日本機械学会 中四国支部
第132回講習会

深層学習の基礎と演習

二宮 崇
愛媛大学
ninomiya@cs.ehime-u.ac.jp

本講座の概要

- 深層学習ツールであるPyTorchを用い、深層学習の技術について学ぶ。

  前半は機械学習の基礎とPythonについて学び、後半は、深層学習の基礎と深層学習ツールであるPyTorchについて学ぶ。

スケジュール

- 11月22日(月) 13:00～18:00
  - 13:00～13:30 導入【講義】
  - 13:30～14:50 環境構築とPython演習【実習】
  - 15:00～16:00 深層学習 ニューラルネットワークの仕組み【講義】
  - 16:10～18:00 PyTorchの基礎

教材

- 授業は配布資料をベースに進める
- ソースコード

http://aiweb.cs.ehime-u.ac.jp/~ninomiya/enpitpro/

- 参考書
  - 斎藤康毅, ゼロから作るDeep Learning――Pythonで学ぶディープラーニングの理論と実装, オライリージャパン, 2016.
  - Guido van Rossum, Pythonチュートリアル 第3版, オライリージャパン, 2016. (https://docs.python.jp/3/tutorial/)
  - 岡谷貴之, 深層学習 (機械学習プロフェッショナルシリーズ), 講談社, 2015.
  - 神嶌 敏弘, 機械学習のPythonとの出会い, 2017. (http://www.kamishima.net/mlmpyja/)
  - Pytorch公式サイト(https://pytorch.org/)

自己紹介

- **名前: 二宮 崇 (にのみや　たかし)**
- **所属: 愛媛大学 大学院理工学研究科 電子情報工学専攻**
- **肩書: 教授**
- **経歴:**
  - 1992～1996 東京大学 理学部 情報科学科
  - 1996～1998 東京大学 大学院理学系研究科 情報科学専攻 修士課程
  - 1998～2001 東京大学 大学院理学系研究科 情報科学専攻 博士課程
  - 2001～2006 JST研究員
  - 2006～2010 東京大学 情報基盤センター 講師
  - 2010～2017 愛媛大学 大学院理工学研究科 准教授
  - 2017～現在 愛媛大学 大学院理工学研究科 教授

自己紹介

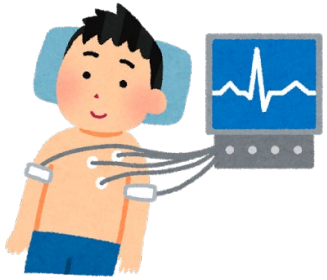
- **自然言語処理の研究(1995～2006 東京大学 辻井研)**
  - 言語学的文法(HPSG)を用いた構文解析の研究
- **機械学習と自然言語処理の研究(2006～2010 東京大学 中川研, 2010～愛媛大学 人工知能研究室)**
  - 言語学的文法(HPSG)を用いた構文解析の研究
  - オンライン学習、特徴選択の研究
- **深層学習と自然言語処理の研究(2014～)**
  - 深層学習を用いた評判分析、述語項構造の意味表現獲得
  - 機械翻訳
  - シンボルグラウンディング、マルチモーダル機械翻訳、文法誤り訂正、自動要約

導入: 人工知能とは？

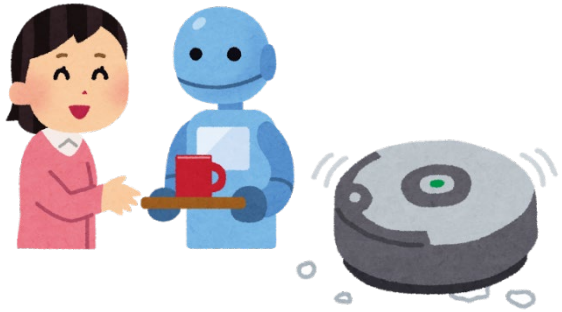
背景

- 人工知能(Artificial Intelligence; AI)が様々な場面で活躍しはじめている

将棋・囲碁



医療診断

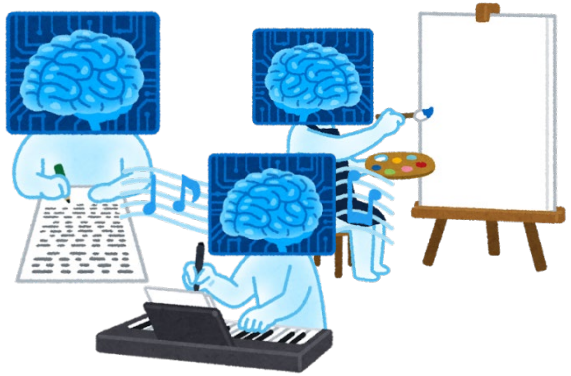


家事手伝いロボット

対話ロボット

証券取引

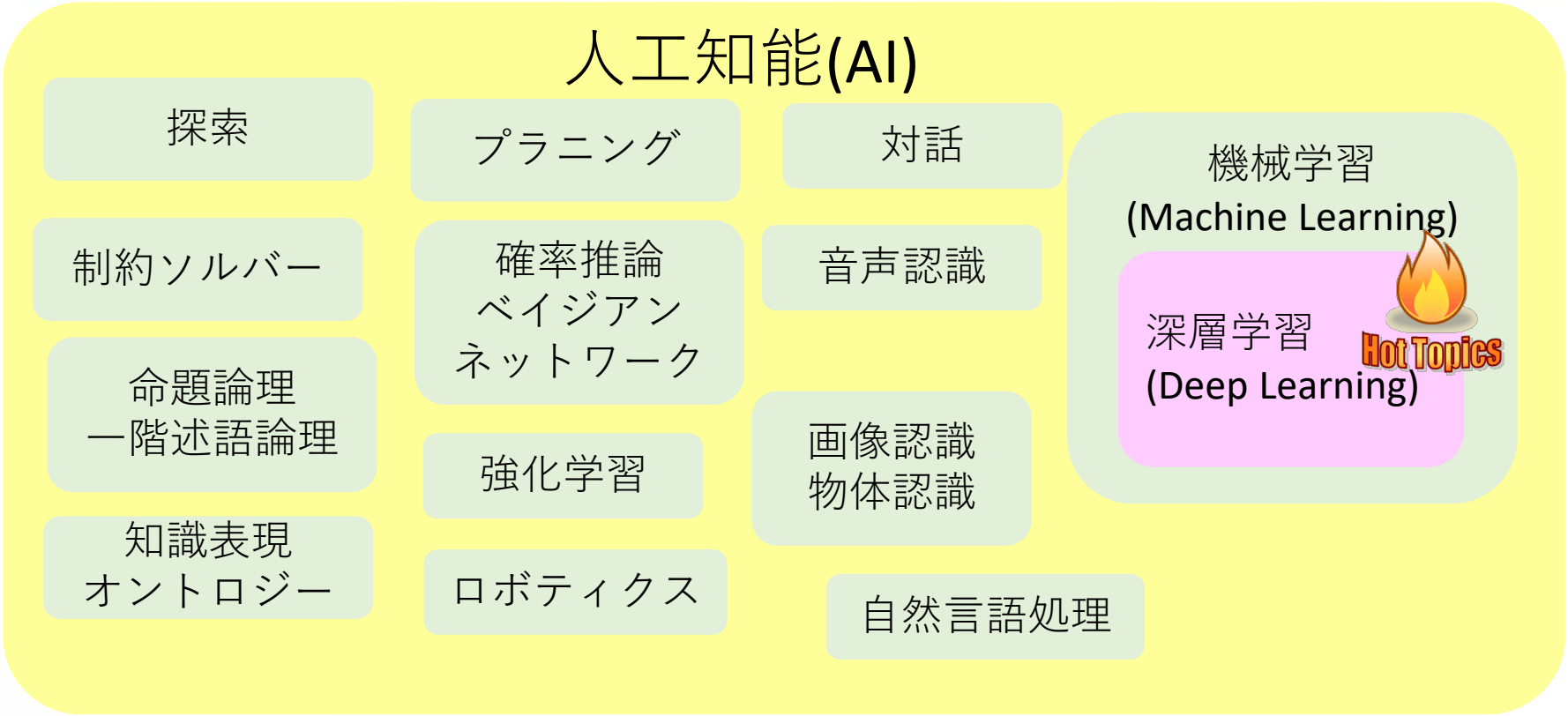
融資



芸術

自動運転

人工知能分野



機械学習：データから法則を見つけ出す手法

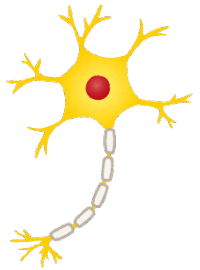
深層学習：ニューラルネットワーク(NN)を複数層重ねたモデル

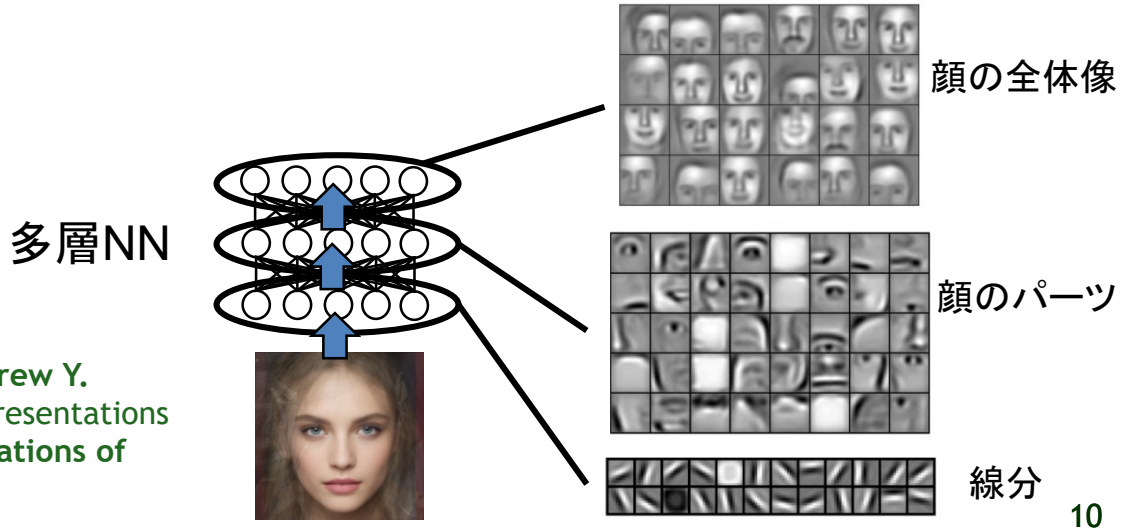
Stuart Russell, Peter Norvig (2010) **Artificial Intelligence: A Modern Approach, 3rd Edition.**

深層学習

- **深層学習 ＝　多層ニューラルネットワーク(NN)による学習**
  - ニューラルネットワーク
    - 人間の脳の神経細胞(ニューロン)の仕組みを模した計算モデル
      - パーセプトロン (Rosenblatt 1958)
      - 畳み込みNN (福島 1980)(LeCun+ 1989)
  - 特徴抽出、表現学習

深層学習では入力の特徴が自動的に学習されることがわかってきた





Honglak Lee, Roger Grosse, Rajesh Ranganath, Andrew Y. Ng (2011) Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks, Communications of the ACM, Vol. 54 No. 10, Pages 95-103 より引用

深層学習のインパクト(1/3)

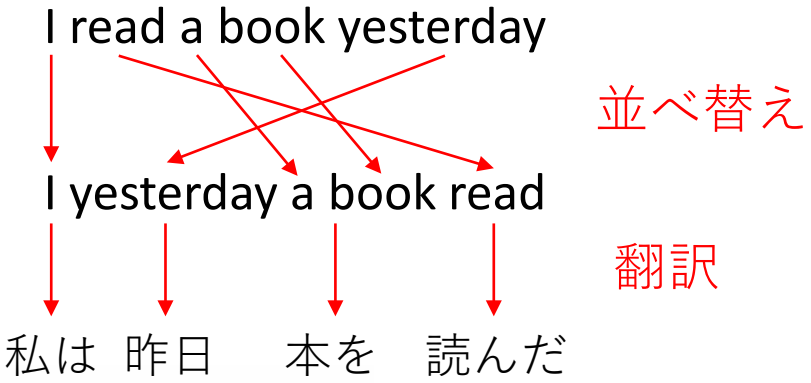
- **深層学習の圧倒的精度**
  - 2012年のILSVRC (ImageNet Large Scale Visual Recognition Challenge)における画像分類の精度
    - 2010年 71.8% (NEC Labs America, Univ. of Illinois at Urbana-Champaign, Rutgers Univ.)
    - 2011年 74.2% (Xerox Research Center Europe, CIII)
    - 2012年 83.6% (Univ. of Toronto) … この年に深層学習が圧勝
    - 2013年 88.3% (Clarifai)
    - 2014年 93.3% (Google)
    - 2015年 96.4% (MSRA) … 人間の分類精度(95%)を超えたと言われる
    - 2016年 97.0% (公安調査庁第3研究所, 中国)
    - 2017年 97.75% (Momenta, Univ. of Oxford)

Olga Russakovsky+ (2015) ImageNet Large Scale Visual Recognition Challenge, International Journal of Computer Vision, Volume 115, Issue3, pp 211-252
http://image-net.org/challenges/LSVRC/2017/index.php

深層学習のインパクト (2/3)

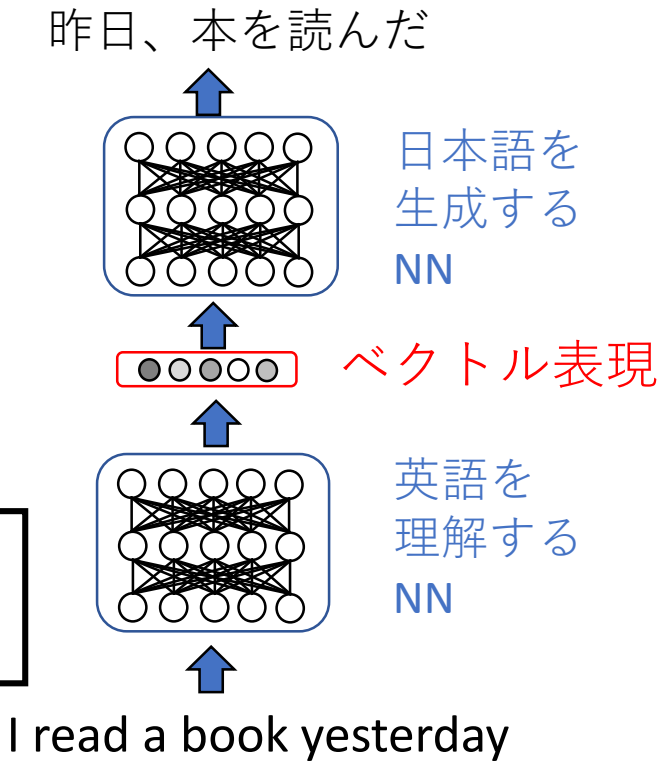
従来の機械翻訳「統計的機械翻訳」

- 並べ替え確率と翻訳確率による確率モデル



深層学習による機械翻訳「ニューラル機械翻訳」

- ベクトル表現を中間表現とするニューラルネットワーク(NN)



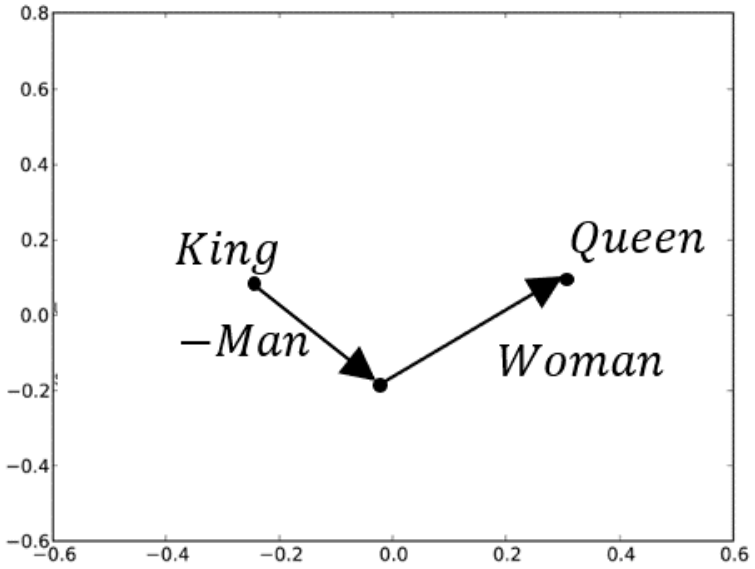
従来の統計的機械翻訳の性能を抜いて
人間が行う翻訳にかなり近くなった

深層学習のインパクト (3/3)

単語の意味表現

- 意味の計算ができる単語表現 (Mikolov+ 13, Pennington 14)

例： King – Man + Woman ≒ Queen



シンボルグラウンディング

- 画像とテキスト間の関係を学習
  - 画像や動画の説明文生成

two young girls are playing with lego toy.

