向けて体裁を重視します
 - 内容6点＋体裁2点
- 原則としてWordかPDF
 - 文字サイズ等は常識の範囲で
 - TeX（で作成したPDF）でも良い
- 手書きの文章は不可
- スクリーンショットを挿入するのはOK
- 手書きの絵を挿入するのはOK

ChatGPTのレポートでの利用

- 自然言語処理と同じ
- 生成AIを利用して、「自分の力 + AIの力」の合計値を高める

実習環境

- ノートPCまたはデスクトップPC
- できれば画像の学習処理ではGPUを使いたい
 - 大学推奨ノートPCにはGPUが搭載されている
 - なるべくノートPCでの実行方法も説明するが、大学推奨PC以外は対応が難しいかもしれない
 - NVIDIAのGPUが搭載されていないと対応不可
 - ↓は未対応の例（デバイスマネージャ）



- 少なくともデスクトップPCでは実習ができる（大学でしかできない）
 - または、ノートPCのCPUを使う（遅い）

デスクトップPCとファイルサーバ

- ログイン
 - IDは自分の学籍番号 (OK2XXXXX)
 - パスワードは B生年月日 (OK21XXXXXXの人)
 - パスワードは #B生年月日 (OK22,OK23,OK24の人)
- ~~再起動するとデスクトップPC上のデータはすべて消えます。~~
 - ~~PCの不調で応答がなくなった場合も消えるので注意~~
 - **今のところ消えませんがメンテナンスの都合で消えるかも**
 - ファイルサーバまたはUSBメモリ等を使う
- ファイルサーバ
 - ¥¥ostdfs61¥IT¥ok2x (または¥¥10.72.162.11¥IT¥ok2x)
 - の中に自分の学籍番号のフォルダが見えるはず

前提知識

- Pythonプログラミング
- 実行方法は自由（Visual StudioやPyCharm等）
 - `python test1.py` のように実行することを想定して説明
 - Jupyter Notebookを使いたい人は使っても良いが、AIサーバでは使えない

参考書

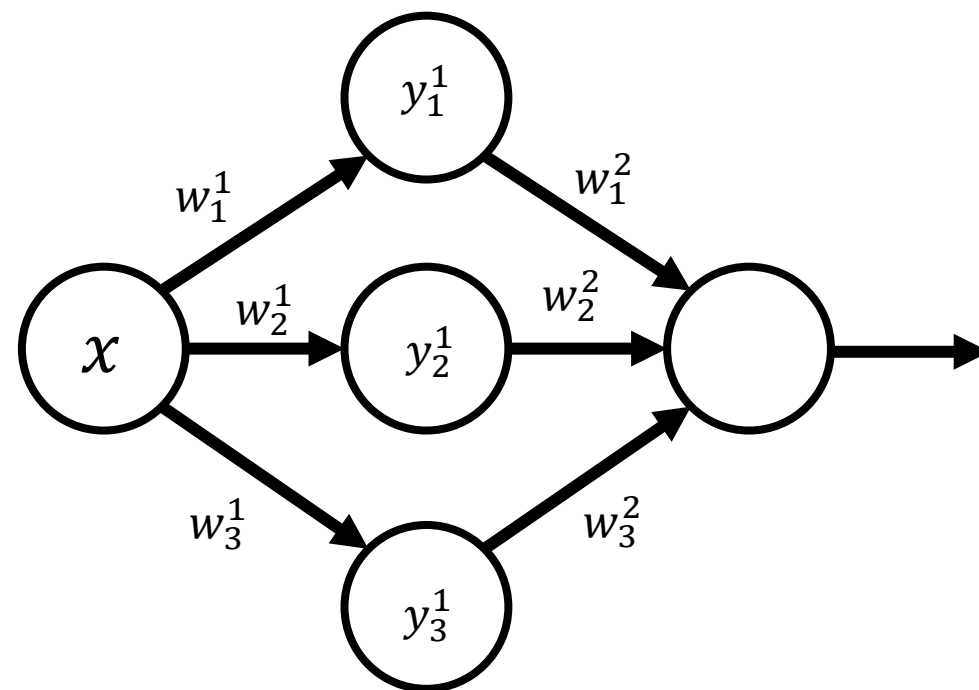
- 「もっと詳しく知りたい人」向け
- 画像処理
 - ゼロから作るDeep Learning 3740円
 - ちょっと細かすぎるかもしれない
 - ゼロから作るDeep Learning ④ —強化学習編
 - メルカリにも多数あり（2000-3000円程度）

深層学習とは

- 人工知能 = AI (Artificial Intelligence)
 - 知能を実現するすべての技術
- 機械学習 = 人工知能の一部
 - 機械 (コンピュータ) がデータを見て学習する
- 深層学習 = 機械学習の一部
 - 層が深い (定義上は4層以上だが、数十層以上となるのが一般的)
 - 2層は単純パーセプトロン
 - 3層は多層パーセプトロン と呼ばれる
 - 入力層 + 複数の中間層 + 出力層

多層パーセプトロン (MLP)

- 入力 \mathbf{x} と重み \mathbf{w} との乗算 (積)
 - 積の総和
 - 活性化関数
- の組み合わせ

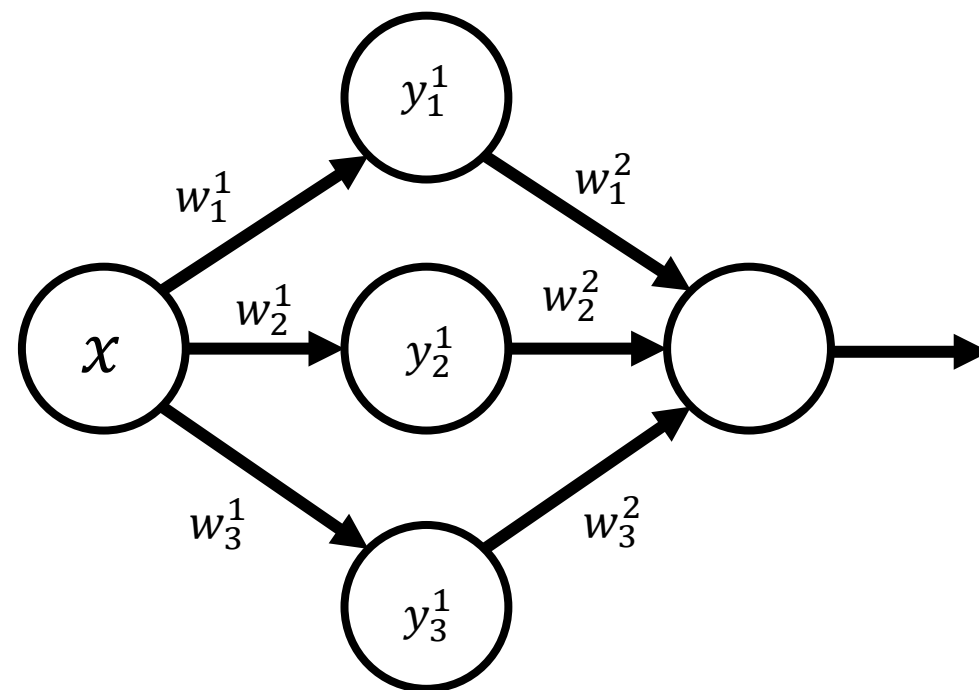


普遍性定理（万能近似定理）の話

- 別紙

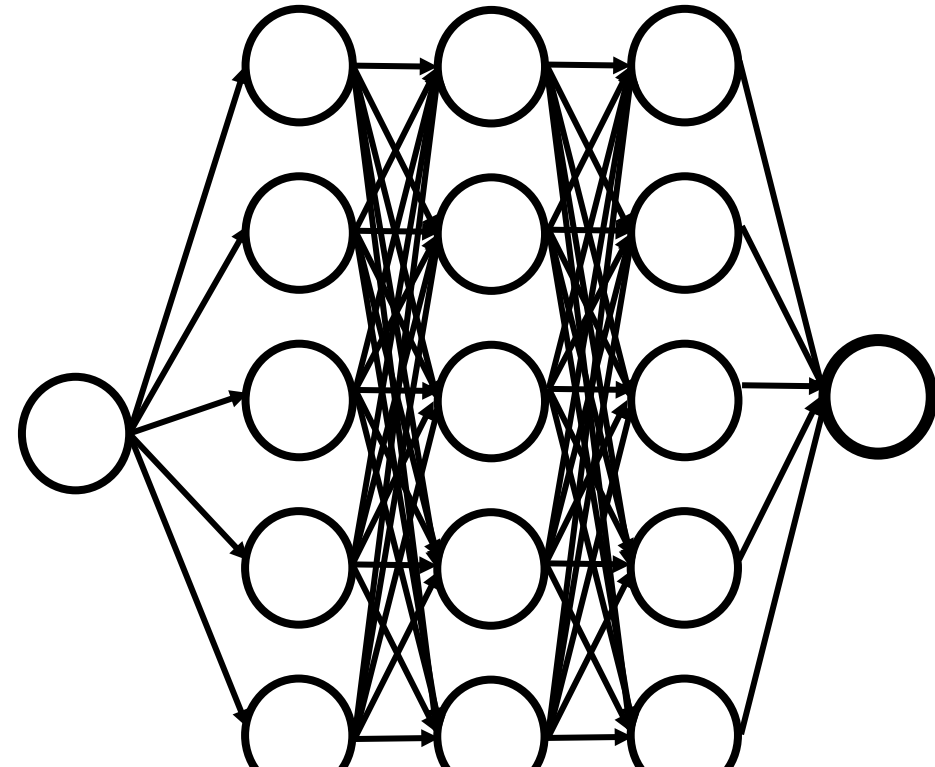
多層パーセプトロン (MLP)