A. Karpathy, Li Fei-Fei (2015) Deep Visual-Semantic Alignments for Generating Image Descriptions, CVPR 2015

AIの歴史

- **AIブームと冬の時代**

1950年代　1960年代　1970年代　1980年代　1990年代　2000年代　2010年代

第一次AIブーム (探索と推論)　冬の時代　第二次AIブーム (知識表現)　冬の時代　第三次AIブーム (機械学習)

社会の期待 >> 技術的限界

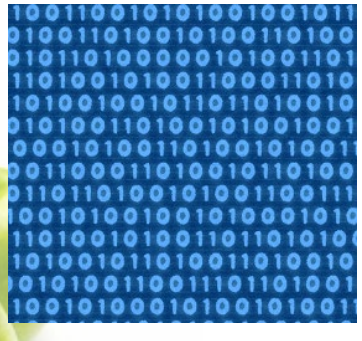
冬の時代を迎えたくない…

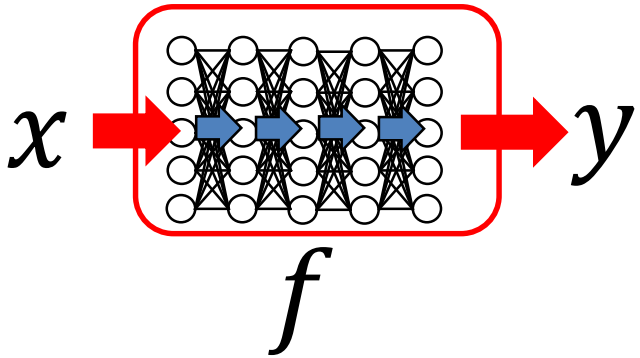
機械学習

- 機械学習 = データから関数を学習する学問分野

大量のデータ  $f(x)$

- 深層学習も機械学習の一種





機械学習

- **データから関数を学習**
  - 関数は、入力(x)と出力(y)の関係を表す

$$x \xrightarrow[f]{} y$$

  - 大量の入出力ペア(x, y)の集まり(データ)から関数fを自動的に学習！

データ

$$(x_1, y_1)$$
$$(x_2, y_2)$$
$$(x_3, y_3)$$
$$\vdots$$
$$(x_n, y_n)$$

学習 → $f(x)$

学習には、入力(x)と出力(y)のペアが揃ったデータが必要！(教師つき学習)

正解を必要としない学習(教師なし学習)もあるけど、そんなに高い精度を実現するわけではない

機械学習

例：犬猫判別機 $f(x)$

入力x

f

出力y

犬

データ


：猫

：犬


：犬

：猫


：犬 …

大量のデータから関数f を学習する　f: 画像 → {犬/猫}

機械学習

- **データから学習**
  - 入力 $\boldsymbol{x} = (x_1, x_2, \cdots, x_m)$　←m次元ベクトル
    （画像の場合は、赤、青、緑の画像に対応する3つの行列）
  - データDは入力$\boldsymbol{x}$と出力yのペアの集合

  - 関数fはパラメータ(重み変数)の集合 $\boldsymbol{w} = (w_1, w_2, \cdots w_m)$から成る
    例: $f(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$
    - fは$f_{\boldsymbol{w}}(\boldsymbol{x})$と書くとわかりやすい

  - 学習 = データDに対する誤差を最小にする重み変数wを求める

$$\text{データ全体の誤差 } L(\boldsymbol{w}) = \sum_{(\boldsymbol{x},y)\in D} \left(y - f_{\boldsymbol{w}}(\boldsymbol{x})\right)^2$$

他にも、マージン最大化、確率最大化による学習がある。
関数の最小値を求める方法は「最適化」という領域で研究されている

機械学習における学習と推論

- **学習 (learning)**
  - データから良いパラメータを推定すること
  - 損失関数を最小化することによりパラメータが得られる
  - 推定 (estimation)、パラメータ推定 (parameter estimation) ともいう
  - 最尤推定、MAP推定、ベイズ推定、マージン最大化が有名

- **推論 (inference)**
  - 未知のデータに対し、学習した関数fを適用し、出力を予測すること
  - 予測ともいう

機械学習の教科書

- **機械学習**
  - Christopher M. Bishop, **Pattern Recognition and Machine Learning**
    - (邦訳) C. M. ビショップ他 パターン認識と機械学習 上・下
  - Nello Cristianini, John Shawe-Taylor, **An Introduction to Support Vector Machines and other kernel-based learning methods**
  - Trevor Hastie, Robert Tibshirani, Jerome Friedman, **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**
    - **(邦訳)** Trevor Hastie, Robert Tibshirani, Jerome Friedman他,統計的学習の基礎 ―データマイニング・推論・予測―
  - 中川裕志,**東京大学工学教程 情報工学 機械学習**
  - 須山敦志, ベイズ深層学習 (機械学習プロフェッショナルシリーズ)
- **数値最適化**
  - Jorge Nocedal, Stephen Wright, **Numerical Optimization**
  - Dimitri P. Bertsekas, **Nonlinear Programming**
  - 金森 敬文, 鈴木 大慈, 竹内 一郎, 佐藤 一誠, **機械学習のための連続最適化 (機械学習プロフェッショナルシリーズ)**

回帰問題と分類問題と構造予測

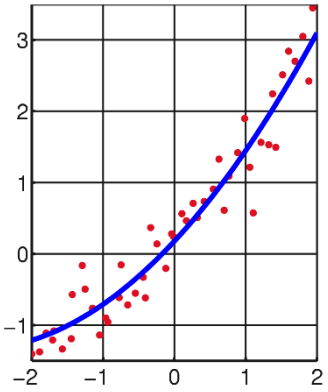
関数: $y = f(\boldsymbol{x})$

- **回帰問題 (regression)**
  - $y \in R$ (実数)

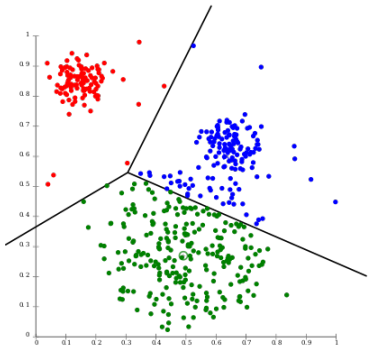
  例: 年齢予測、降水確率の予測、気温の予測
- **分類問題 (classification)**
  - $y \in \{C_1, C_2, \cdots, C_K\}$ (ラベル集合)

  例: 文書分類(政治、経済、スポーツ等)
- **構造予測 (structured prediction)**
  - $y \in G$ (グラフ集合)

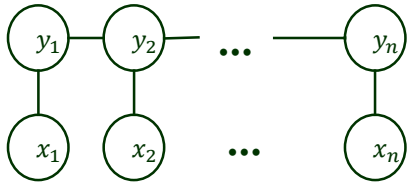
回帰

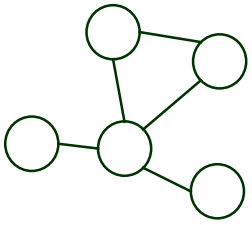


分類



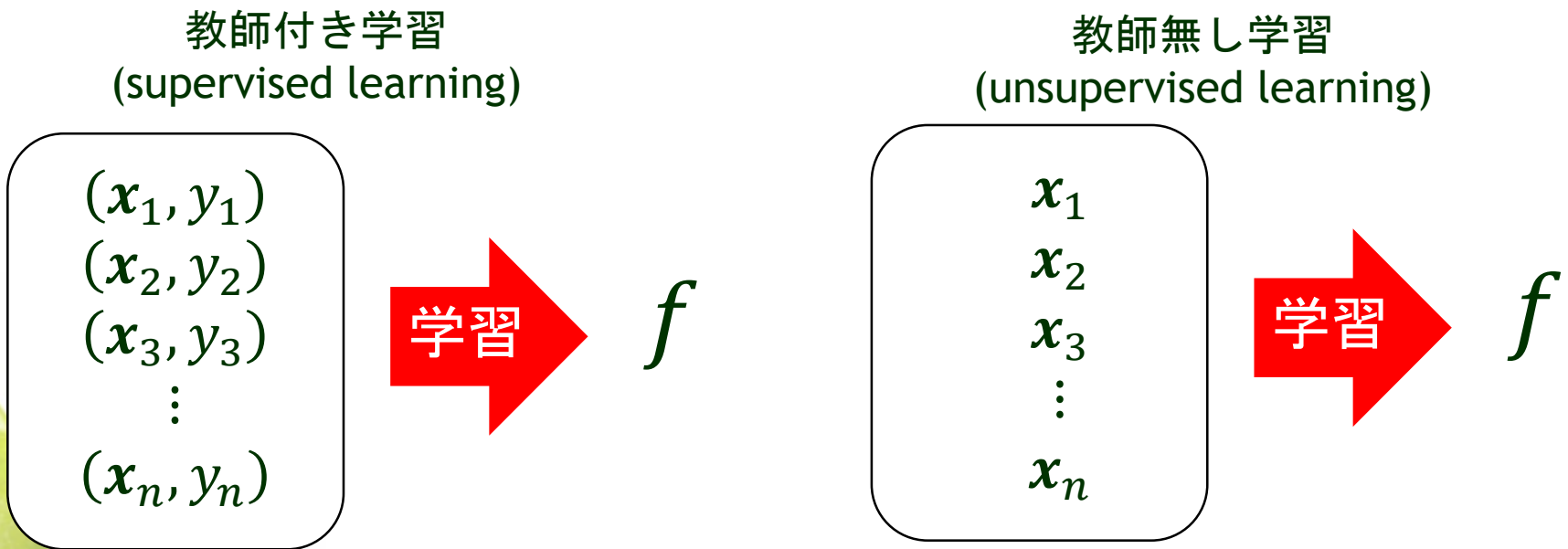
構造予測





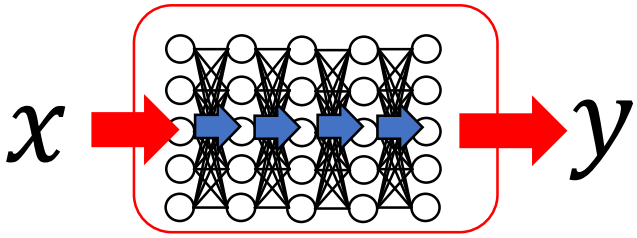
教師付き学習と教師無し学習

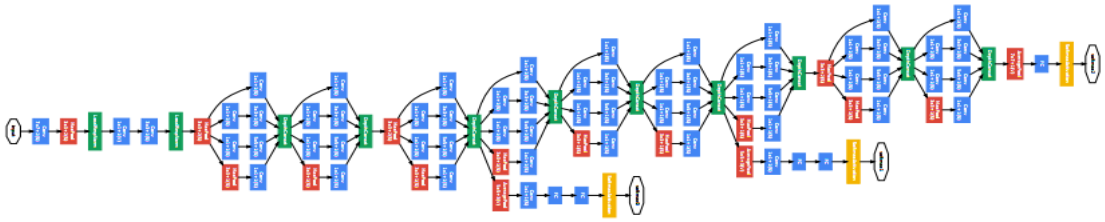
- **教師付き学習**
  - 入出力ペア$(\boldsymbol{x}, y)$の集まり(データ)から関数fを予測
- **教師無し学習**
  - 入力xの集まり(データ)から関数fを予測



深層学習

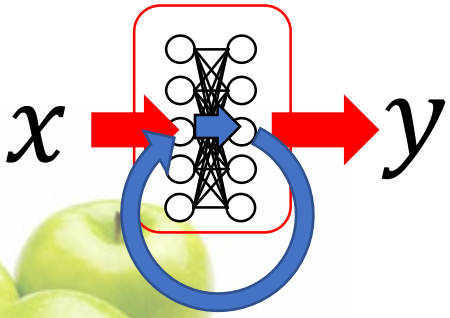
- 関数：重み付き線形和と非線形変換を多層化したもの
  - フィードフォワードニューラルネットワーク

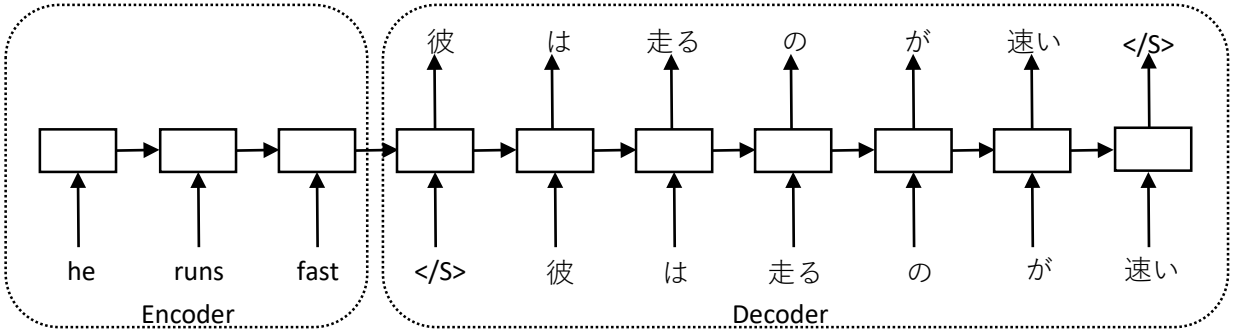




GoogLeNet (22層)　※画像分類に有効なネットワーク

  - リカレントニューラルネットワーク



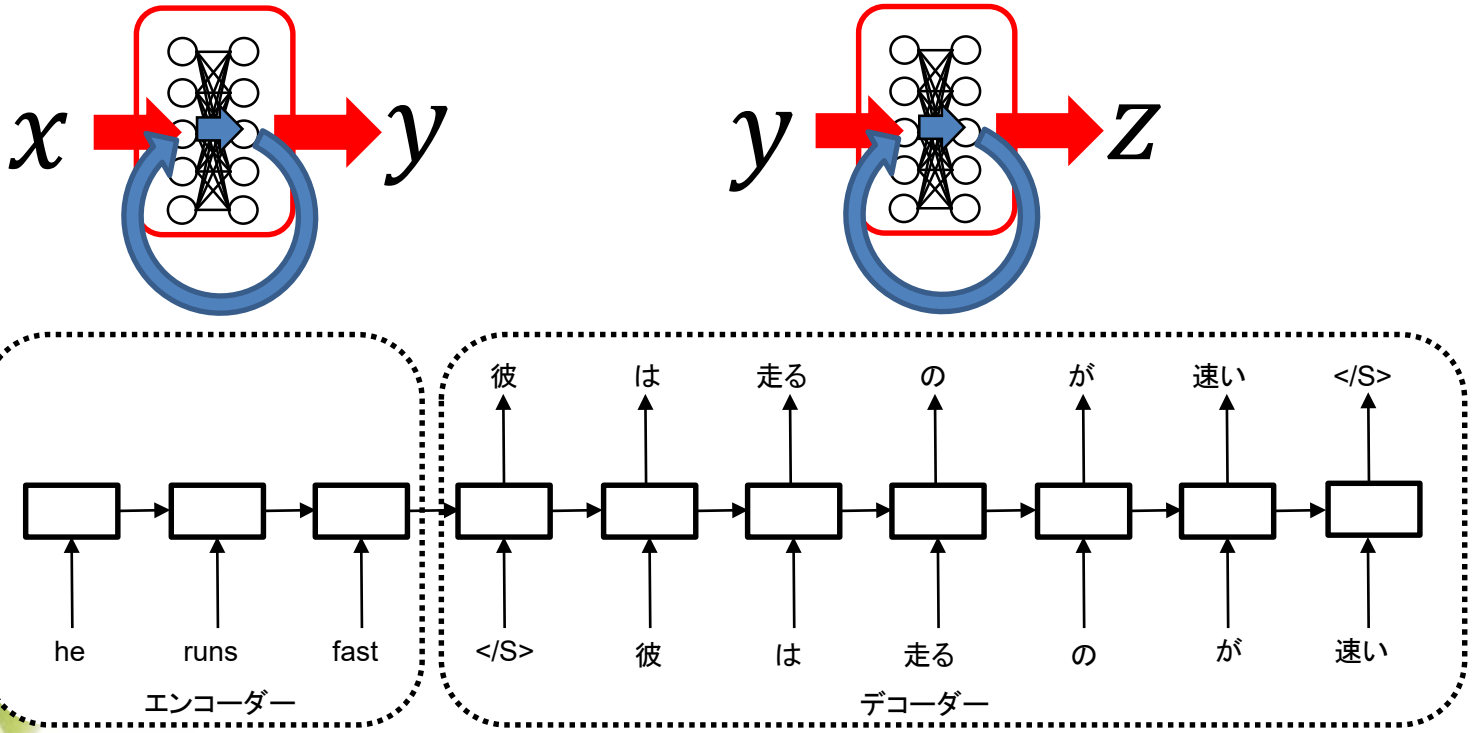


Encoder-Decoderによる機械翻訳

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich (2014) Going Deeper with Convolutions, CVPR 2015

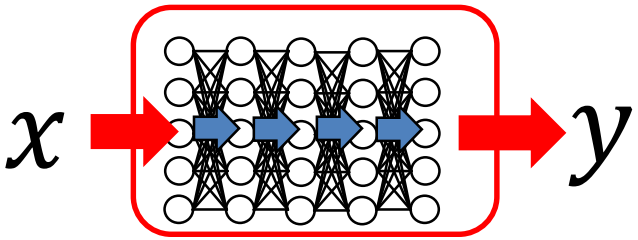
ニューラル機械翻訳

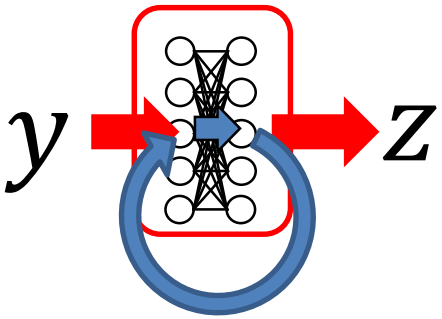
- **2つのリカレントニューラルネットワークを用いた機械翻訳**
  - エンコーダー: 入力文(英語)を中間表現(数百次元のベクトル)に変換
  - デコーダー: 中間表現から出力文(日本語)に変換

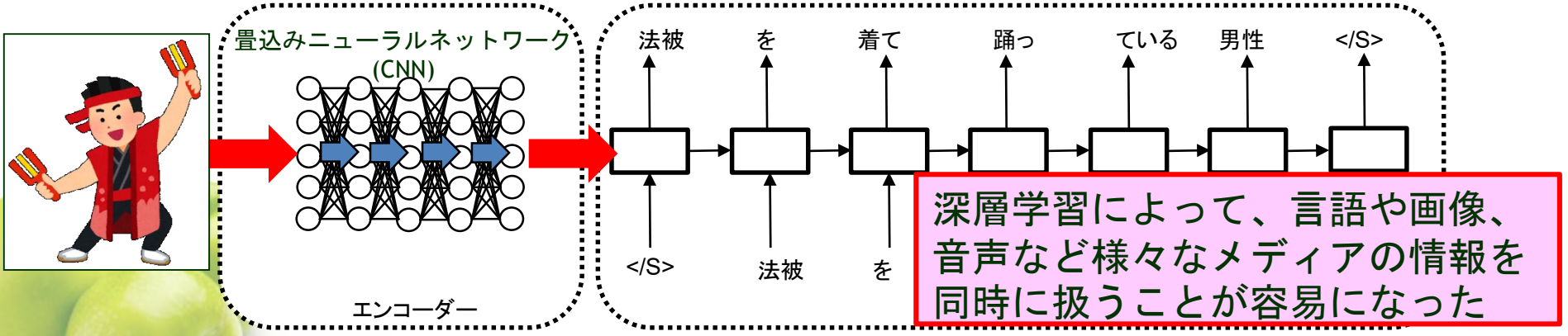


キャプション生成(説明文生成)

- **畳込みニューラルネットワーク(CNN)とリカレントニューラルネットワーク(RNN)を用いたキャプション生成**
  - エンコーダー: 画像を中間表現(数百次元のベクトル)に変換
  - デコーダー: 中間表現から出力文(日本語)に変換







深層学習の高速化

- **学習アルゴリズム**
  - 計算グラフによる誤差逆伝搬法
  - オンライン学習(確率的勾配降下法、モーメンタム、AdaGrad、Adam)

- **GPU (CPU1コアよりも10倍以上速い)**
  - NVIDIA GeForce RTX 3060 (約8.5万円)
  - NVIDIA Tesla V100 32GB (約100万円)

まとめ：AIの現状

- **機械学習、深層学習が現在のAIブームを牽引**
- **機械学習**
  - データから関数を学習する
  - 入力と出力の両方が揃った教師データが必要
- **深層学習**
  - 機械学習の一種、多層ニューラルネットワーク
  - 高速化

実習編: PYTHON

環境設定

- **次のいずれかの環境設定を行いましょう（上から順に易しい設定となっています）**
  - 環境設定 Google Colaboratory（クラウド環境）Googleアカウントがあればすぐ
  - 環境設定 Anaconda Windows/Linux/MacOS（PC環境）
  - 環境設定 VirtualBox+Ubuntu（サーバー環境およびエッジ環境）
  - 環境設定 Jetson Nano編（エッジ環境）

Python演習

- Python https://www.python.jp/
  - 最もよく使われているプログラミング言語の一つ
    - 多くのプログラミング言語ランキングで1位～2位。
  - ライブラリが充実しており、データ処理、データ解析、機械学習のためによく使われる。特に深層学習のプログラムはPythonで書くことが多い。
  - ドキュメント https://docs.python.jp/3/

※Pythonには2系と3系があり、構文や同じ関数でも処理が違う場合があるので注意。本講義ではPython 3系を想定。

Python環境設定

- **深層学習(ディープラーニング)を行うには？**
  - クラウド環境を使う方法
    - Google Colaboratory (略称 Colab)
      - https://colab.research.google.com/
      - Googleが提供するクラウド環境の深層学習プラットフォーム
      - Googleアカウントがあれば無料で制限付きながら使える
      - Googleアカウントをもっていればおすすめ
    - Microsoft Azure Machine Learning (通称 Azure)
      - Microsoftアカウント+サブスクライブで使える。
      - 無料で1年間だけ制限付きながら使える。

Python環境設定

- **深層学習(ディープラーニング)を行うには？**
  - 自分が持っているサーバー/PCを使う方法
    - Ubuntuに自力でいろんなライブラリを追加する
      - バージョンの違いでうまくインストールできなかったり動かなかったりで結構大変。
    - Anacondaを使う（オススメ！）
      - Python + 科学技術計算(データサイエンス)ライブラリ
      - データサイエンス向けライブラリが全部入りで最初から入っている
      - 最初から入っているライブラリ
        - NumPy ベクトル行列演算ライブラリ
        - SciPy 数値演算、最適化、信号処理、関数
        - Pandas 統計処理、データ分析、時系列解析
        - scikit-learn 機械学習

Python演習

PYTHONプログラミング

Python 算術演算

- **Pythonでの算術演算**
  - 足し算 +
  - 引き算 -
  - 掛け算 *
  - 割り算 /
  - 整数の割り算 //
  - 余りの計算 %
  - べき乗 **
- **右のプログラムコードを実行してみよう**

```
[1] 1+3
    4

[2] 3-1
    2

[3] 5*2
    10

[4] 2**4
    16

[5] 7/5
    1.4

[6] 7//5
    1

[7] 7%5
    2
```

Python 変数

- アルファベットで定義
- 整数型(int)や実数型(float)といった型は宣言せず自動的に決定される
- 変数への代入は「=」
- 変数の型や計算結果の型はtype関数で調べることができる
- 右のプログラムを実行してみよう

```
[8]  x=100

[9]  x

     100

[10] type(x)

     int

[11] y=3.14

[12] type(y)

     float

[13] x*y

     314.0

[14] type(x*y)

     float
```

変数の中身をみることができる

intは整数型という意味

floatは実数型という意味

計算結果にも型がついている

Python 変数

- **次のプログラムを実行してみよう**

```
[15] a=1

[16] a
     1

[17] type(a)
     int

[18] b=1.0

[19] type(b)
     float
```

小数点をつけない数字は整数型と判断される

実数型にするには小数点をつけて入力する

```
[20] c=1.

[21] c
     1.0

[22] type(c)
     float
```

実数型をこのように入力しても良い

ブール型

- **ブール型(bool): TrueかFalseの値**
- **ブール演算**
  - and演算: $x\ and\ y = \begin{cases} True & xとyの両方ともTrueのとき \\ False & それ以外 \end{cases}$
  - or演算: $x\ or\ y = \begin{cases} True & xとyのどちらかもしくは両方がTrueのとき \\ False & それ以外 \end{cases}$
  - not演算: $not\ x = \begin{cases} True & xがFalse \\ False & xがTrue \end{cases}$ (xのTrue/Falseの反転)

ブール型

- コンソールで次のPythonプログラムを書いてみよう

```
[23] hungry=True

[24] sleepy=False

[25] type(hungry)

     bool

[26] not hungry

     False

[27] hungry and sleepy

     False

[28] hungry or sleepy

     True
```

文字列型

- **文字列(string)**
  - ’(シングルクォート)で囲む
  - ”(ダブルクォート)で囲む
  - ”””(ダブルクォート3つ)で囲む
  - どの方法でも同じ文字列が得られる
- **文字列の連結は「+」で実行できる**
- **右のプログラムを実行してみよう**

```
[29] a = "こんにちは"

[30] b = 'こんにちは'

[31] c = """こんにちは"""

[32] a

     'こんにちは'

[33] b

     'こんにちは'

[34] c

     'こんにちは'

[35] a+b+c

     'こんにちはこんにちはこんにちは'
```

文字列の連結

リスト型

- リスト型(list)
  - 複数のデータをまとめる
  - コンマ区切りの値の並びを角括弧で囲む

```
[36] a = [1, 2, 3, 4, 5]

[37] a
     [1, 2, 3, 4, 5]

[38] len(a)
     5

[39] a[0]
     1

[40] a[1]
     2
```

角括弧の中にリストの各要素を並べる。各要素はカンマ(,)で区切る

リストの長さは**len関数**で計算できる

a[n]と書くことでn番目の要素を得ることができる。ただし、リストの最初の要素は0番目と数える

```
[41] a[4]
     5

[42] a[4]=99

[43] a
     [1, 2, 3, 4, 99]
```

a[n]=xと書くことでn番目の要素にxを代入することができる

リスト型

- **リストの連結**
  - +でリストを連結することができる（文字列の連結と似ている）

リストの連結

```
[44] a=[1,2,3]

[45] b=[4,5,6]

[46] a+b

     [1, 2, 3, 4, 5, 6]
```

- **リストの生成**
  - [x]*n でxを要素とする長さnのリストを生成できる

要素1の長さ10のリスト

```
[47] a=[1]*10

[48] a

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

辞書型

- **辞書型(dictionary): 一般には連想配列と呼ばれるデータ構造**
  - キーと値のペアをたくさん格納したリスト
  - キーに対応する値をすぐに取り出すことができる
  - 「キー:値」を並べて中括弧で囲むことで生成する

```
[49] height={"sato":180, "yamada":170}

[50] height["sato"]
     180

[51] height["kato"]=175

[52] height
     {'kato': 175, 'sato': 180, 'yamada': 170}
```

”sato”は180
”yamada”は170

d[x]は辞書dにおけるxの値

d[x]=yで、辞書dにおけるxの値をyにする

”kato”:175が追加されている

if文

- **条件分岐**
  - ブロック: 一連の処理の範囲のことをブロックという。
  - CやProcessingなど普通の言語では中括弧{}を用いてブロックを指定する。
    - Processingの例

```
int a = 150;
if( a > 100) {
  background(255,0,0);
  ellipse(50, 50, 50, 50);
}
```

If文の条件式が真であったときのブロック

  - Pythonのブロックはインデントで指定する
    - Pythonの例

```
[53] a = 150

[54] if a > 100:
       a = 0
       print("1")
     else:
       print("0")

     1
```

If文の条件式が真であったときのブロック

空白4個かタブが一般的

If文の条件式が偽であったときのブロック

if文

- 次のコードを実行してみよう

```
[55] a=150
     if a > 100:
       print("aは100より大きい")
     elif a > 0:
       print("aは0より大きくて100以下です")
     else:
       print("aは0以下です")
```

aの値をいろいろと変えてどのように結果が変わるのかみてみよう

```
aは100より大きい
```

if文の構文

条件式の後にコロン(:)が必要

```
if 条件式1:
        条件式1が真のときの処理
elif 条件式2:
        条件式1が偽で条件式2が真だったときの処理
…
else:
        上記の条件式がすべて偽だったときの処理
```

上から下に向かって実行

if文の条件式

- **条件式には次のようなものが使えます**
  - 等号(等しい) ==
  - 不等号(より大きい) >
  - 不等号(より小さい) <
  - 不等号(以上) >=
  - 不等号(以下) <=
- **また、ブール演算を使って、複雑な条件を書くこともできます**
  - 例

```
[56] x=150
     if 100 < x and x <= 200:
       print("xは100より大きく200以下です")

     xは100より大きく200以下です
```

for文

- Pythonのfor文はリストに対する繰り返し処理

```
[57] words = ["cat", "dog", "lion"]
     a=1
     for w in words:
       print(w+": "+str(a))
       a = a + 1
```

```
cat: 1
dog: 2
lion: 3
```

for文の構文

```
for 変数 in リスト:
        繰り返し処理のブロック
        （変数を参照しながら処理することができる）
```

for文

- Pythonのfor文はリストに対する繰り返し処理

```
[57]  words = ["cat", "dog", "lion"]
      a=1
      for w in words:
          print(w+": "+str(a))
          a = a + 1
```

① ② ③

繰り返し処理ブロック

文字列と数値の足し算はできない。
str関数を用いると数値を文字列に変換できる。

①に対する処理　②に対する処理　③に対する処理

リストの先頭から順に各要素が変数に代入されて繰り返し処理ブロックの処理が行われる

①w = "cat"として繰り返し処理ブロックを実行
②w = "dog"として繰り返し処理ブロックを実行
③w = "lion"として繰り返し処理ブロックを実行

for文とrange関数

- 一定回数繰り返したいときは？→range関数を使う

- range関数
  - range(n)で[0, 1, 2, …, n−1]の(仮想的な)リストを作る
  - range(i, j)で[i, i+1, i+2, …, j−1]の(仮想的な)リストを作る

```
[58]  list(range(20))

      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
[59]  list(range(10, 20))

      [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

for文とrange関数

- for文とrange関数を用いた繰り返し処理

[0, 1, 2, …, 99]のリストがあると思えば良い

x = i + 1とすることで、xは1, 2, …, 100の値になる

```
[60] sum=0
     for i in range(100):
       x = i + 1
       sum = sum + x
     print(sum)
```

5050

i = 0 としてx = 1, sum = sum + x
i = 1 としてx = 2, sum = sum + x
i = 2 としてx = 3, sum = sum + x
…
i = 99としてx = 100, sum = sum + x

これは$\sum_{x=1}^{100} x$ =1 + 2 + 3 + … + 100=5050の計算をしている

関数

- lenなど組み込みの関数を用いてきたが自分で新しく関数を定義することができる
- defで関数を定義

```
[61] def add(x, y):
        ans = x + y
        return ans

     print(add(10,30))
```

```
40
```

関数定義の構文

```
def 関数名(引数の列):
        関数内での処理
        return 返り値
        (返り値は関数が返す値)
```

Python演習

- 次のプログラムを入力して実行してみよう

```
def sign(x):
  if x < 0:
    print("negative")
    return -1
  elif x == 0:
    print("zero")
    return 0
  else:
    print("positive")
    return 1

x = sign(10)
y = sign(0)
z = sign(-5.3)
print(x, y, z)
```

```
positive
zero
negative
1 0 -1
```

このような結果がでていたらうまく動いているということ

Python演習

- **数値のリストを受け取ったとき、受け取ったリストの各要素を10倍にして返す関数を書きましょう。**
  - まず、「ファイル」→「ノートブックを新規作成」を選んで新しいファイルを作成しましょう。
  - 続いて、新しく作ったファイル名を変更しましょう。（例えば、tentimes.ipynbなど）

```
def tentimes(numlist):
  #ここにプログラムを書く


print(tentimes([1,2,3,4,5]))
```

```
[10, 20, 30, 40, 50]
```

うまくプログラムが書けたらこのような結果が返ってくる

Python演習 解答例

- **解答例**

```
def tentimes(numlist):
  r = []
  for x in numlist:
    r = r + [10*x]
  return r

print(tentimes([1,2,3,4,5]))
```

最初にrに空リストをいれる

rにnumlistの各要素を10倍にした値を追加していく

最後にrを返す

リストの連結はリスト同士でないとできないので、要素1個のリスト[10*x]を作ってから連結している

Python演習 解答例

- **別解**

```
def tentimes(numlist):
  r = [0]*len(numlist)
  for i in range(len(numlist)):
    r[i] = 10 * numlist[i]
  return r

print(tentimes([1,2,3,4,5]))
```

最初にnumlistと同じ長さのダミーのリストを作っておく

i=0, .., (numlistの大きさ-1)まで繰り返す

r[i]をnumlist[i]の10倍の値にする

最後にrを返す

クラス

- **独自のデータ型の定義**
  - メソッド(クラス用の関数)や属性(クラス用の変数)の定義が可能

クラスの定義

```
class Man:
  def __init__(self, name):
    self.name = name
    print('Initialized!')

  def hello(self):
    print('hello ' + self.name + '!')

  def goodbye(self):
    print('good bye '+ self.name + '!')

m = Man('Taro')
m.hello()
m.goodbye()
```

__init__：コンストラクタ（初期化用メソッド）

.でメソッドや属性にアクセス（クラス内ではself.）

メソッドの第一引数にはselfを明示的に書く

Man型オブジェクトの生成

実行結果

```
Initialized!
hello Taro!
good bye Taro!
```

クラスの継承

- 親クラス(基底クラス)に新しい機能を追加したクラス(派生クラス)を作りたい
  - 基底クラスの属性やメソッドを継承
  - 属性やメソッドの追加、上書き(オーバーライド)ができる

派生クラスの定義

```
class YoungMan(Man):
  def __init__(self, name):
    super(YoungMan, self).__init__(name)
    self.name = 'Mr. ' + self.name

  def hello(self):
    print('hi ' + self.name + '!')

m = YoungMan('Taro')
m.hello()
m.goodbye()
```

class 派生クラス名(基底クラス名):

super(派生クラス名, self).メソッド名で基底クラスのメソッドを呼び出せる

Helloメソッドを上書き

派生クラスのhelloが呼ばれる

基底クラスのgoodbyeが呼ばれる

実行結果

```
Initialized!
hi Mr. Taro!
good bye Mr. Taro!
```

(参考) Python: リストのメソッド

- **sort**
  - リストの各要素を並べ替えて昇順にソートする
  - x.sort(reverse=True)で、降順にソート
  - x.sort(key=関数)と書くことで各要素を関数に適用した値の順に並べ替える
    - 例

      x = [[1,2,3], [4,5], [6]]

      x.sort(key=len)

      →[[6], [4,5], [1,2,3]]
- **append**
  - リストの末尾に要素を追加する
  - 例 x.append(10)

(参考) Python: ラムダ式

- **ラムダ式**

lambda 変数x: 式e

- 名前無し関数を生成
- 変数xを受け取って、式eの計算結果を返す関数
- 例 f = lambda x: x+10

- 次の2つは同じ定義

```
f=lambda x: x+10
```

```
def f(x):
    return x + 10
```

(参考) Python: 内包表記

- **内包表記 (comprehension)**

  [式e for 変数x in リストl]

  - for文を用いたリストの生成を簡便に行うための構文
  - リストlの各要素を変数xに代入し、式eを実行する
  - 式eの実行結果をリストとして順に並べて出力する

  - 例

    [10*x for x in [1,2,3,4]]

    →[10, 20, 30, 40]

(参考) Python: 3項演算 (if関数)

- **Pythonにおいて通常用いられるifは制御構造文(返値がない)であったが、if関数(3項演算子)も用意されている**

式e if 条件式c else 式f

  - 条件式cを満たす時、式eの計算結果を返す。そうでなければ、式fの計算結果を返す

  - 例 x = 'odd' if y % 2 == 1 else 'even'

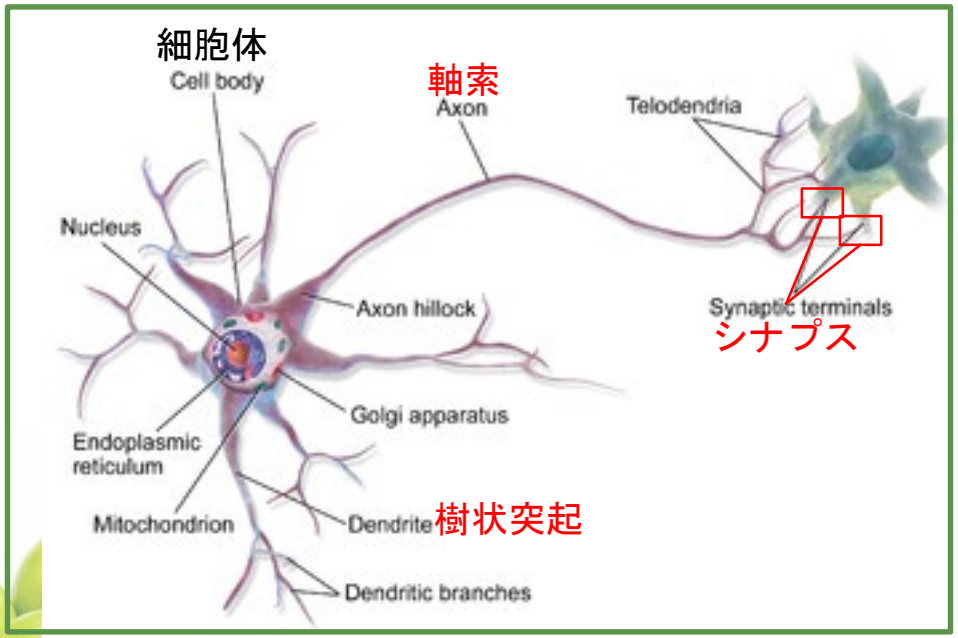
深層学習 (1)

– ニューラルネットワークの仕組み、活性化関数、推論 –

単純パーセプトロン (1)

- ニューラルネットワーク（深層学習）の起源
- 人間の脳の神経細胞（ニューロン）の仕組みを模したモデル

ニューロン

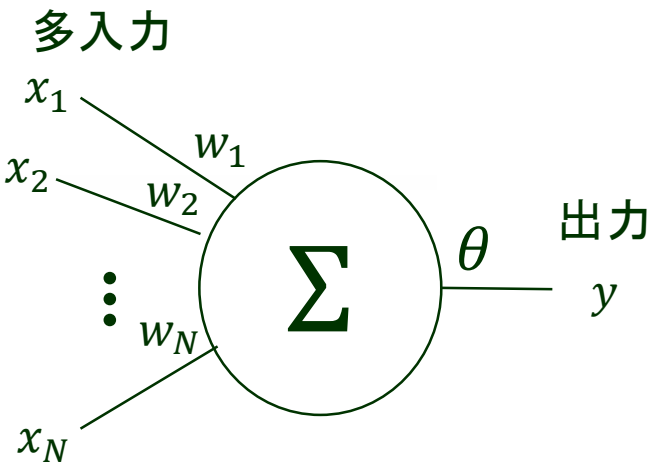


動作：

- シナプスを介して他のニューロンから電気信号を受け取る/他のニューロンへ電気信号を伝える
- 電気信号を受け取ったら電位が変化する
- 電位差が閾値を超えると出力信号を発生する（発火する）

単純パーセプトロン (2)

- ニューロンを模したもの
- 入力信号に対する固有の重みと閾値を持つ
- 動作
  - 多入力を受け付ける
  - 入力の重み付き和を計算する
  - 重み付き和が閾値を超える「1」、そうでなければ「0」を出力する



入力： $x_i \in \{0,1\}\ (i = 1 \cdots N)$

重み： $w_i \in R (i = 1 \cdots N)$

各入力の重要度をコントロール

閾値：$\theta \in R$

発火のしやすさをコントロール

出力： $y = \begin{cases} 1 & if\ \sum_{i=1}^{N} w_i x_i > \theta \\ 0 & otherwise \end{cases}$