- 入力と出力
- $y=f(x)$
- 関数
- あらゆる関数を表現できる
 - 普遍性定理
 - 2層ではできない



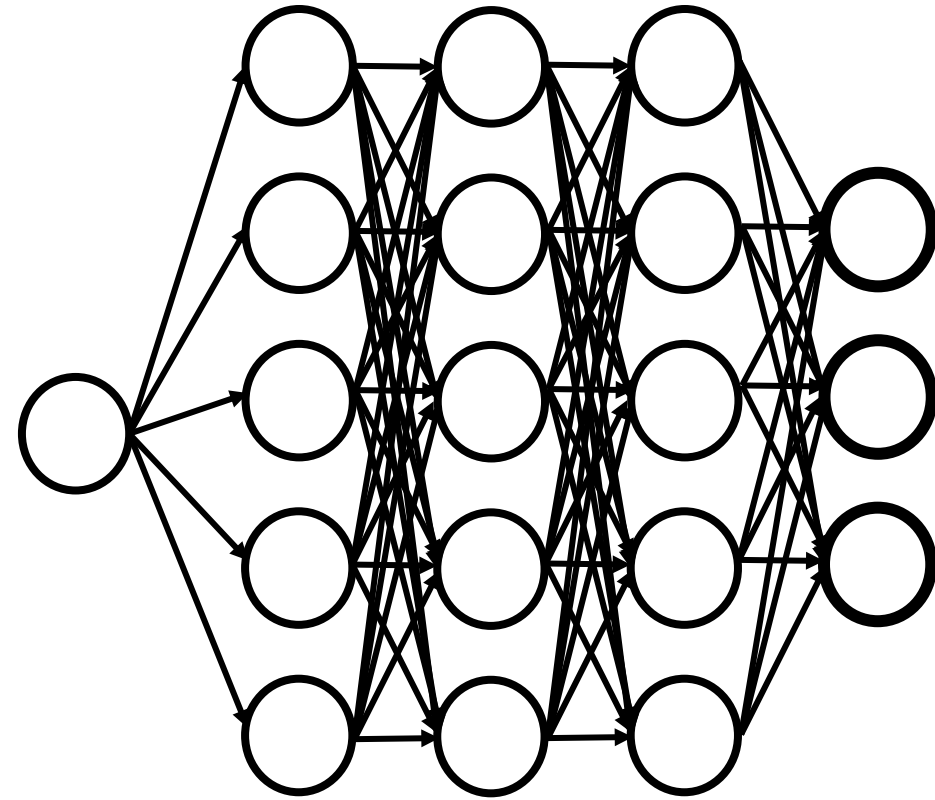
多層化

- 普遍性定理は「有限の範囲でどこまでも中間層を増やせる」という条件の下で成り立つ
 - 現実的には中間層1層のノード数を増やし続けることは現実的ではない
- 多層にしよう