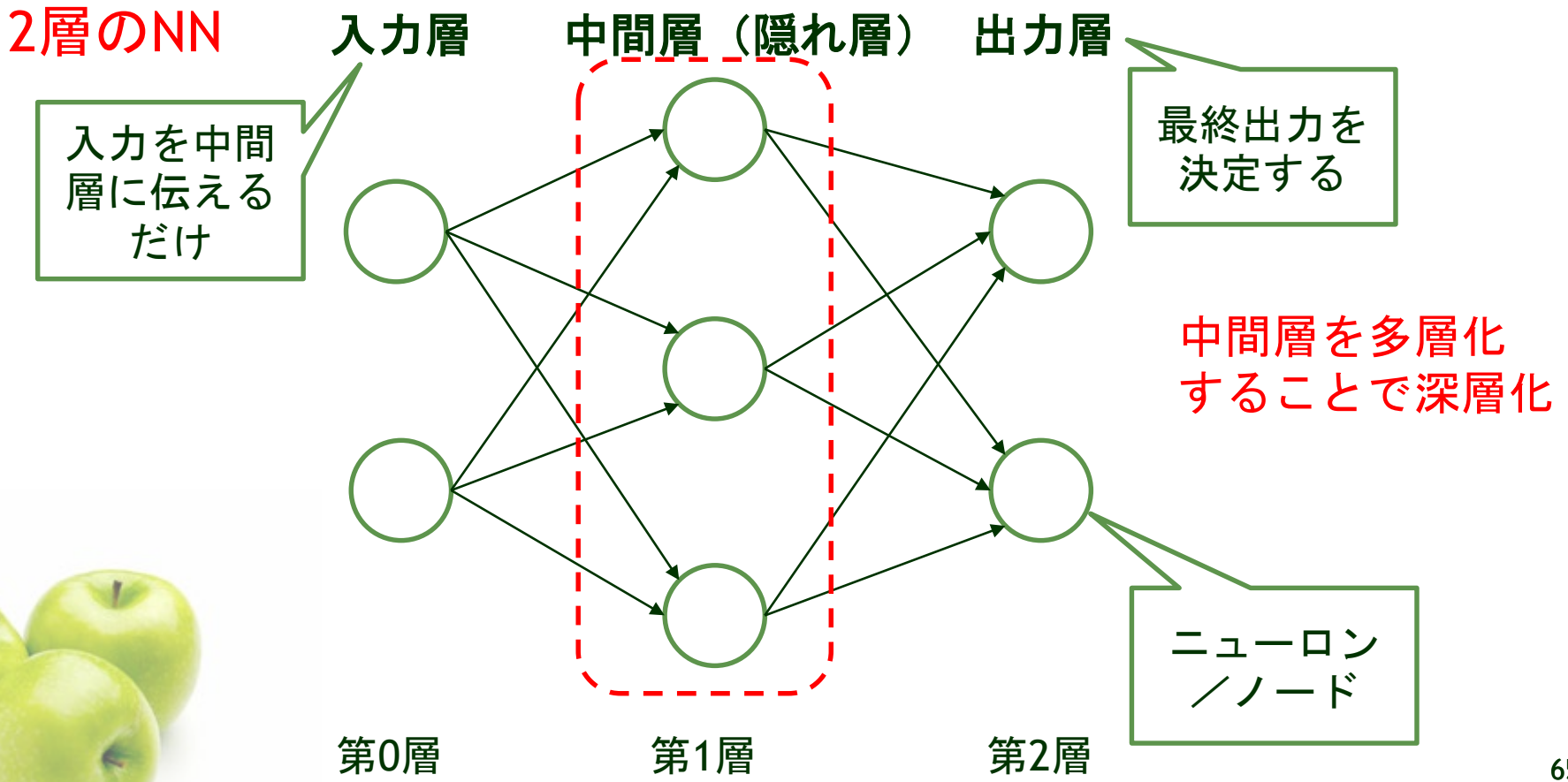
パラメータは w_i と θ

ニューラルネットワーク（NN）

- **パーセプトロンを発展させたもの**
- **大きな違いは、**
  - 中間層がある（多層）
  - 活性化関数が導入されている

標準的なNNの構造

- 順伝搬型(フィードフォワード)NN:情報が入力側から出力側に一方向に伝わるNN



活性化関数の導入

- **パーセプトロン**

  入力：$x_i \in \{0,1\}\ (i = 1 \cdots N)$

  パラメータ：$w_i, b \in R (i = 1 \cdots N)$

  出力：$y = h(a)$

$$a = \sum_{i=1}^{N} w_i x_i + b$$

$$h(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

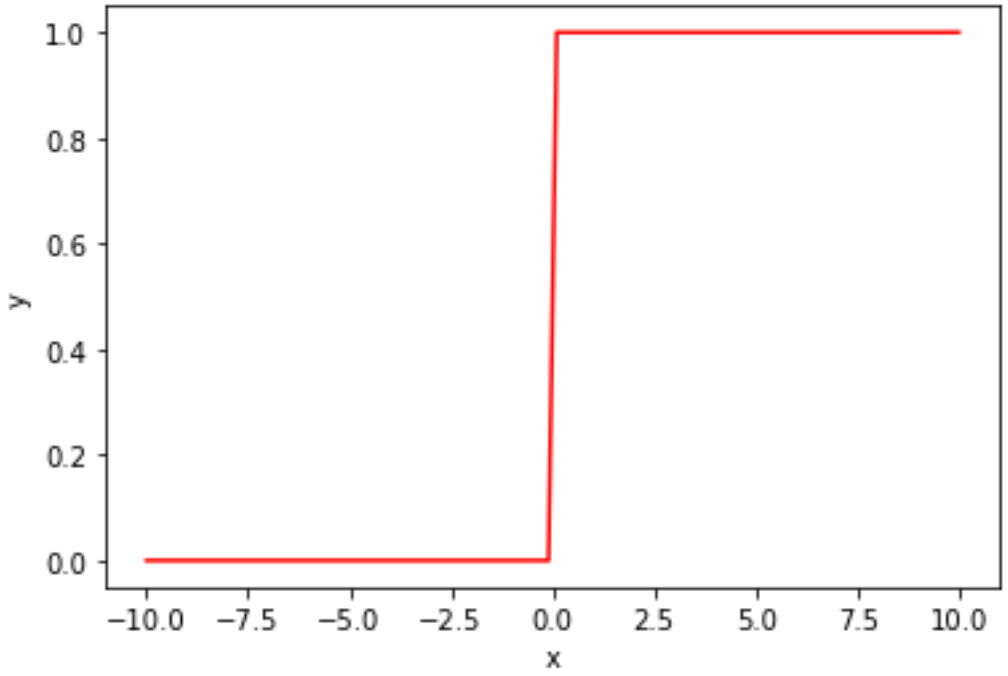
- **入力信号の総和を出力信号に変換する関数$h(x)$を活性化関数と呼ぶ**

入力信号の総和がどのように発火（活性化）するかを決定する役割

活性化関数の例

- ステップ関数(パーセプトロンの活性化関数)

$$h(x) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

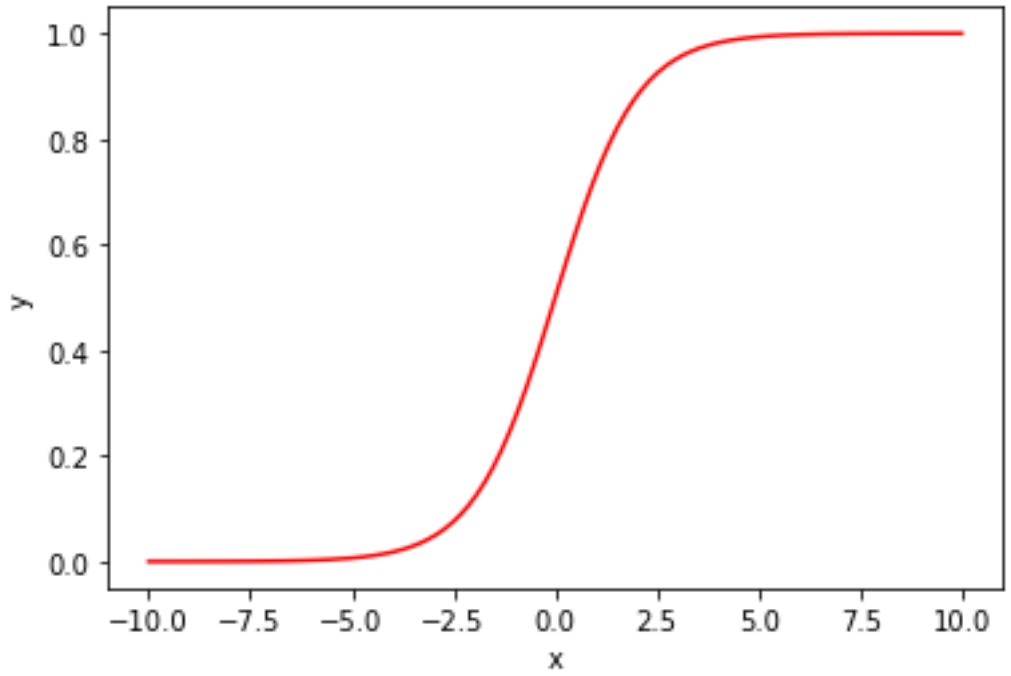


活性化関数の例

- シグモイド関数(ロジスティック関数とも呼ばれる)

$$h(x) = \frac{1}{1 + \exp(-x)}$$

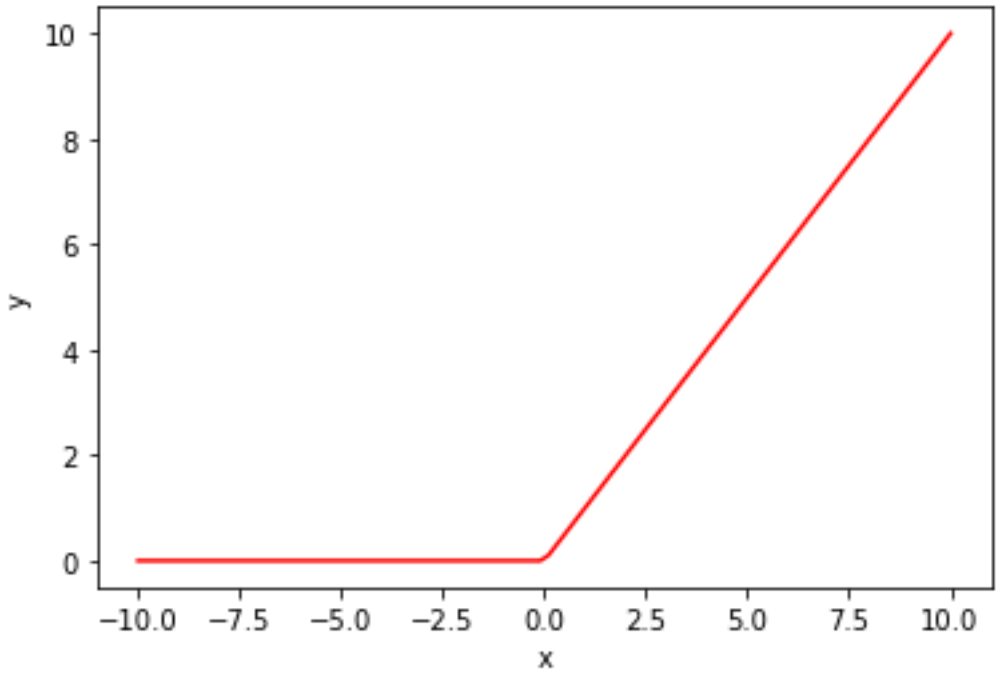
e^{-x}



活性化関数の例

- ReLU関数

$$h(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$



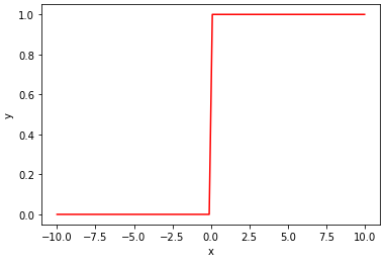
隠れ層の活性化関数の共通点

- **非線形関数**

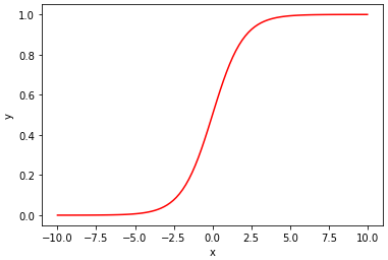
NNの隠れ層は非線形関数を活性化関数として用いないと深層化する意味がない！

仮に、線形関数$h(x) = cx$を活性化関数とすると、3層隠れ層を重ねた場合、その出力は、$y(x) = h\Big(h\big(h(x)\big)\Big) = c^3x$となる。
これは、$h(x) = ax$（**ただし、$a = c^3$**）を活性化関数とした1層の層と同等で、多層にするメリットなし

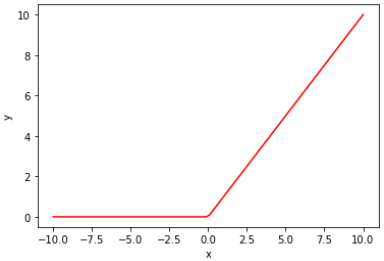
ステップ関数



シグモイド関数



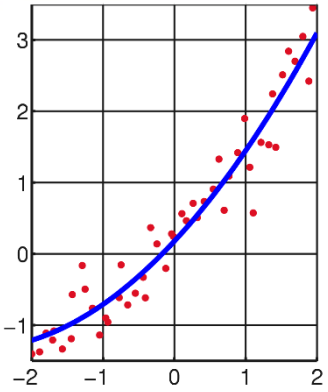
ReLU関数



出力層の活性化関数

関数: $y = f(\boldsymbol{x})$

回帰

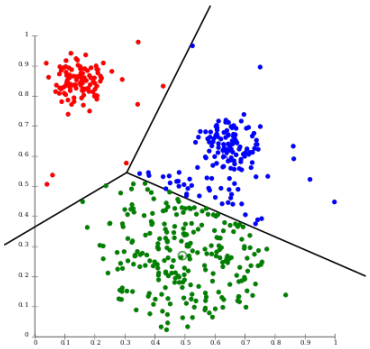


- **回帰問題 (regression)**
  - $y \in R$ (実数)

  例: 年齢予測、降水確率の予測、気温の予測
- **分類問題 (classification)**
  - $y \in \{C_1, C_2, \cdots, C_K\}$ (ラベル集合)

  例: 文書分類(政治、経済、スポーツ等)

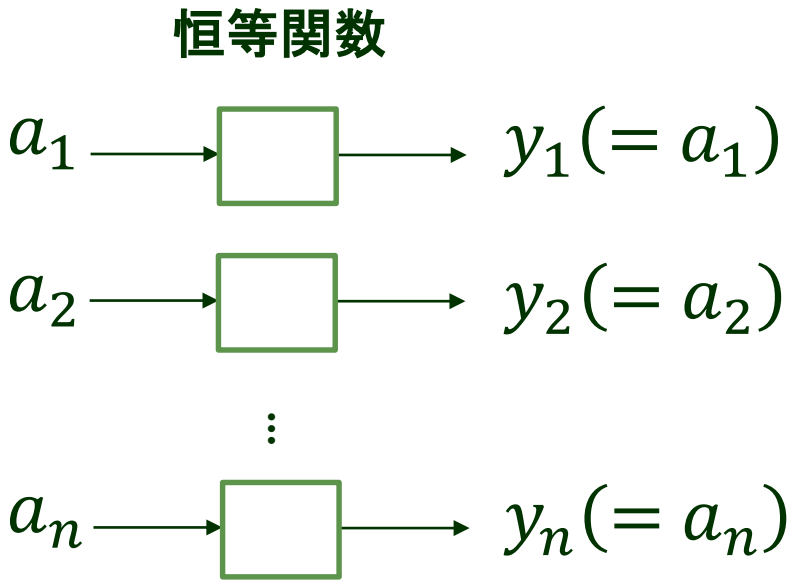
分類



回帰問題 → 恒等関数
分類問題 → ソフトマックス関数

回帰問題の活性化関数

- **恒等関数**
  - 入力をそのまま出力



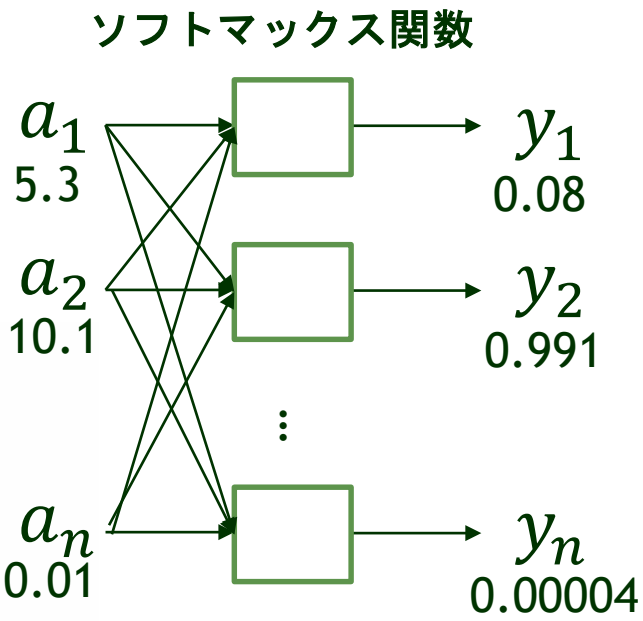
分類問題の活性化関数

- **ソフトマックス関数**
  - 自分以外のノードの出力も使って正規化

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

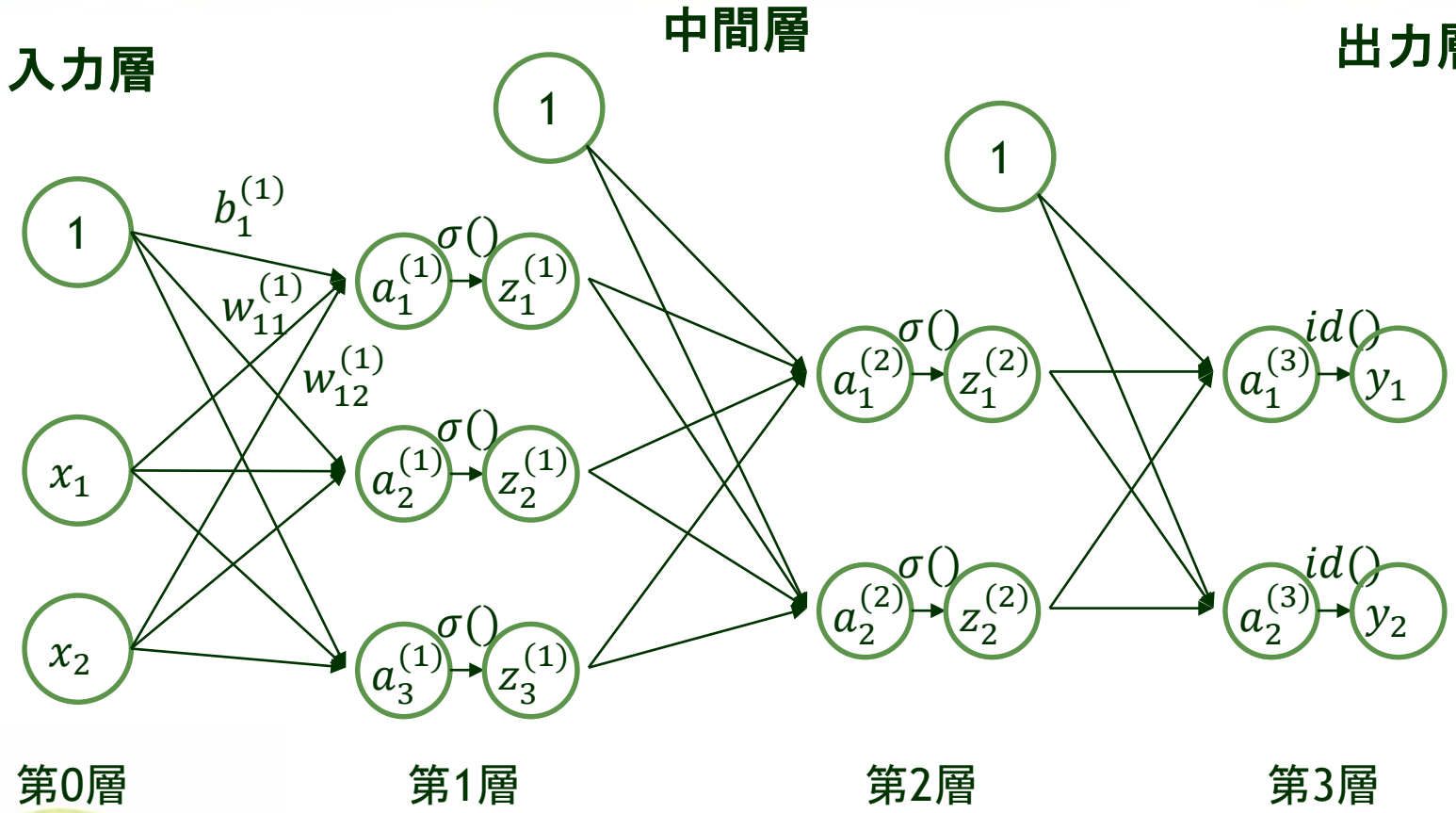
$$0 \leq y_k \leq 1 \ (1 \leq k \leq n)$$

$$\sum_{k=1}^{n} y_k = 1$$



出力層のノード数を「分類クラス数」にする
→ 各ノードの出力はそのクラスの分類確率として解釈可能
→ 分類確率の最も高いクラスに分類

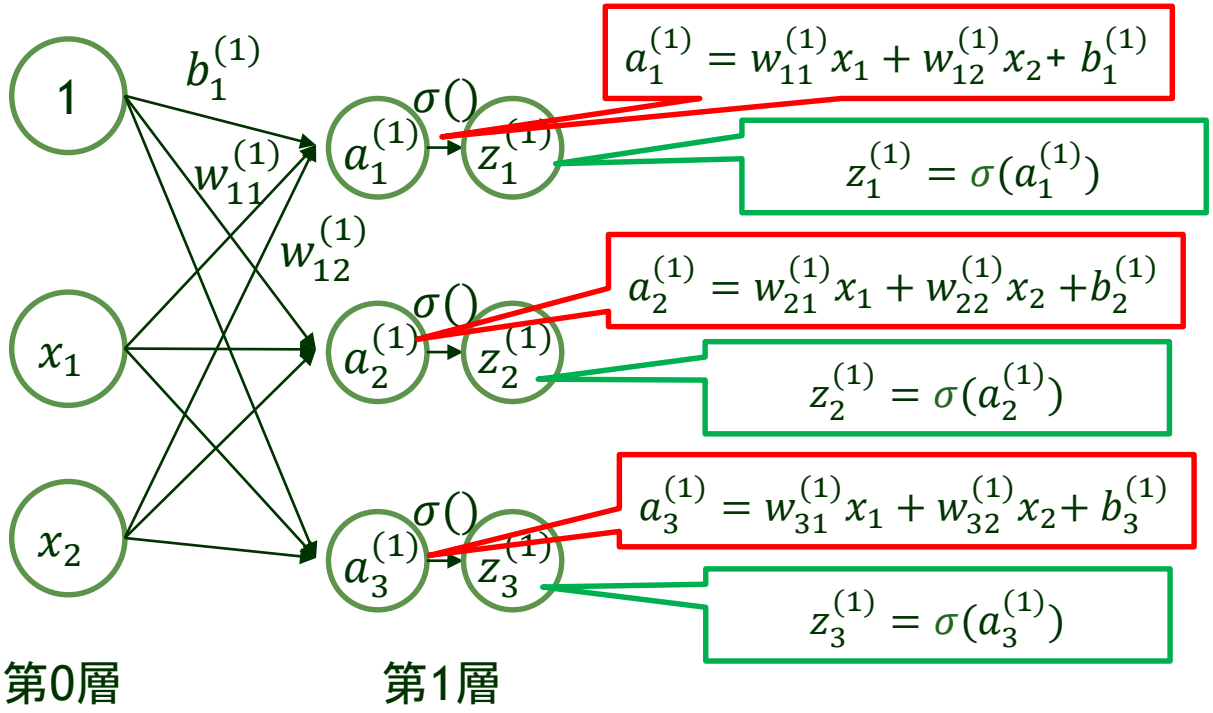
3層NNの推論



$\sigma()$：シグモイド関数
$id()$：恒等関数

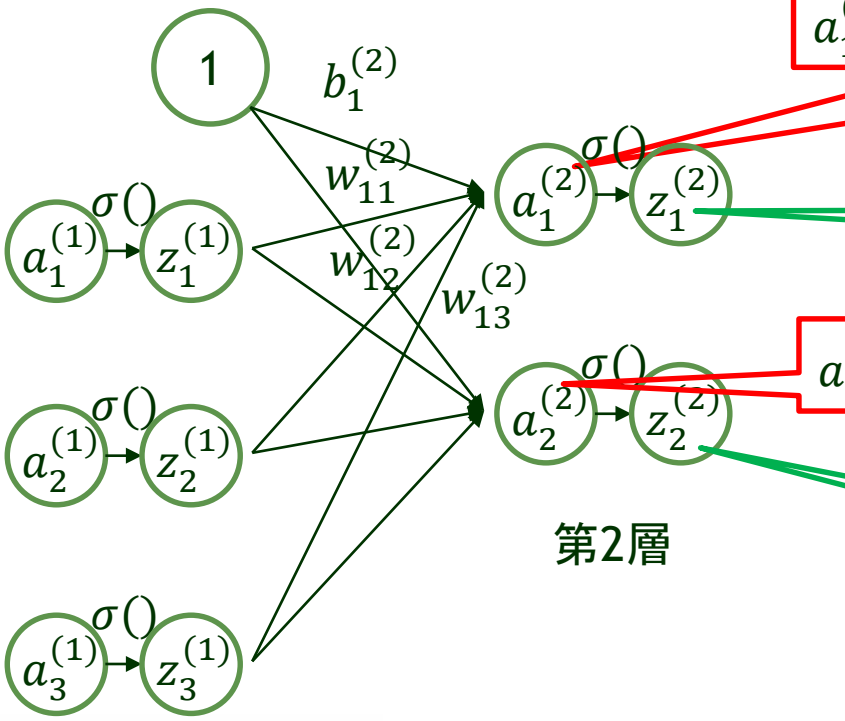
信号は低層から順に伝わる
→低層の演算から順に行う

1層目の動作



$\sigma()$：シグモイド関数

2層目の動作



$$a_1^{(2)} = w_{11}^{(2)} z_1^{(1)} + w_{12}^{(2)} z_2^{(1)} + w_{13}^{(2)} z_3^{(1)} + b_1^{(2)}$$

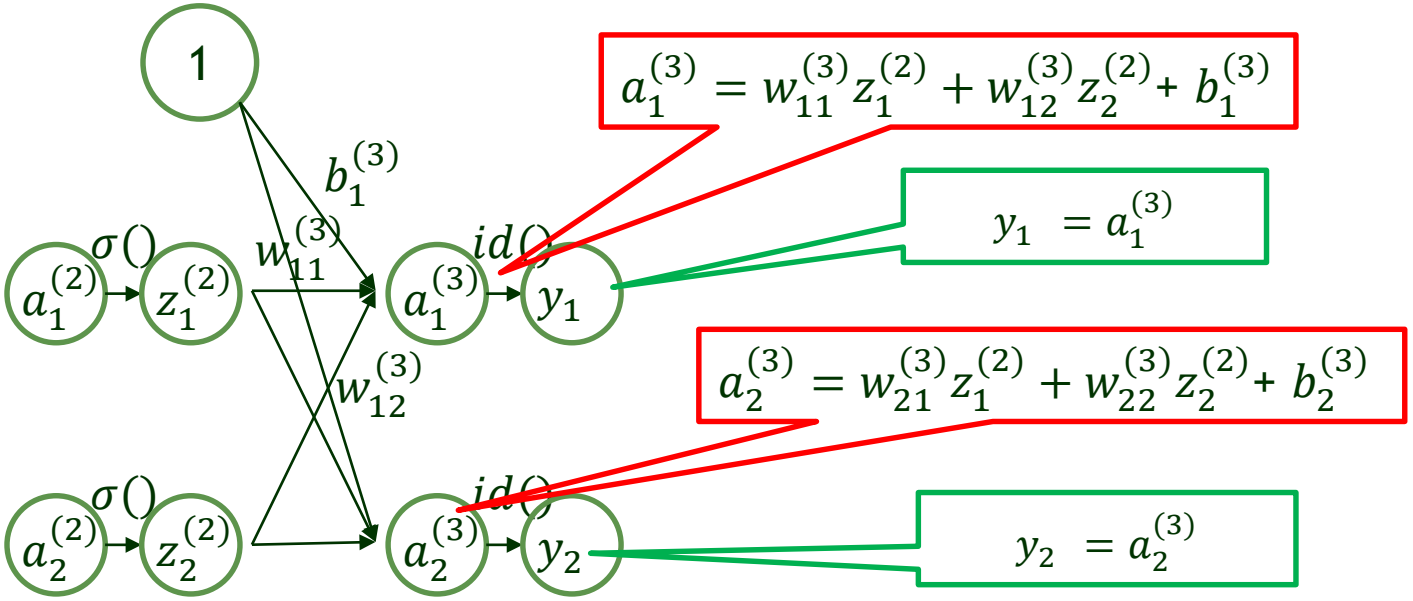
$$z_1^{(2)} = \sigma(a_1^{(2)})$$

$$a_2^{(2)} = w_{21}^{(2)} z_1^{(1)} + w_{22}^{(2)} z_2^{(1)} + w_{23}^{(2)} z_3^{(1)} + b_2^{(2)}$$

$$z_2^{(2)} = \sigma(a_2^{(2)})$$

$\sigma()$：シグモイド関数

3層目（出力層）の動作



$\sigma()$：シグモイド関数
$id()$：恒等関数

まとめ

- **NNの推論**
  - NNは入力層、中間層、出力層の多層構造
  - 信号は入力層、中間層、出力層へと階層的に伝わる
  - 中間層、出力層には活性化関数を用いる

深層学習 (2)

– ニューラルネットワークの学習、損失関数、勾配法 –

機械学習

- **データから学習**
  - 入力 $\boldsymbol{x} = (x_1, x_2, \cdots, x_m)$　←m次元ベクトル
    （画像の場合は、赤、青、緑の画像に対応する3つの行列）
  - データDは入力$\boldsymbol{x}$と出力yのペアの集合
  - 関数fはパラメータ(重み変数)の集合 $\boldsymbol{w} = (w_1, w_2, \cdots)$から成る計算式
    - NNも巨大な一つの関数
  - 学習 = データDに対する誤差を最小にする重み変数wを求める

データ全体の誤差 $L(\boldsymbol{w}) = \sum_{(\boldsymbol{x},y)\in D} \left(y - f_{\boldsymbol{w}}(\boldsymbol{x})\right)^2$ ← 損失関数

NNの学習：教師データに対する誤差が最小になる重み変数を探すこと

NNの学習

教師データに対する損失関数の値が最小になる
各ノードの重みを探すこと

$\mathrm{argmin}_w L(w)$　　$L(w)$：損失関数

平均2乗誤差　$L(w) = \frac{1}{2N}\sum_n\sum_k (y_{nk} - t_{nk})^2$

交差エントロピー誤差　$L(w) = -\frac{1}{N}\sum_n\sum_k t_{nk} log_e\, y_{nk}$

y_{nk}：n個目のデータのk次元目のNNの出力、
t_{nk}：教師データ
N　：教師データの個数

2乗和誤差

$$\frac{1}{2}\sum_k (y_k - t_k)^2$$

y_k：NNの出力、
t_k：教師データ

例)　$y = (13.5, 20.1, 0.7)$

$t = (12.1, 22.8, 0.3)$

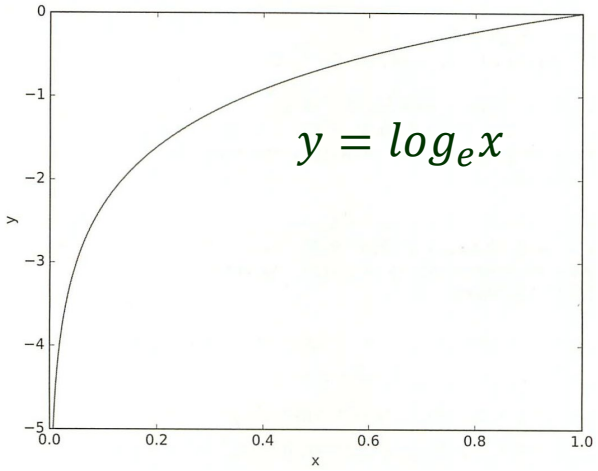
誤差$= \frac{1}{2} \times (1.4^2 + 2.7^2 + 0.4^2) = 4.705$

回帰問題の場合によく用いられる

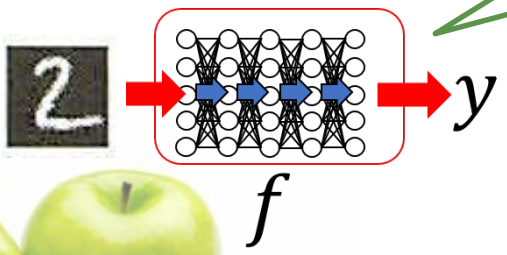
交差エントロピー誤差

$$-\sum_k t_k log_e y_k$$

y_k ：NNの出力、
t_k ：教師データ



（例）手書き数字認識



出力層の活性化関数：ソフトマックス関数
出力層のノード数：10

$y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]$

$t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$

誤差$= -log_e 0.6 = 0.51$

分類問題の場合によく用いられる

ミニバッチ学習

平均2乗誤差 $L = \frac{1}{2N}\sum_n\sum_k (y_{nk} - t_{nk})^2$

交差エントロピー誤差 $L = -\frac{1}{N}\sum_n\sum_k t_{nk} log_e y_{nk}$

y_{nk}：n個目のデータのk次元目のNNの出力、
t_{nk}：教師データ、
N　：教師データの個数

- **バッチ学習：教師データ中の全データに対する損失関数を利用**
  - $N = |D|$：教師データ中の全データ数

データが増えると計算大変

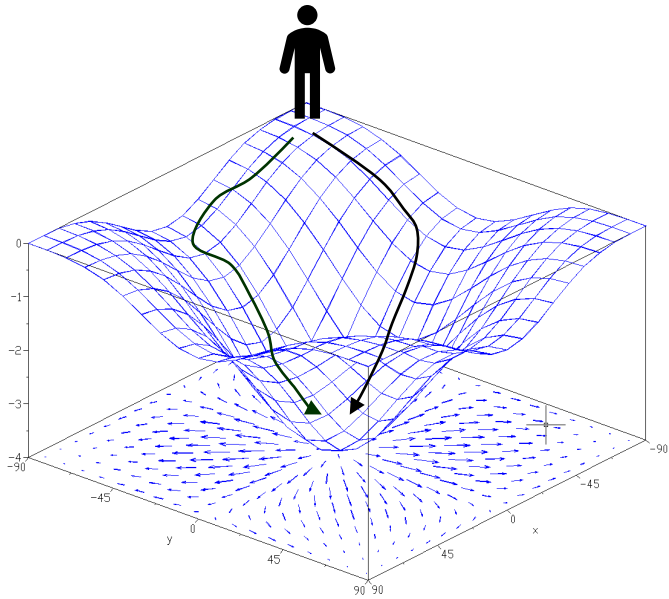
- **ミニバッチ学習：教師データ中の一部のデータに対する損失関数を利用**
  - $N = M (M < |D|)$：バッチサイズ

誤差関数の最小化

- **誤差関数を最小化するには数値最適化の手法を用いる**
  - ニュートン法、準ニュートン法
  - 最急降下法
  - オンライン学習(確率的勾配法(SGD), AdaGrad, Adamなど)

- **勾配を用いて計算する手法が多い**
  - 勾配

$$\nabla L = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \cdots, \frac{\partial L}{\partial w_m} \right)$$

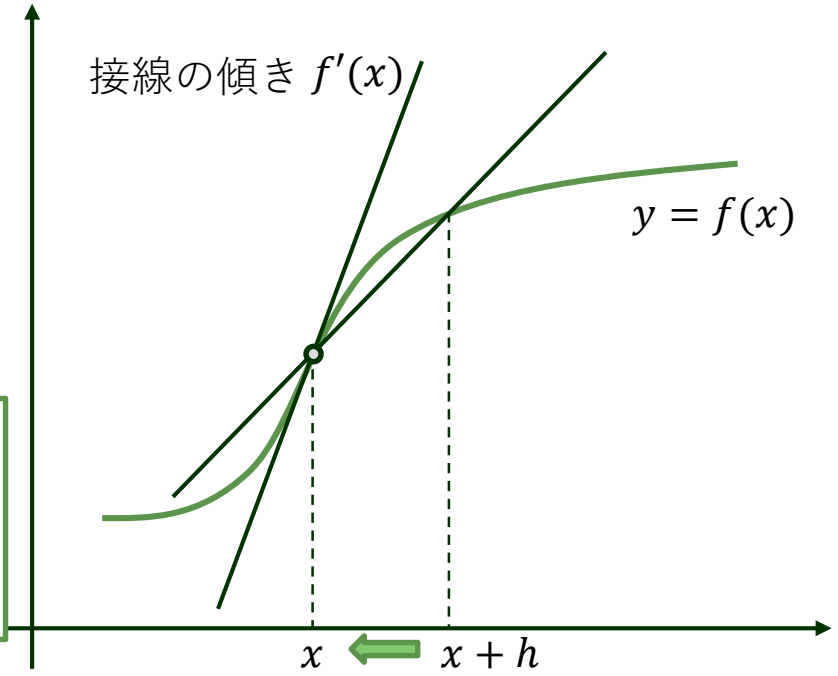


微分

- xに対する$f(x)$の変化（接線の傾き）

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

符号：変化の方向
大きさ：変化の度合い



実装

```
def numerical_diff(f, x):
    h = 1e-4
    return (f(x+h) - f(x)) / h
```

偏微分 (1/2)