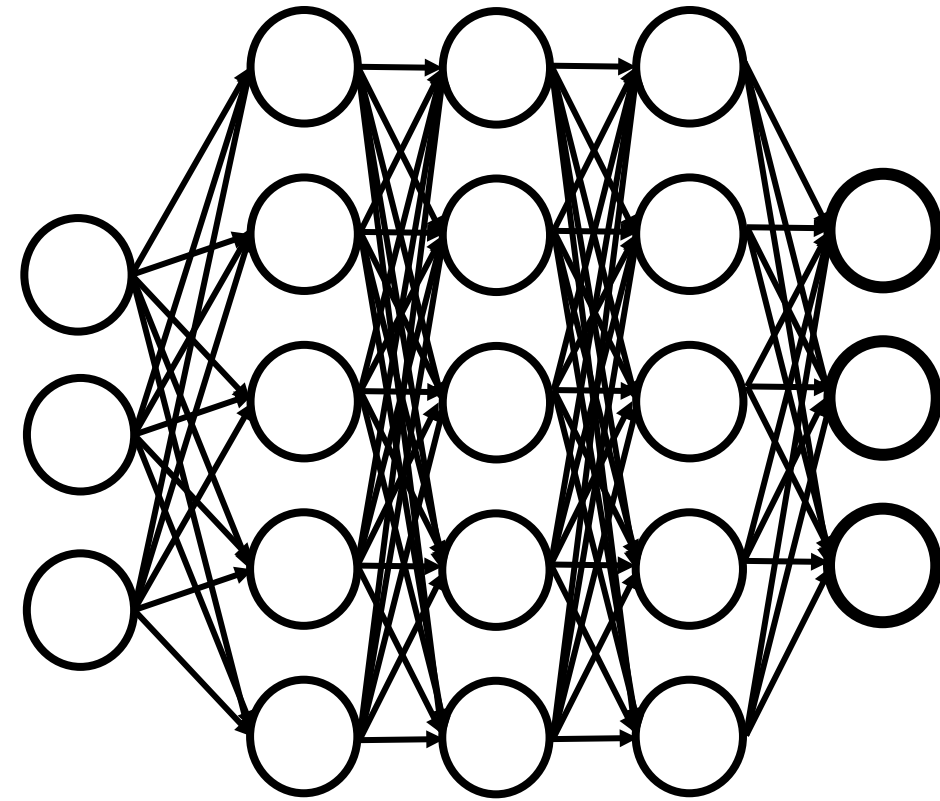
出力を増やす

- 例
 - 認識可能な物体の種類を増やす
 - 2次元、3次元空間の結果を出力する



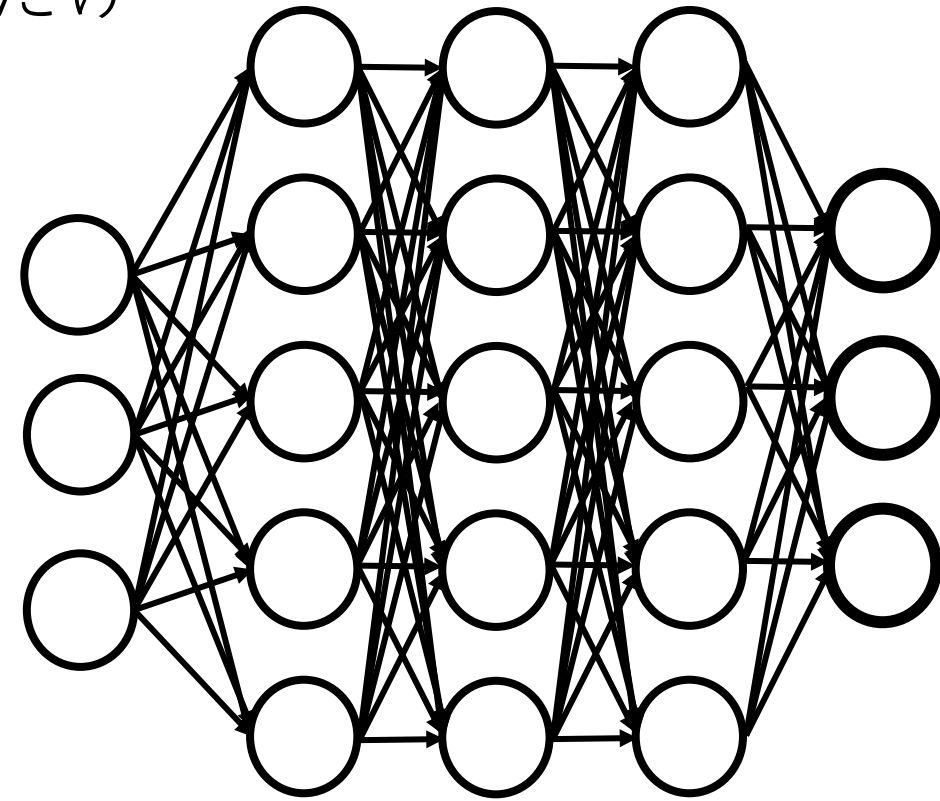
入力を増やす

- 例
 - 画像、音声をそのまま入力とする



一般的な深層ニューラルネットワーク

- 後はそれぞれの「数」を適切に設定するだけ
- 入力、中間層、出力 = ベクトル
- 矢印の重み（乗数） = 行列
- これ以外に加算（バイアス）があるが図では省略されることが多い



実際の計算とその実装

余談

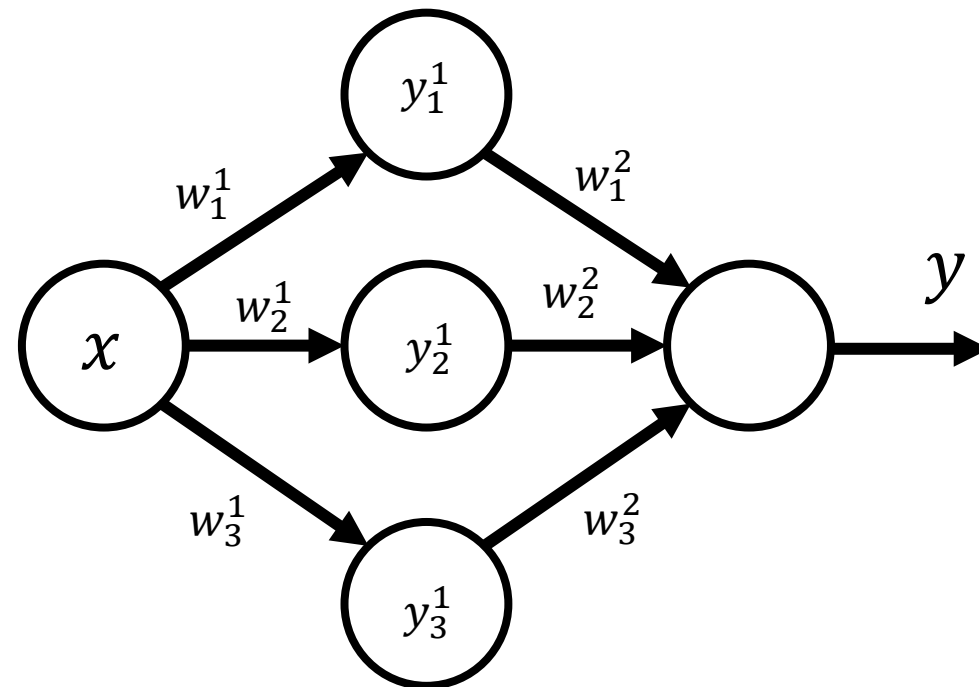
- Scalar
 - スカラー スケーラー
- Vector
 - ベクトル ヴェクター
- Tensor
 - テンソル テンサー
- TensorFlowをテンサーフローと言うが、Scalarをスカラーと言う人が多くてモヤる

多層パーセプトロンの計算

- $y_1^1 = w_1^1 x$
- $y_2^1 = w_2^1 x$
- $y_3^1 = w_3^1 x$
- $\mathbf{y}^1 = \mathbf{w}^1 x$
- ベクトル = ベクトル・スカラ
- 1次元配列 = 1次元配列・変数
- バイアスと活性化関数は省略

凡例

小文字の細字はスカラ (変数)
小文字の太字はベクトル (1次元配列)

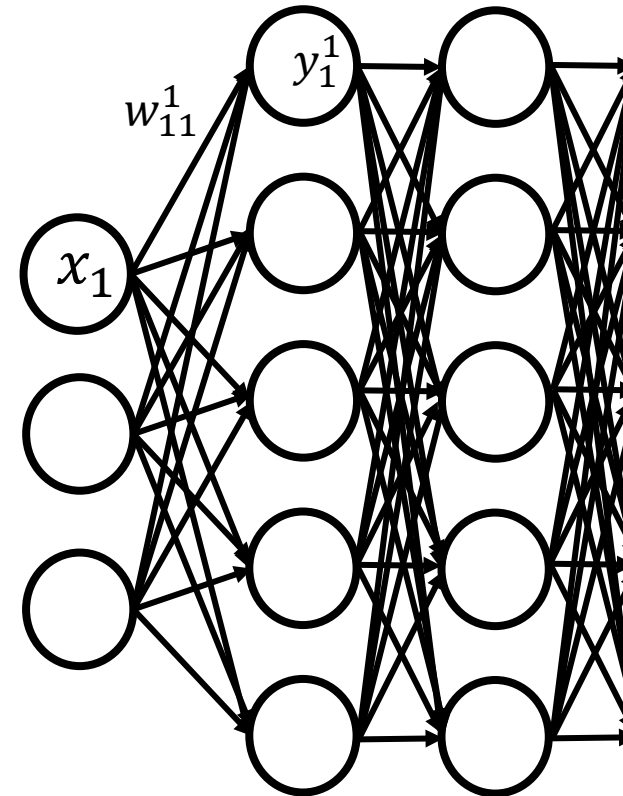


入力が複数の場合

- $y_1^1 = w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3$
- $y_2^1 = w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3$
- $y_3^1 = w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3$
- $\mathbf{y}^1 = \mathbf{W}^1 \mathbf{x}$
- ベクトル = 行列 \cdot ベクトル
- 1次元配列 = 2次元配列 \cdot 1次元配列
- バイアスと活性化関数は省略