- **変数が複数ある場合の微分**
  - 特定の変数に着目して微分を計算（その他の変数は定数とみなす）

例：$f(x_0, x_1) = {x_0}^2 + {x_1}^2$、$x_0 = 3$、 $x_1 = 4$のときの各変数に対する偏微分

解析的に解く

$$\frac{\partial f}{\partial x_0} = 2x_0 \qquad \frac{\partial f}{\partial x_1} = 2x_1$$

$$\frac{\partial f}{\partial x_0} = 6 \text{（} x_0 = 3 \text{のとき）} \qquad \frac{\partial f}{\partial x_1} = 8 \text{（} x_1 = 4 \text{のとき）}$$

勾配（gradient）(1/2)

- **全ての変数の偏微分をベクトルとしてまとめたもの**

例：　関数$L(w_0, w_1)$の勾配

$$\nabla L = \left(\frac{\partial L}{\partial w_0}, \frac{\partial L}{\partial w_1} \right)$$

$L(w_0, w_1) = {w_0}^2 + {w_1}^2$の$w_0 = 3$、 $w_1 = 4$における勾配は$(6, 8)$

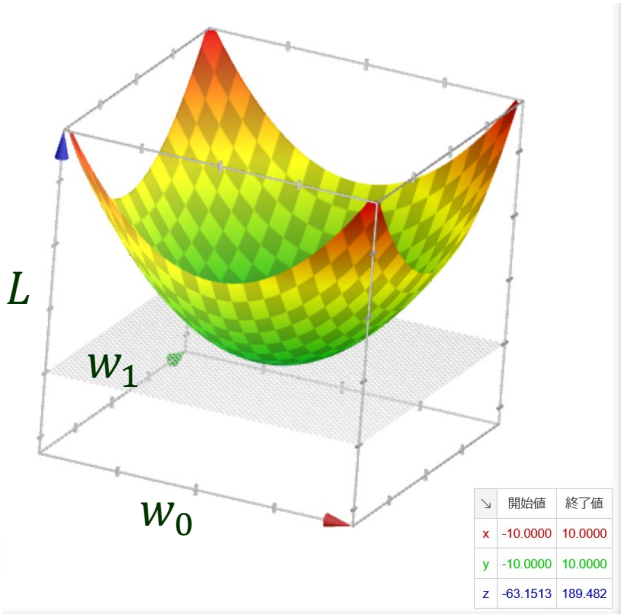
$$\nabla L = \left(\frac{\partial L}{\partial w_0}, \frac{\partial L}{\partial w_1} \right) = (2w_0, 2w_1)$$

$$\nabla L(3,4) = (6,8)$$

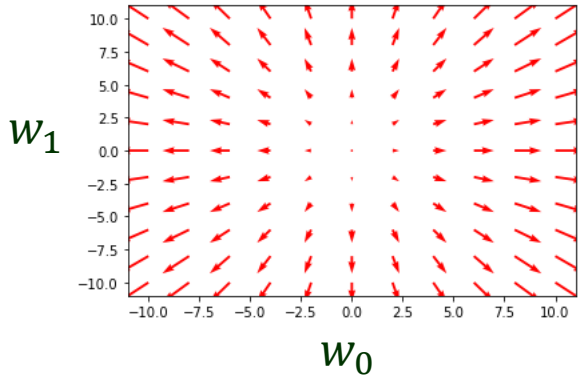
勾配(gradient)(2/2)

- **その点における傾斜方向を表す**

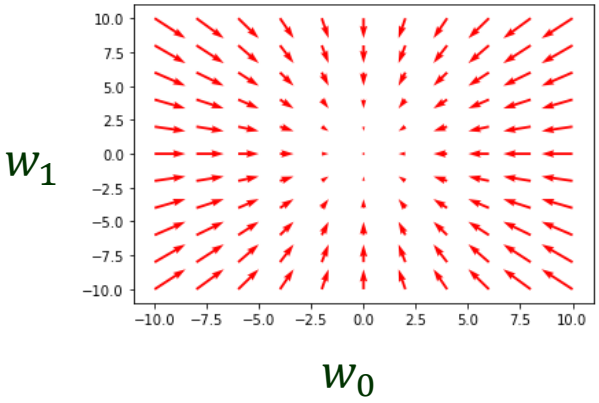
$$L(w_0, w_1) = {w_0}^2 + {w_1}^2$$



各点における勾配



各点における勾配に-1をかけたベクトル



勾配に-1をかけたベクトルは各場所で関数の値を最も減らす方向を示す！！

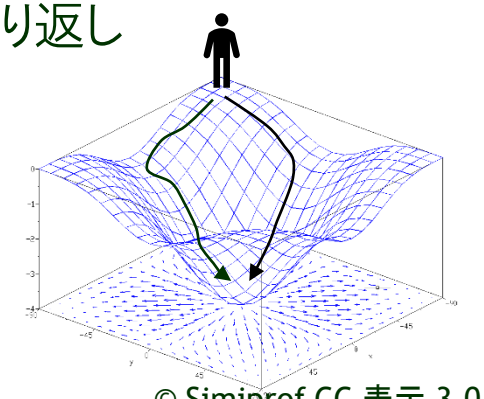
勾配降下法(Gradient Descent)

- **関数の勾配を利用して最小値となる所を見つける方法**
  - 「現在の場所から勾配方向に一定の距離進む」動作を繰り返して関数の値を徐々に減らす

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla L(\boldsymbol{w})$$

η:学習率

←この計算を繰り返す



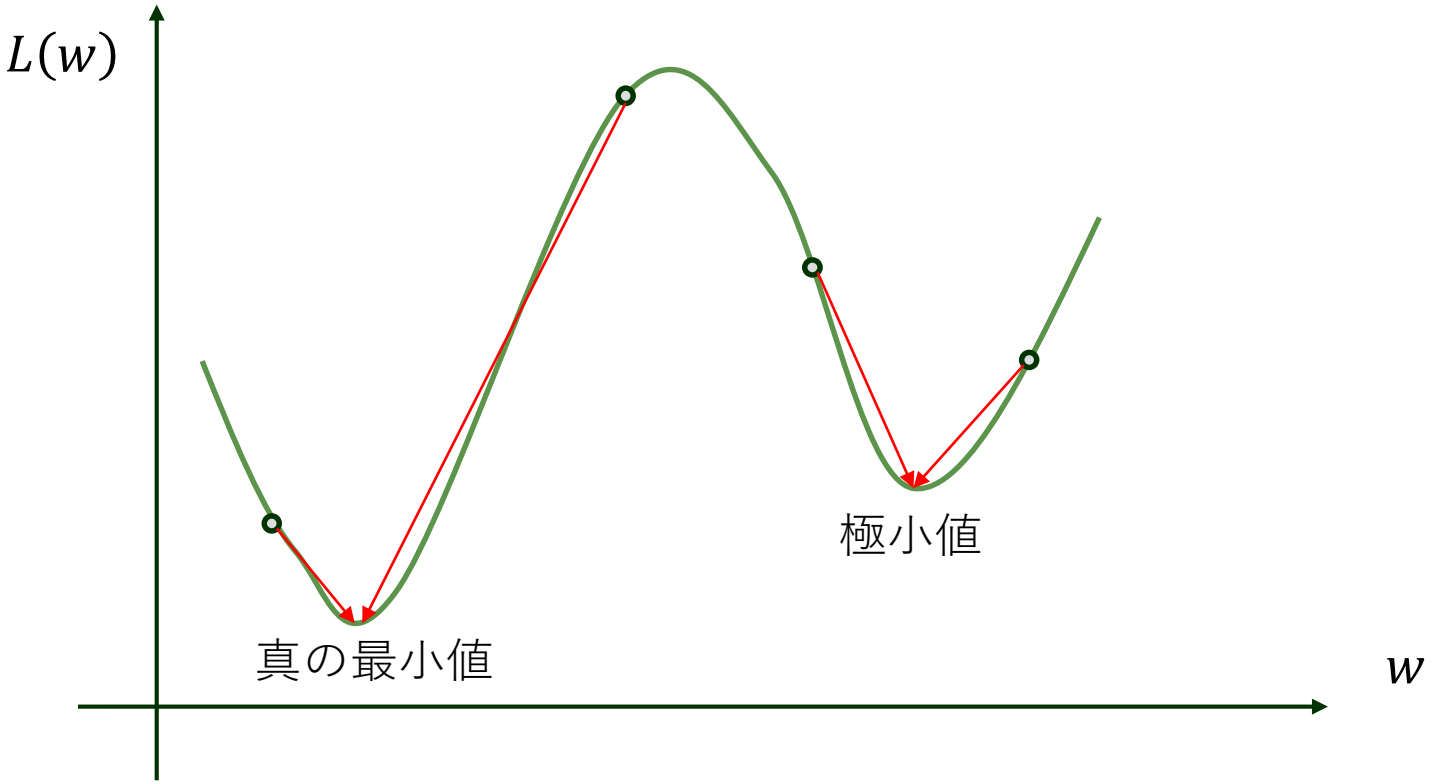
ステップ0：初期値の設定
　各パラメータの初期値を設定
ステップ1：勾配の算出
　最小化する関数に基づき、パラメータの勾配を求める
ステップ2：パラメータの更新
　パラメータを勾配方向とは逆向きに微小量更新する
ステップ3：繰り返す
　ステップ1に戻る

$\nabla L(\boldsymbol{w})$

$-\eta \nabla L(\boldsymbol{w})$

勾配降下法(Gradient Descent)

- 注意点：勾配のさし先が本当の最小値とは限らない。その場合、極小値(局所的な最小値)となる。



確率的勾配降下法 (Stochastic Gradient Descent; SGD)

- **ミニバッチ学習版の勾配降下法**
  - 損失関数の計算に教師データの一部を使用する

- **NNのSGDによる学習**
  1. ミニバッチ：教師データの中からランダムにミニバッチサイズ分のデータを選ぶ
  2. 勾配の算出：ミニバッチの損失関数に基づき、各重みパラメータの勾配を求める
  3. パラメータの更新：重みパラメータを勾配方向とは逆方向に微小量更新する
  4. 繰り返す：ステップ1に戻る

$$\boldsymbol{w} = \boldsymbol{w} - \eta \frac{\partial L}{\partial \boldsymbol{w}}$$

【参考】SGDにおける繰り返し

- **エポック数：全ての教師データを何回繰り返したかを表す表現**
  - 教師データが1,000個で、バッチサイズを100とした場合、ミニバッチSGDを10回繰り返すと全ての教師データを見たことに相当する

    → 10回の繰り返し=1エポック
- **「教師データD個、ミニバッチサイズMのSGDでNエポック学習した」**

→「 (D/M)×N回パラメータを更新した」ということ

学習した関数の評価

- **未知のデータに対する推論性能(汎化能力、汎化性能)が重要**

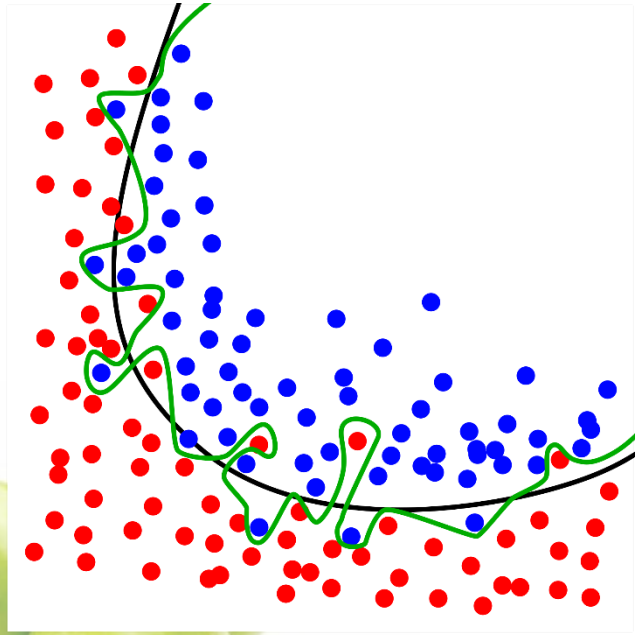
→ 教師データとは違う評価データ(未知のデータ)で性能評価

- 過学習の問題があるため

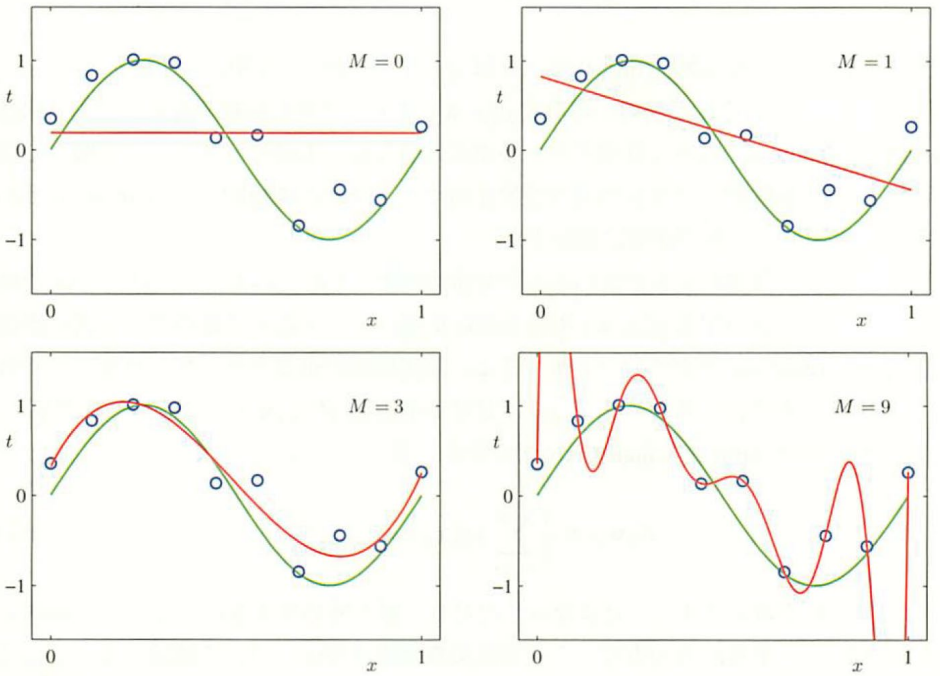
教師データに対する性能で評価してはダメ！

過学習の問題

- **過学習の問題**
  - 関数が複雑すぎると過剰に適合してしまいうまく学習できない



$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$



Bishop, Pattern Recognition and Machine Learningより引用

まとめ

- **機械学習の基礎**
  - データから入力と出力をつなぐ関数を学習する手法
  - 問題の種類
    - 回帰問題、分類問題、構造予測
  - 損失関数を最小化することで重み変数(パラメータ)を学習
    - 回帰問題→平均2乗誤差
    - 分類問題→交差エントロピー誤差
  - 過学習
    - 訓練データに過剰に適合してしまうこと
  - 学習の方法
    - 損失関数の勾配を計算して勾配方向に重みベクトルを更新
    - 勾配降下法、SGD

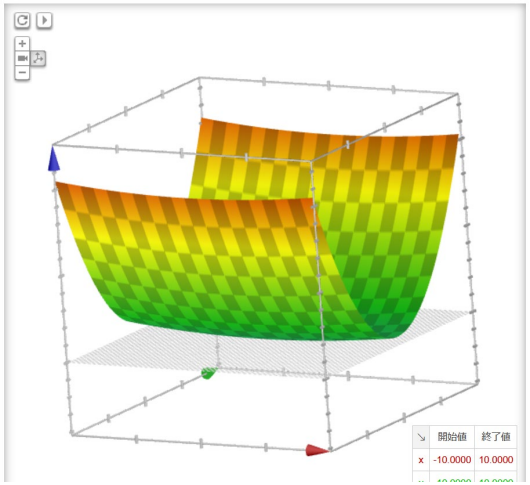
(参考) SGDの欠点

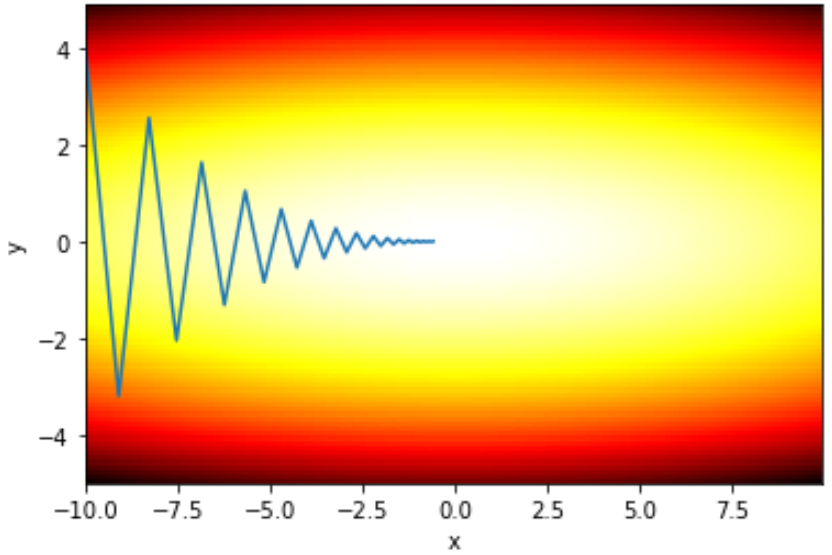
- **勾配の方向が本来の最小値ではない方向を指しているときは非効率な場合がある**

例：

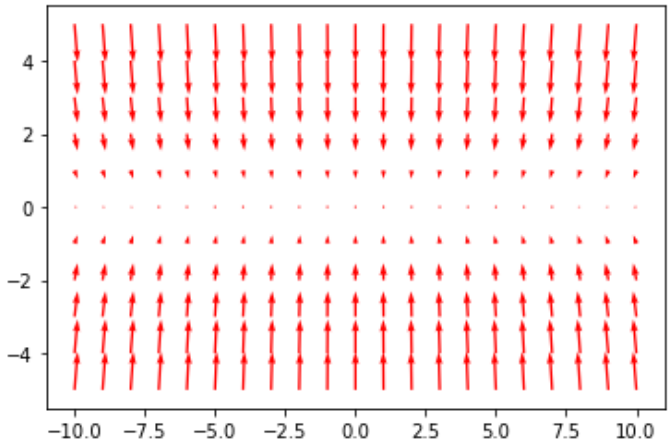
$$f(x,y) = \frac{1}{20}x^2 + y^2$$

最小値(0,0)





勾配



$$\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \frac{\partial L}{\partial \boldsymbol{w}}$$

ジグザグで最小値に近づく
→非効率

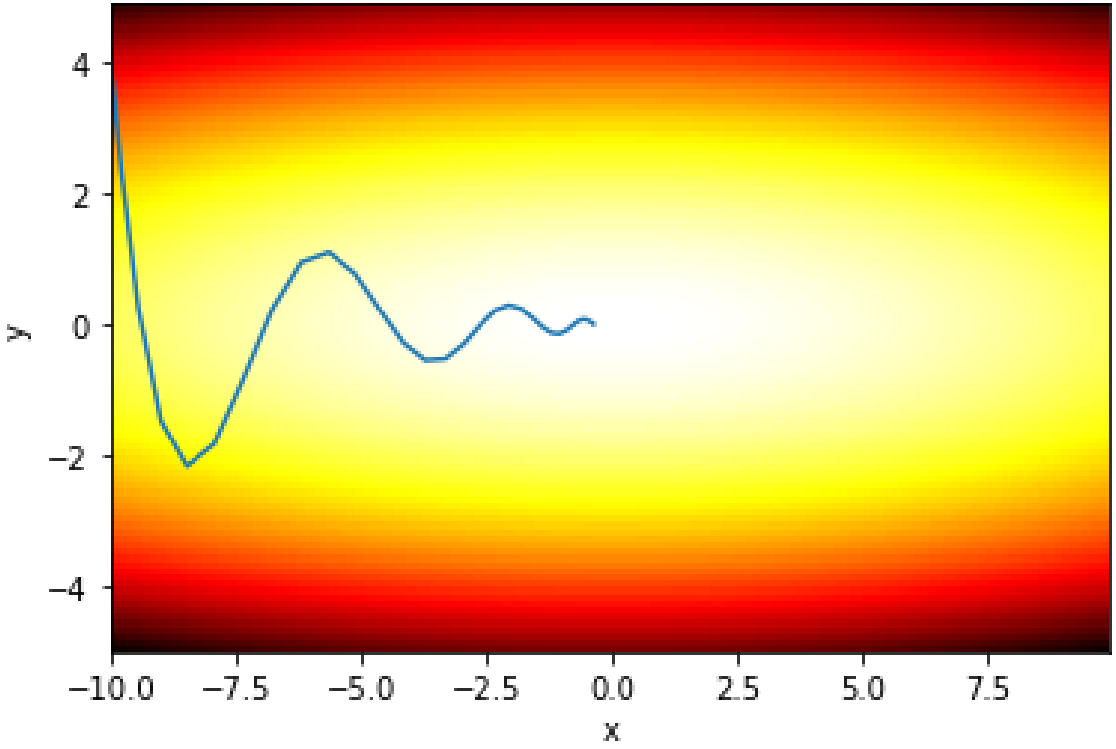
(参考) 最適化手法:Momentum

$$\boldsymbol{v} = \alpha \boldsymbol{v} - \eta \cdot \frac{\partial L}{\partial \boldsymbol{w}}$$

$$\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{v}$$

$\boldsymbol{v}$：速度の役割
（勾配方向に速度が加算される）

α：ハイパーパラメータ(0.9等)



・x軸の勾配の方向は一定
→x軸方向には加速される
・y軸の勾配の方向は交互
→y軸方向への動きは抑制

⇒ジグザグの動きが軽減

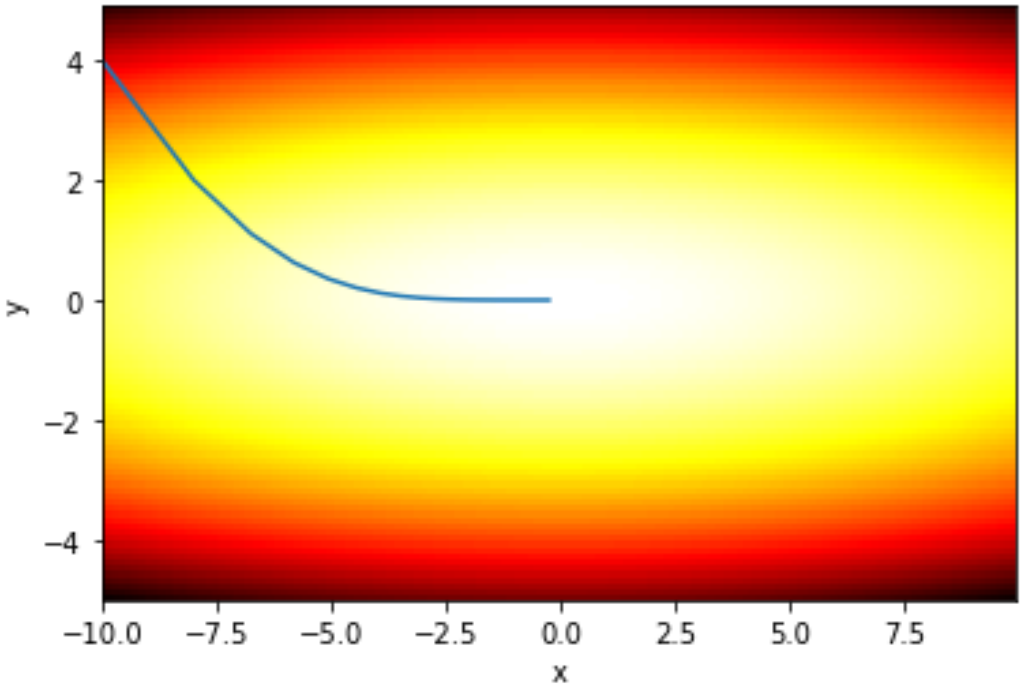
(参考) 最適化手法: AdaGrad

- **パラメータ毎に学習率をコントロール**
  - 最初は大きく次第に小さく
  - 大きく更新されたパラメータほど小さくする

$$\boldsymbol{h} = \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{w}} \otimes \frac{\partial L}{\partial \boldsymbol{w}}$$

$$\boldsymbol{w} = \boldsymbol{w} - \eta \frac{1}{\sqrt{\boldsymbol{h}}} \otimes \frac{\partial L}{\partial \boldsymbol{w}}$$

$\otimes$: 要素毎の積



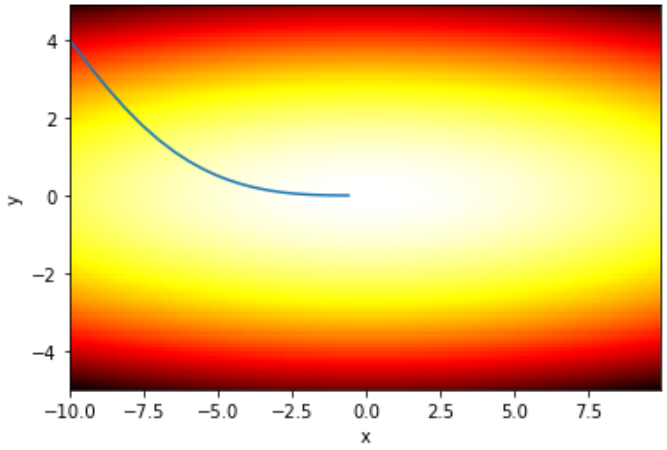
Y軸方向の勾配は大きい
→最初は大きく更新されるが
　更新が進むほど更新度合いが弱まる

⇒ジグザグの動きが軽減

(参考) その他の最適化手法

- Adadelta
  - AdaGradの発展形

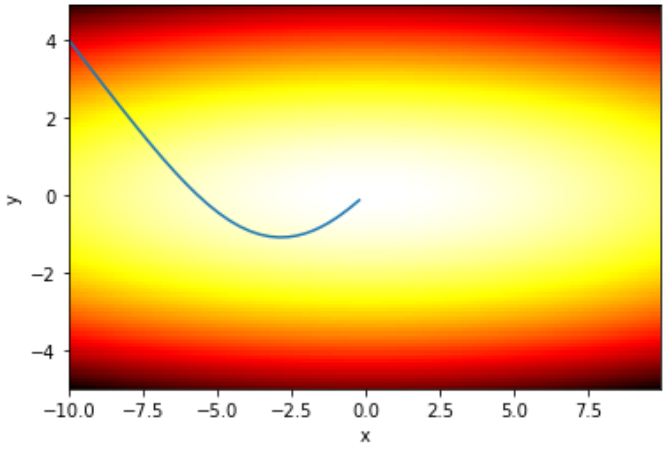
※ Matthew Zeiler, ADADELTA: An Adaptive Learning Rate Method, 2012.



- Adam
  - AdaGradのアイデア＋Momentumのアイデアの融合

※ Diederik Kingma and Jimmy Ba, Adam: a Method for Stochastic Optimization, 2015.

など



(参考) 重みの初期値

- **活性化関数がシグモイド関数の場合**
  - Xavierの初期値：$\frac{1}{\sqrt{n}}$を標準偏差とするガウス分布で各ノードの重みを初期化（n：前層のノード数）

  Xavier Glorot and Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, 2010.

- **活性化関数がReLUの場合**
  - Heの初期値：$\frac{2}{\sqrt{n}}$を標準偏差とするガウス分布で各ノードの重み初期化（n：前層のノード数）

  Kaiming He, Xiangyu Zhang, Shaoqing Re, and Jian Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.

(参考) 過学習への対応法：荷重減衰 (Weight Decay)

- **重みのL2ノルムを正則化項として加えた損失関数に基づき学習**
  - Weight Decayを用いた損失関数

$$L(\boldsymbol{w}) + \frac{1}{2}\lambda\|\boldsymbol{w}\|_2^2$$

λ：ハイパーパラメータ

$\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$に対するL2ノルムは

$$\|\boldsymbol{w}\|_2 = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$$

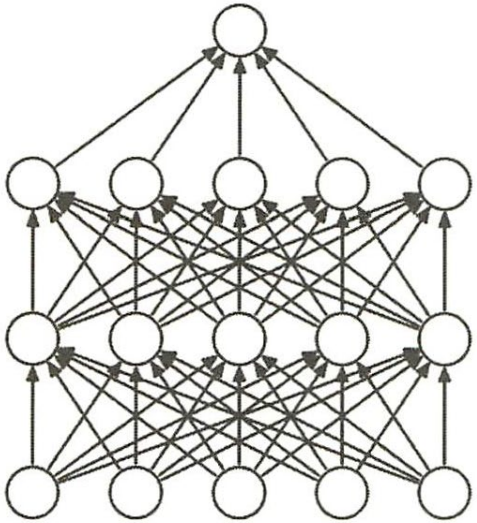
従って、

$$\|\boldsymbol{w}\|_2^2 = w_1^2 + w_2^2 + \cdots + w_n^2$$

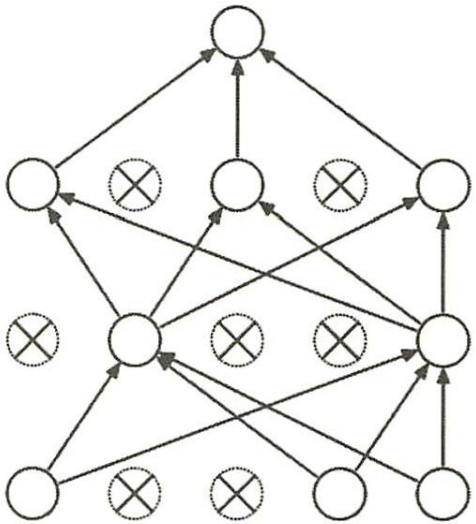
(参考) 過学習への対応法：Dropout

- 学習時にはデータが流れるたびに隠れ層のノードをある割合でランダムに選択し消去して学習
- テスト時には各層の出力に対して訓練時に消去した割合を乗算した値を出力

N. Srivastava, G. Hinton, A. Krizhevsky, I.sutskever, and R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, 2014.



Dropoutなし



Dropoutあり

(参考) Batch Normalization

- **ミニバッチ毎に入力データを正規化**

ミニバッチ：$\{x_1, x_2, \ldots x_m\}$
ε：小さな値(10e-7等)

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu)^2$$

$$\widehat{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

- **効果**
  - 学習が速くなる
  - 初期値への依存度を減らせる
  - 過学習を抑制する

Sergey Ioffe and Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

深層学習 (3)

– 誤差逆伝搬法、計算グラフ –

誤差逆伝搬法（Back Propagation）

- **勾配(損失関数に対する各パラメータの微分)をどうやって計算するのか？**

- **誤差逆伝播法**
  - 次の計算ステップで全てのパラメータの微分を計算

1. 計算グラフを作る
2. 入力から出力に向かって順方向に計算(順伝搬)
3. 出力から入力に向かって逆方向に微分を計算(逆伝搬)

  - 同じ微分の計算を繰り返さない(動的計画法の一種)

多変数の連鎖律

- **多変数から成る合成関数の微分に関するルール**

$z = f(u, v, w)$**のとき**

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial x} + \frac{\partial z}{\partial w}\frac{\partial w}{\partial x}$$

関数fに対しu, v, wについてのみ偏微分すれば良い。

u, v, wに対して再帰的にxについて微分すれば良い。

例：
$$z = uv$$
$$u = x^2 + y^2$$
$$v = \sin xy$$

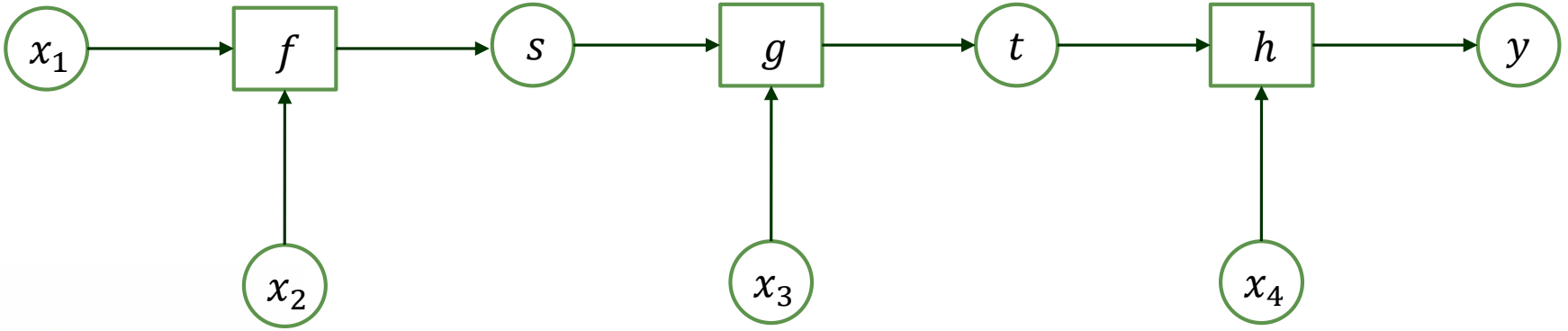
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial x} = v\frac{\partial u}{\partial x} + u\frac{\partial v}{\partial x} = v \cdot 2x + u \cdot \cos xy \cdot y$$

計算グラフによる解釈

- **計算グラフ：計算の過程をグラフ（ノードとエッジ）で表したもの**

○：変数
□：関数（演算の中身）

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ

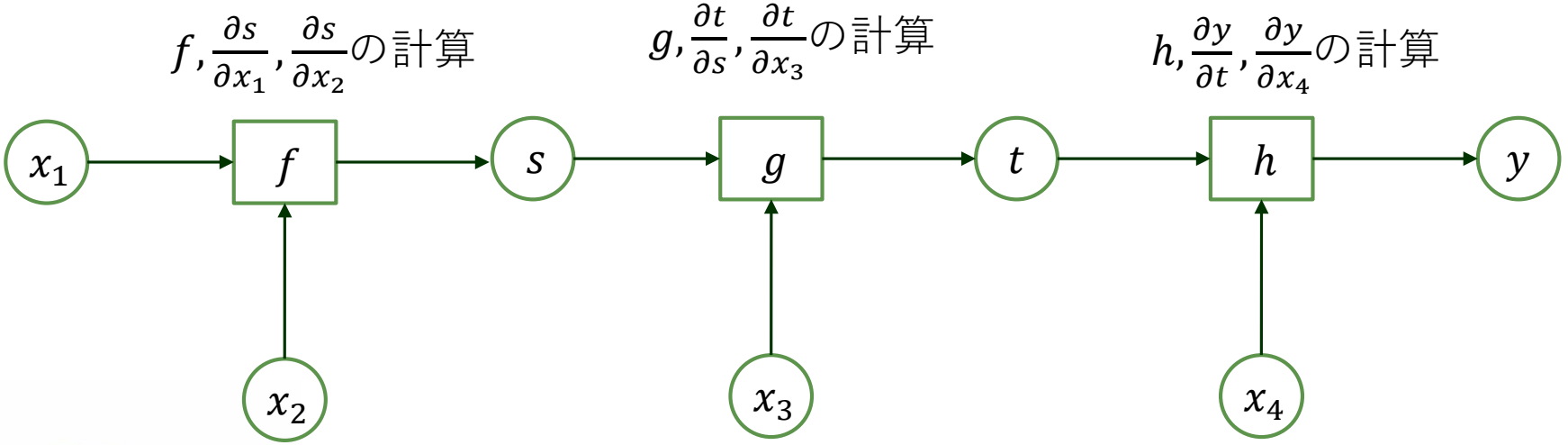


計算グラフによる解釈

- **計算グラフ：計算の過程をグラフ（ノードとエッジ）で表したもの**

○：変数
□：関数（演算の中身）

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ



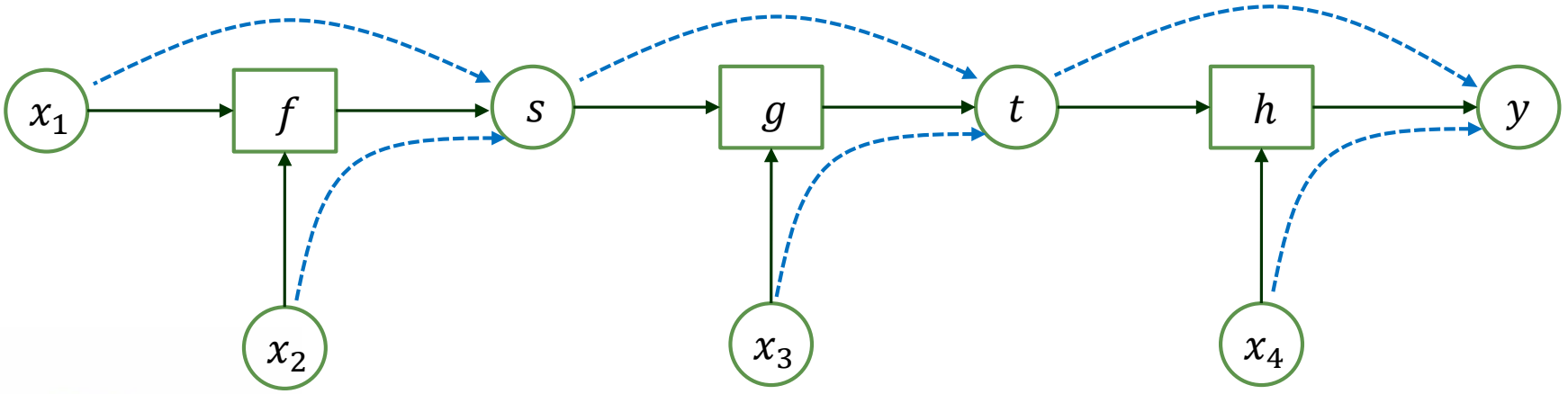
各関数は
・入力を受け取ったときの出力の計算
・入力となる各変数に対する偏微分の計算
ができれば良い