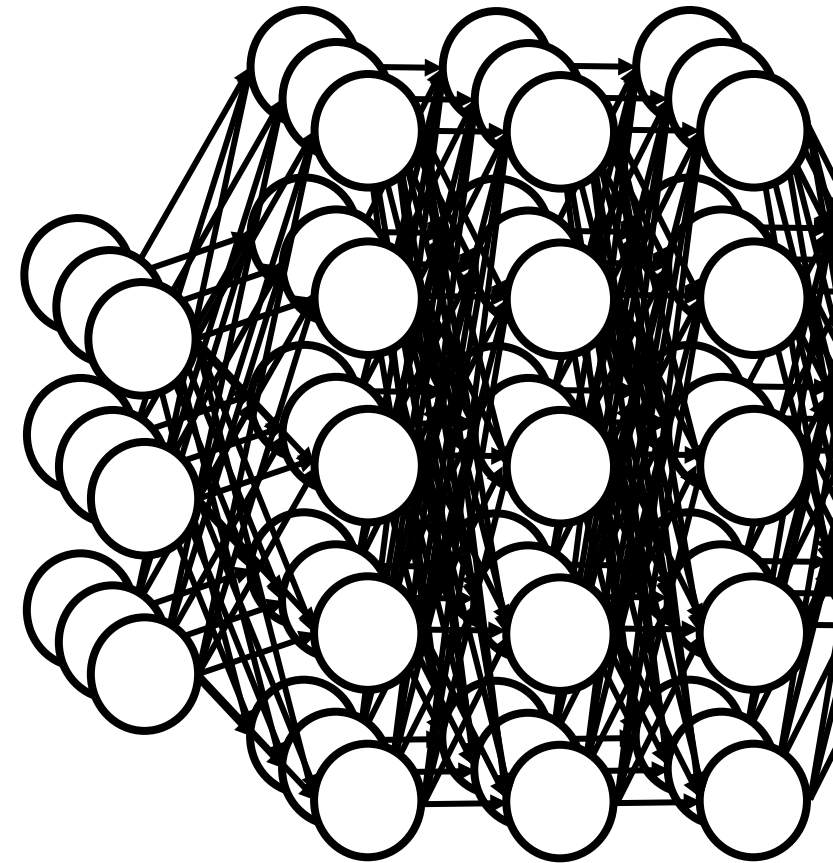
凡例

小文字の細字はスカラー (変数)
小文字の太字はベクトル (1次元配列)
大文字の太字は行列 (2次元配列)



入力、中間層、出力が2次元の場合

- $y_{11}^1 = w_{11,11}^1 x_{11} + w_{11,12}^1 x_{12} + w_{11,13}^1 x_{13} + w_{11,21}^1 x_{21} + w_{11,22}^1 x_{22} + w_{11,23}^1 x_{23} + w_{11,31}^1 x_{31} + w_{11,32}^1 x_{32} + w_{11,33}^1 x_{33}$
- $Y^1 = W^1 X$
- 行列 = (4階の) テンソル・行列
- 2次元配列 = 4次元配列・2次元配列
 - バイアスと活性化関数は省略
- **あまりこの構成は使われない**
 - なぜ？
 - レポート課題にしようかとも思いましたが、



ChatGPTに聞いてみた

NA

ニューラルネットワークの全結合層が2次元で使われることが少ない理由を説明してください。



2次元データに対して全結合層が直接使われることが少ない理由は、以下の3点です。

1. 空間情報の喪失
2. パラメータ数の増加
3. 畳み込み層の優位性

2次元データを扱うニューラルネットワークでは、通常、畳み込み層やプーリング層で特徴抽出を行い、その後全結合層を用いて最終的な予測を行います。

NA

表にしてまとめて



理由	説明
空間情報の喪失	全結合層は空間情報を考慮せず、空間構造や関連性が失われる。
パラメータ数の増加	全結合層はパラメータ数が多く、計算コストが増加し過学習のリスクが高まる。
畳み込み層の優位性	畳み込み層は空間情報を維持し、計算効率が良く、2次元データに適している。

だからどうなのか？
(やや惜しい)

2次元かどうかは
関係ない

畳み込みの話は
聞いていない

2025年度版

ChatGPT 4.5 ▾

たとえ話（お弁当の例）

全結合層とは、たとえば**「お弁当箱の中身を全部ぐちゃぐちゃに混ぜて食べる」**ような処理だと思ってください。

- 2次元データ（画像など）をそのまま使うということは、お弁当箱の中でおかずやご飯が「どこに入っているか』『どの順番で食べるか』をちゃんと分けているような状態です。
- しかし、全結合層はそれを完全に混ぜてしまいます。混ぜてしまうなら、最初からきれいに並べる必要はありませんよね？

そのため、「最初から1次元（並び順だけがあるデータ）にして混ぜやすくしておこう」と考えるわけです。

1次元の方が
混ぜやすい？？

実用上の問題

- 2次元と1次元の全結合層は同じなので、2次元のデータを1次元だと思って1次元の全結合層を使えば良い

どちらも全結合層のこと

- `pytorch`の`nn.Linear` や `tf.keras.layers.Dense`
 - 2次元（以上）のデータが与えられた場合は、1次元の全結合層が複数組あるとして扱う
 - **そのため、入力を1次元にしなければ全結合層が実行できない**
 - 2次元データを1次元データに変換するコストは無視できるほど小さいので実用上の問題はない
 - というところまでこの授業で理解してほしい

Pythonプログラミング

- 2次元配列の作り方
- 1次元配列の作り方

```
In [2]: ▶ x = [0]*5  
print(x)  
x[3] = 1  
print(x)
```

```
[0, 0, 0, 0, 0]  
[0, 0, 0, 1, 0]
```

実はこの方法は
推奨されない

Pythonプログラミング

- 2次元配列の作り方

```
In [4]: ▶ x = [[0]*5] * 3  
print(x)  
x[1][3] = 1  
print(x)
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]  
[[0, 0, 0, 1, 0], [0, 0, 0, 1, 0], [0, 0, 0, 1, 0]]
```

1要素しか変更して
いないつもりなのに
3つも変更されている

- 1次元配列の作り方

```
In [2]: ▶ x = [0]*5  
print(x)  
x[3] = 1  
print(x)
```

```
[0, 0, 0, 0, 0]  
[0, 0, 0, 1, 0]
```



Pythonで2次元配列を0で初期化する方法は、次のようにリスト内包表記を使って行うことができます。

python

Copy code

```
rows = 5    # 行数  
cols = 4    # 列数
```

```
# 2次元配列を0で初期化
```

```
array_2d = [[0 for _ in range(cols)] for _ in range(rows)]
```

```
print(array_2d)
```

本来は変数が入るが、
使わないことを明示的に
示すために _ としている

行列積の計算 (matmul : Matrix Multiply)

- python (の基本機能)
- numpy
- cupy
- pytorch

セットアップ方法（Anacondaの場合）

PyTorchがAnacondaの
サポートを終了したので
使うと不幸になります

<https://forest.watch.impress.co.jp/docs/news/1640199.html>

セットアップ方法（venvの場合）

- （Anacondaではない）Pythonがインストール済みとする
- > python -m venv adev
- > .¥adev¥Scripts¥activate
- (adev) > pip install torch torchvision torchaudio --index-url <https://download.pytorch.org/whl/cu118>
- (adev) > pip install cupy-cuda11x
- 2回目以降は毎回
¥adev¥Scripts¥activateだけ必要

デスクトップPCは
（昨年度から）この設定

デスクトップPCのvenvの利用方法

1. 2年生用にインストール済みのものを使う
 - C:¥Program Files¥Python311¥aidev 以下にある
 - PyCharmの設定は知っているという前提です
2. 前ページを参考に新たにインストールする
 - 席を移動すると再インストールが必要なので注意

練習1 (test1.py)

- ベクトル (1次元配列) とスカラ (変数) の積

```
cols = 4 # 列数
```

```
array_1d = [2 for _ in range(cols)] # 1次元配列を2で初期化
```

```
a = 3.0 # スカラ変数
```

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

```
    array_1d[i] = array_1d[i] * a
```

```
print(array_1d)
```


練習2 (test2.py)

- ベクトル (1次元配列) とベクトルの和

```
cols = 4 # 列数
```

```
x = [2 for _ in range(cols)] # 1次元配列を2で初期化
```

```
y = [3 for _ in range(cols)] # 1次元配列を3で初期化
```

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

```
    x[i] = x[i] + y[i]
```

```
print(x)
```

練習3 (test3.py)

- 行列（2次元配列）と行列の和

```
rows = 3 # 行数
```

```
cols = 4 # 列数
```

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

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

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

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

```
        x[j][i] = x[j][i] + y[j][i]
```

```
print(x)
```

練習4-1

- 行ベクトル（1次元配列）と行列（2次元配列）の積
- 1行N列の行列とN行M列の行列の積
- 答えは1行M列の行列、つまり行ベクトルになる
- 復習
 - L行N列の行列とN行M列の行列の積はL行M列の行列になる
 - $L \times N$ 行列と $N \times M$ 行列の積は $L \times M$ 行列 ともいう
 - Nの部分と同じ場合にしか行列積は計算できない

行列積の定義

$$\bullet \begin{bmatrix} \textcolor{red}{1} & \textcolor{red}{2} & \textcolor{red}{3} \\ \textcolor{blue}{4} & \textcolor{blue}{5} & \textcolor{blue}{6} \end{bmatrix} \begin{bmatrix} \textcolor{green}{7} & \textcolor{yellow}{8} \\ \textcolor{green}{9} & \textcolor{yellow}{10} \\ \textcolor{green}{11} & \textcolor{yellow}{12} \end{bmatrix} =$$

$$\begin{bmatrix} \textcolor{red}{1} \times \textcolor{green}{7} + \textcolor{red}{2} \times \textcolor{green}{9} + \textcolor{red}{3} \times \textcolor{green}{11} & \textcolor{red}{1} \times \textcolor{yellow}{8} + \textcolor{red}{2} \times \textcolor{yellow}{10} + \textcolor{red}{3} \times \textcolor{yellow}{12} \\ \textcolor{blue}{4} \times \textcolor{green}{7} + \textcolor{blue}{5} \times \textcolor{green}{9} + \textcolor{blue}{6} \times \textcolor{green}{11} & \textcolor{blue}{4} \times \textcolor{yellow}{8} + \textcolor{blue}{5} \times \textcolor{yellow}{10} + \textcolor{blue}{6} \times \textcolor{yellow}{12} \end{bmatrix}$$

行列積の定義

$$\bullet \begin{bmatrix} \boxed{1} & \boxed{2} & \boxed{3} \\ \boxed{4} & \boxed{5} & \boxed{6} \end{bmatrix} \cdot \begin{bmatrix} \boxed{7} & \boxed{8} \\ \boxed{9} & \boxed{10} \\ \boxed{11} & \boxed{12} \end{bmatrix} = \begin{bmatrix} \boxed{1 \times 7 + 2 \times 9 + 3 \times 11} & \boxed{1 \times 8 + 2 \times 10 + 3 \times 12} \\ \boxed{4 \times 7 + 5 \times 9 + 6 \times 11} & \boxed{4 \times 8 + 5 \times 10 + 6 \times 12} \end{bmatrix}$$

練習4-1 (test4-1.py)

- ベクトル (1次元配列) と行列 (2次元配列) の積

```
rows = 3 # 行数
```

```
cols = 4 # 列数
```

```
v = [i for i in range(rows)]
```

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

```
w = [0 for _ in range(cols)]
```

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

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

```
        w[j] = w[j] + v[i] * x[i][j]
```

```
print(w)
```

練習4-2 (test4-2.py)

- ベクトル（1次元配列）と行列（2次元配列）の積の実行時間の測定

```
import time
rows = 3000 # 行数
cols = 4000 # 列数
v = [i for i in range(rows)]
x = [[(i+j) for i in range(cols)] for j in range(rows)]
w = [0 for _ in range(cols)]
t1=time.time()
for j in range(cols):
    for i in range(rows):
        w[j] = w[j] + v[i] * x[i][j]
t2=time.time()
print("Execution time = ",t2-t1)
```

実行例

```
(base) PS C:\Users\nakada\deeplearning> python test1.py
[6.0, 6.0, 6.0, 6.0]
(base) PS C:\Users\nakada\deeplearning> python test2.py
[5, 5, 5, 5]
(base) PS C:\Users\nakada\deeplearning> python test3.py
[[1, 3, 5, 7], [3, 5, 7, 9], [5, 7, 9, 11]]
(base) PS C:\Users\nakada\deeplearning> python test4-1.py
[5, 8, 11, 14]
(base) PS C:\Users\nakada\deeplearning> python test4-2.py
Execution time = 2.961264133453369
(base) PS C:\Users\nakada\deeplearning> |
```


実行時間測定のポイント

- 実時間（Wall-clock）で測定すること
 - CPU時間（`process_time()`）を使うと大きすぎたり小さすぎたりする
 - 体感時間と一致していることを確認すること
- 少なくとも0.1秒以上実行すること
 - 短すぎる場合は配列のサイズを増やしたり、複数回実行を繰り返すようにする
- 配列サイズを一気に大きくすると急に終わらなくなるので注意
 - 「10でも1,000でも一瞬だったから100,000」とすると、いきなり10分かかったりする
- 基本的にグラフを描くのが良い

演習

- 行列（2次元配列）と行列（2次元配列）の積

☆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（仮）

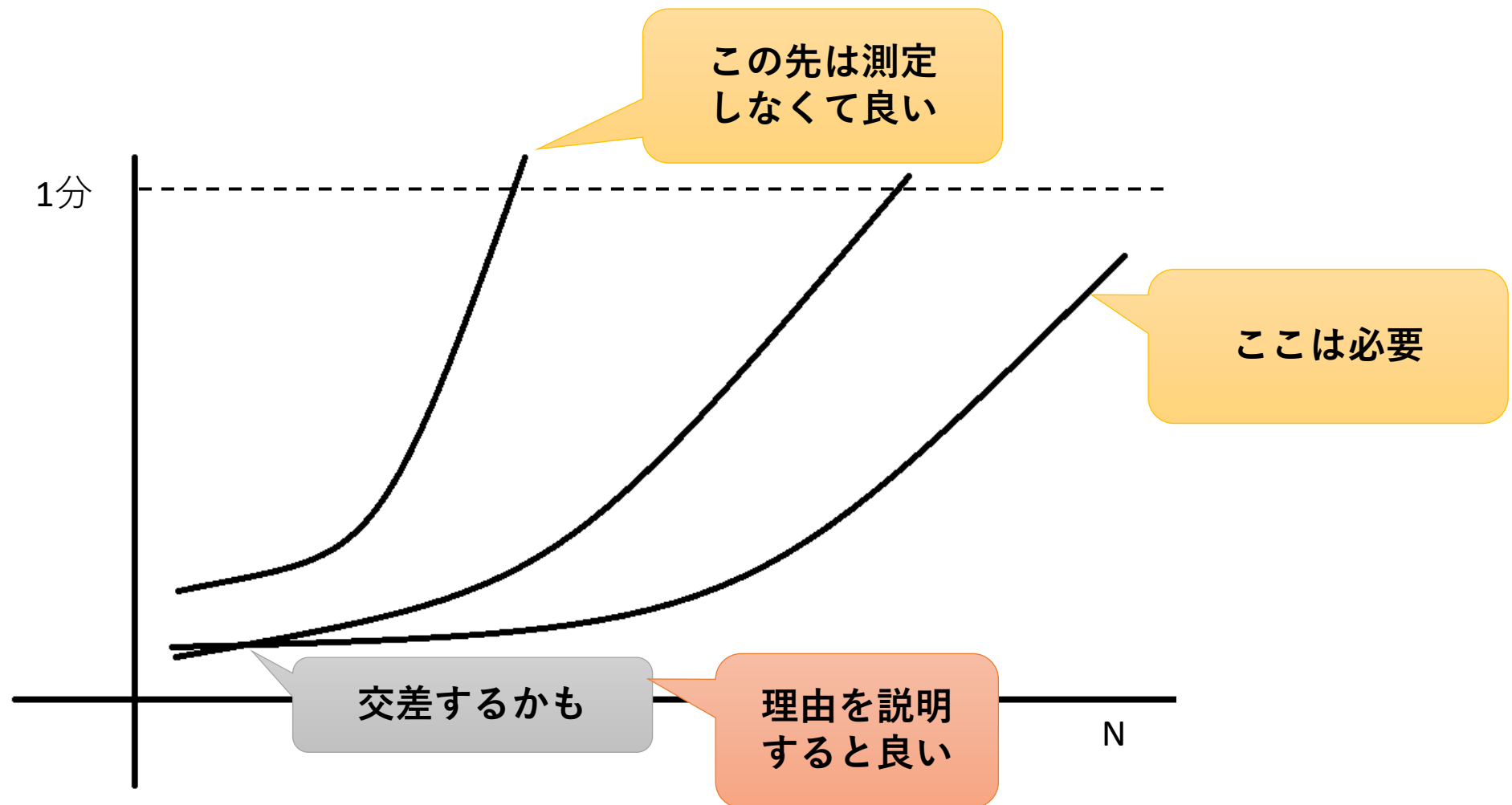
- **2つの正方行列の積と正方行列と 3×3 行列の畳み込み**それぞれを求めるpythonプログラムを作成して、それぞれの行列サイズと実行時間の関係を調べる
 - python, numpy, pytorchの実行時間を比較する
 - cupyを含めても良い
- レポートにはグラフを含めること
 - 単に載せるのではなくグラフから読み取れる内容を説明すること
 - 実行時間の上限は「これ以上増やしても読み取れる内容が増えない」
と思えるところまで
 - とはいえ1分以上の実行は不要です
- 締切：4/24 23:59

締切：約2週間後

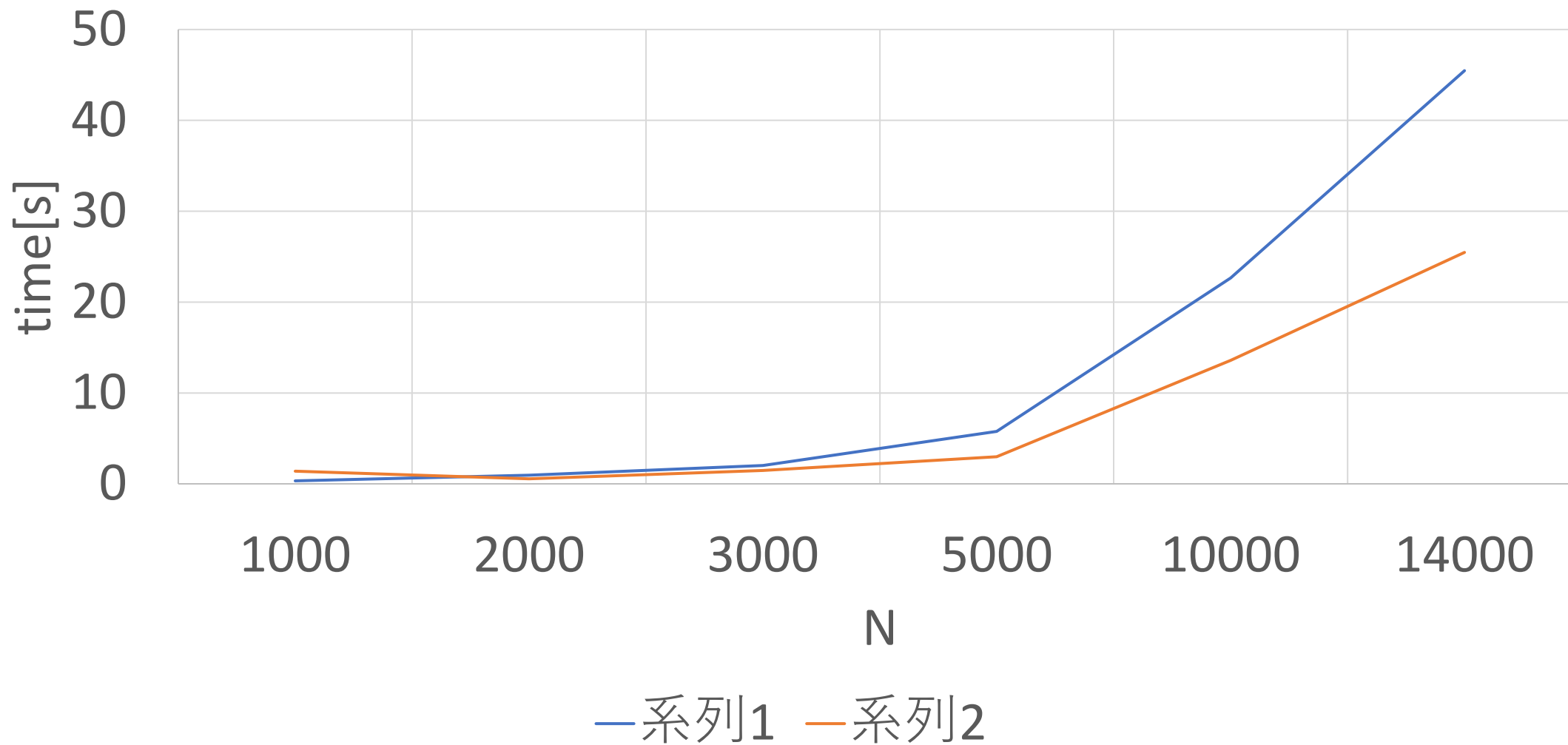
任意課題（仮） 満点5点

- 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
 - これは課題2で検証してもらいます
- 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

実行時間の測定



良くないグラフ



良くないグラフと良くない考察

折れ線グラフ