計算グラフによる解釈

- **順伝搬の計算**

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ

○：変数
□：関数（演算の中身）



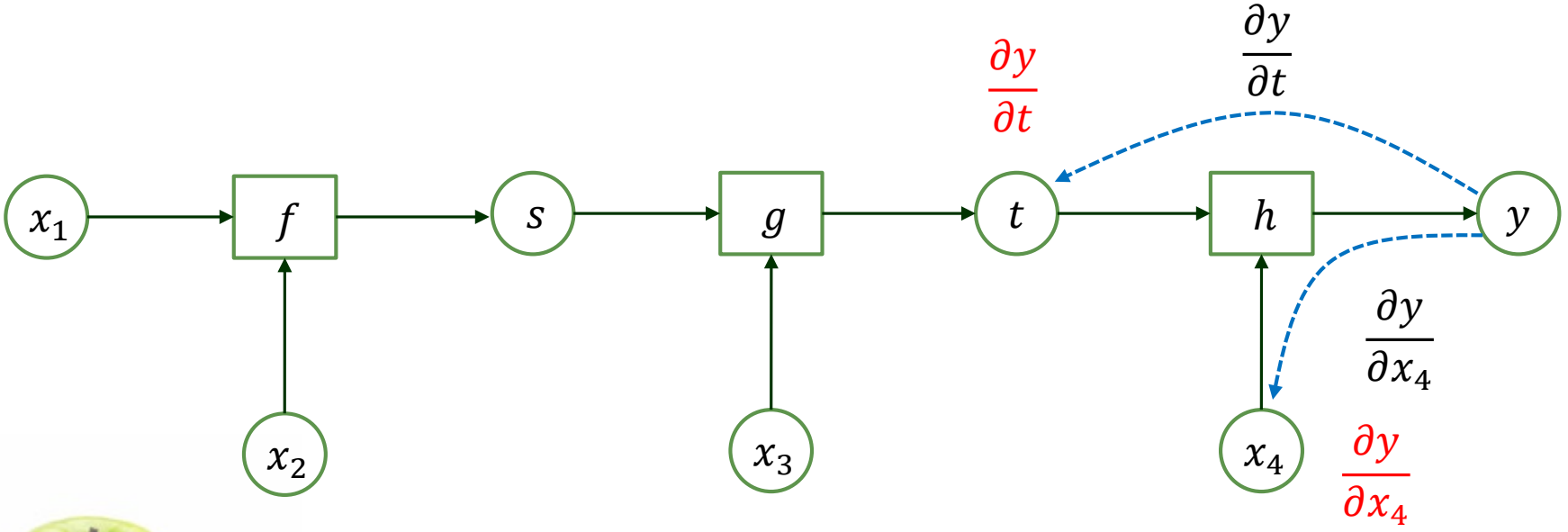
全体の入力(x_1、 x_2、 x_3、x_4)を与えて、すべてのノードの値(s, t, y)を計算する。

計算グラフによる解釈

- **逆伝搬の計算**

○：変数
□：関数（演算の中身）

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ



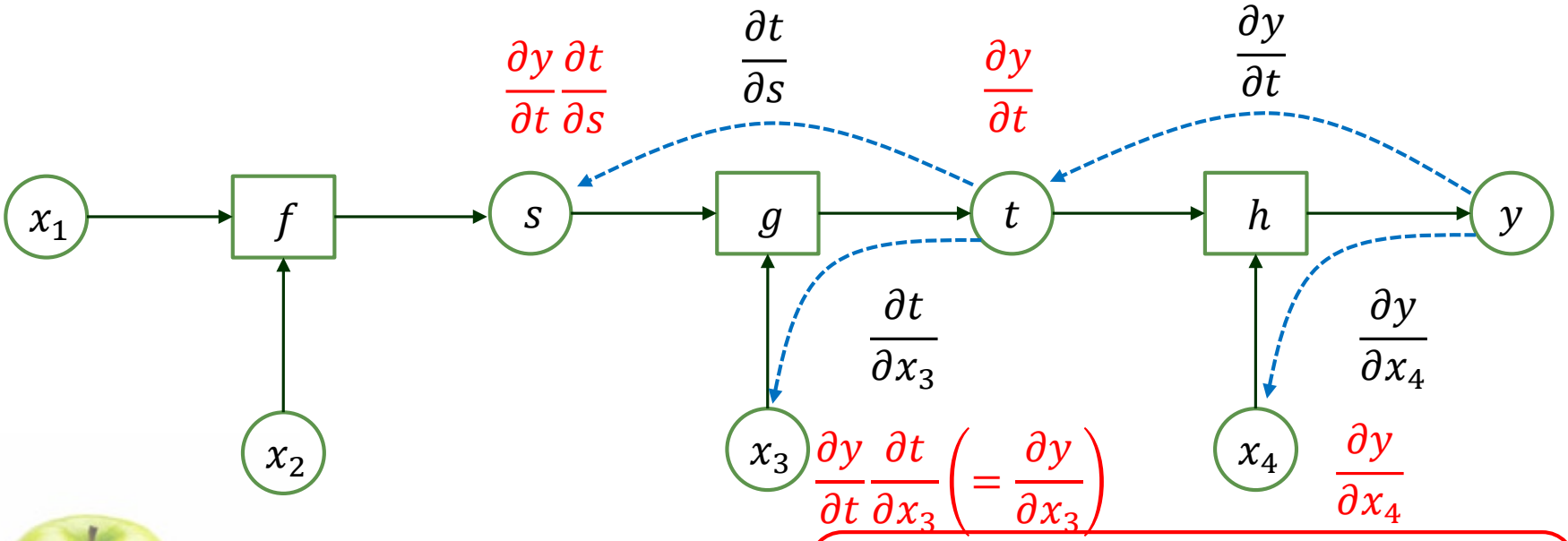
誤差逆伝搬法：上流の関数から伝達された値に局所的な微分を乗算して前のノードへ渡していく

計算グラフによる解釈

- **逆伝搬の計算**

○：変数
□：関数（演算の中身）

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ



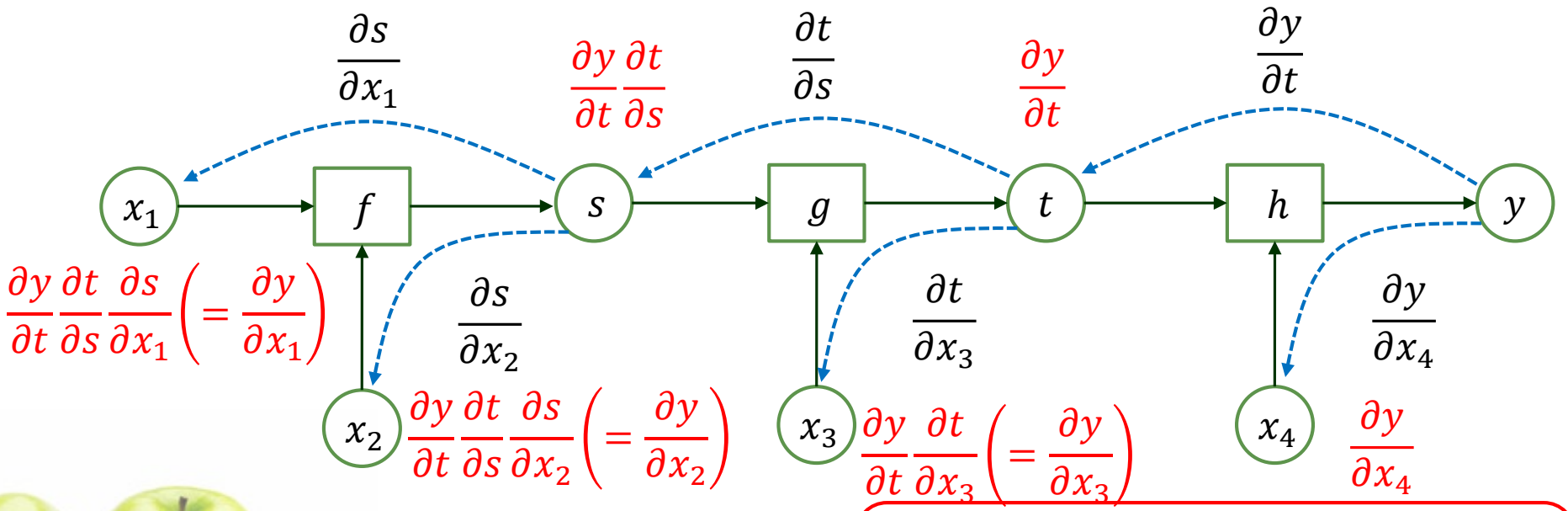
誤差逆伝搬法：上流の関数から伝達された値に局所的な微分を乗算して前のノードへ渡していく

計算グラフによる解釈

- **逆伝搬の計算**

○：変数
□：関数（演算の中身）

$s = f(x_1, x_2), t = g(s, x_3), y = h(t, x_4)$の計算グラフ



yの勾配$\left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \frac{\partial y}{\partial x_3}, \frac{\partial y}{\partial x_4}\right)$の計算できた!

誤差逆伝搬法：上流の関数から伝達された値に局所的な微分を乗算して前のノードへ渡していく

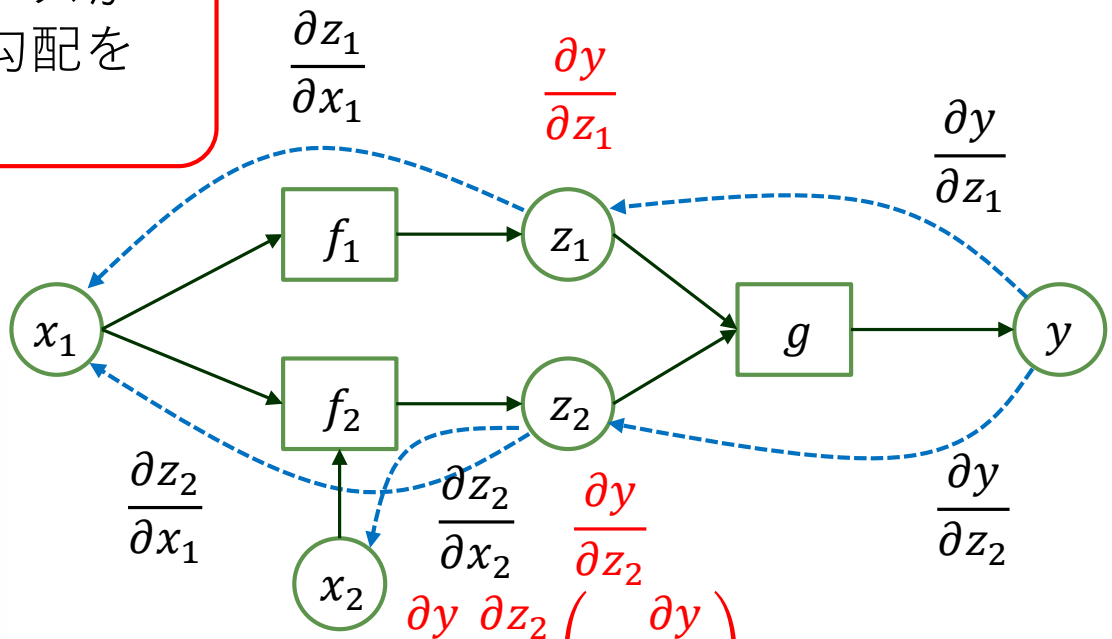
計算グラフによる解釈2

$$z_1 = f_1(x_1), z_2 = f_2(x_1, x_2), y = g(z_1, z_2)$$

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial x_1}, \qquad \frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial x_2}$$

合流地点では各パスから伝搬してきた勾配を足し合わせる

$$\frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial x_1} \left(= \frac{\partial y}{\partial x_1}\right)$$



誤差逆伝搬法によるNN学習の全体像

損失関数の値を最小にする重みパラメータを探す

1. 勾配の算出：損失関数に基づき、各重みパラメータの勾配を求める（誤差逆伝搬法により算出）
2. パラメータの更新：重みパラメータを勾配方向とは逆方向に微小量更新する
3. 繰り返す：ステップ1に戻る

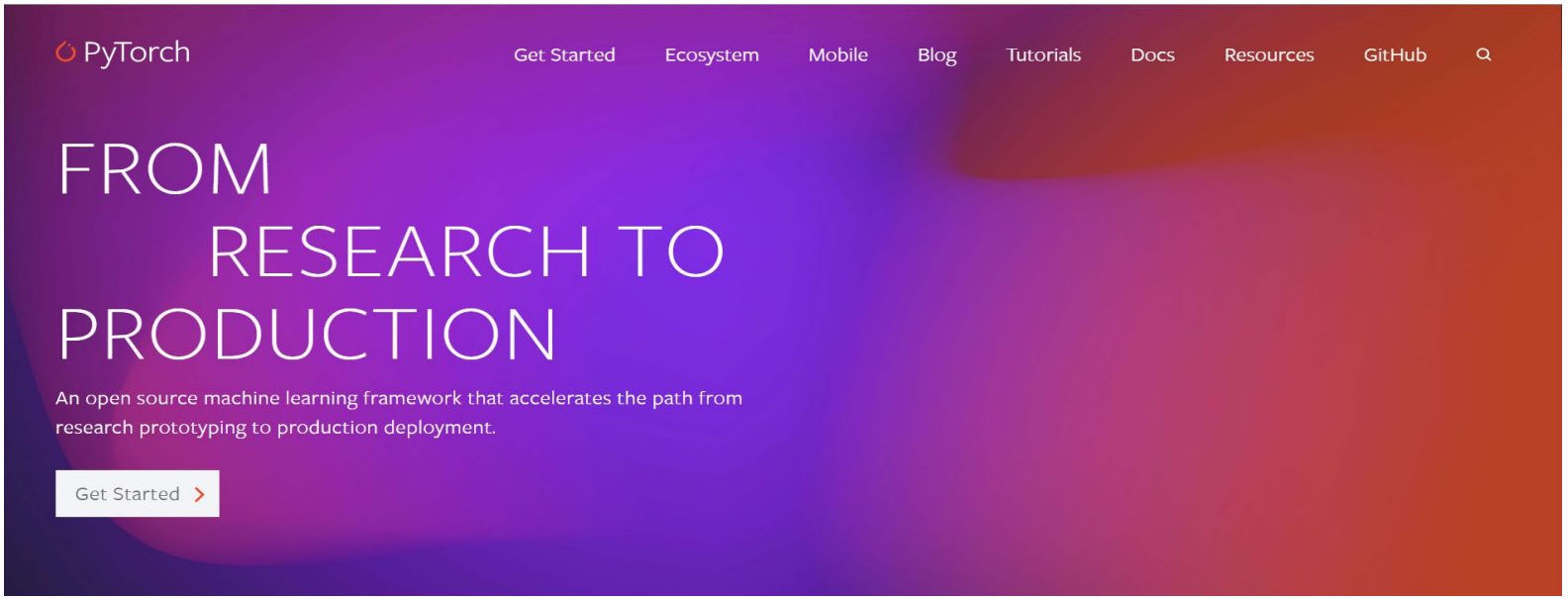
まとめ

- **NNの推論**
  - NNは入力層、中間層、出力層の多層構造
  - 信号は入力層、中間層、出力層へと階層的に伝わる
  - 中間層、出力層には活性化関数を用いる
- **NNの学習**
  - 損失関数の値が最小になる重みパラメータを探す
  - 重みパラメータを勾配に基づき更新する作業を繰り返すことで探す（勾配降下法）
  - 誤差逆伝搬により勾配を求める
- **計算グラフにより計算過程を視覚的に把握できる**
  - ノードは局所的な計算で構成
  - 順伝搬：通常の計算、逆伝搬：各ノードの微分（出力からのパスの積）を求められる

PYTORCH

PyTorch

- https://pytorch.org/



PyTorch

- **Facebookによって開発されたニューラルネットワーク学習フレームワーク**
  - NumPyと互換性の高いテンソル演算ツール+誤差逆伝播法
  - 誤差逆伝播法を用いて損失関数に対する勾配の計算を行うこともできる
  - GPUを用いた高速演算にも対応
  - Define-by-Runによるネットワーク定義
    - データごとに構成が変化するネットワークにも柔軟に対応
    - ⇔Define-and-Run
  - Pythonさえ知っていれば記述できる

torch.tensor

- **テンソル型（torch.tensor）**
  - **Pythonで高速に行列計算を行うためのクラス**
  - 行列演算は単純な足し算掛け算を繰り返すがpythonはこういった計算が遅くて苦手
  - Pythonの便利さと高速計算を同時に実現する
- **テンソル型には多次元配列を操作するための便利なメソッドが多数用意されている**
- **PyTorchをインポートして利用する**

```
>>> import torch
```

テンソルを作る

- torch.zeros: 0で埋まったテンソル(多次元行列)を作る
- torch.ones: 1で埋まったテンソルを作る
- torch.tensor: リストで表された行列からテンソルを作る
  - requires_grad=Trueをつけることで勾配を計算してくれる

```
[1]  import torch

     a = torch.zeros((10, 15), requires_grad=True)
     print(a)
     b = torch.ones((10, 15), requires_grad=True)
     print(b)
     c = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
     print(c)
```

実行結果

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],
       requires_grad=True)
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]],
       requires_grad=True)
tensor([1., 2., 3.], requires_grad=True)
```

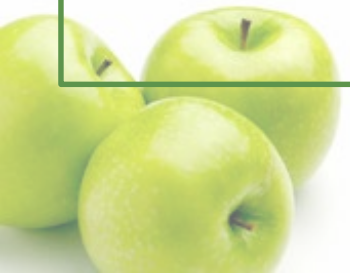
うまくいけば10x15の0の行列と10x15の1の行列と[1,2,3]の行列が表示される

テンソル型の定義

- 計算グラフのノード(変数)を表す
- Pytorchではtorch.float32(実数), torch.float64(実数)の型を用いる

```
import torch

x = torch.zeros((10, 20), dtype=torch.float32, requires_grad=True)
print(x)
x = torch.ones((10, 20), dtype=torch.float32, requires_grad=True)
print(x)
x = torch.tensor([1,2,3], dtype=torch.float32, requires_grad=True)
print(x)
```



torch.tensor：テンソル型の生成

- **テンソル型の生成**
  - メソッドtensor()を使用する。引数はPythonのリスト。
  - 多次元配列も生成可能。2次元配列は行列と捉えられる。
  - メソッドsize()で配列の形状が得られる
  - 属性dtypeに要素のデータ型が保存されている

2×3の行列生成

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
>>> import torch
>>> x = torch.tensor([1.0, 2.0, 3.0])
>>> x
tensor([ 1.,  2.,  3. ])
>> type(x)
<class 'torch.Tensor'>
>>> A = torch.tensor([[1, 2, 3], [4,5,6]])
>>> A
tensor([[1, 2, 3],
        [4, 5, 6]])
>>> A.size()
torch.Size([2, 3])
>>> A.dtype
torch.int64
```

torch.tensor：テンソル型の演算 (1)

- 四則演算子による演算は**要素毎**の演算

```
>>> x = torch.tensor([1., 2., 3.])
>>> y = torch.tensor([2., 4., 6.])
>>> x + y
tensor([3., 6., 9.])
>>> x - y
tensor([-1., -2., -3.])
>>> x * y
tensor([2., 8.,18.])
>>> x / y
tensor([0.5000,0.5000,0.5000])
```

要素毎の積

行列の積（ドット積）は matmulメソッド

$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, B = \begin{pmatrix} -1 & 1 \\ 5 & 6 \end{pmatrix}$

```
>>> A = torch.tensor([[1, 2], [3, 4]])
>>> B = torch.tensor([[-1, 1], [5, 6]])
>>> A + B
tensor([[ 0,  3],
        [ 8, 10]])
>>> A * B
tensor([[-1,  2],
        [15, 24]])
>>> torch.matmul(A,B)
tensor([[ 9, 13],
        [17, 27]])
```

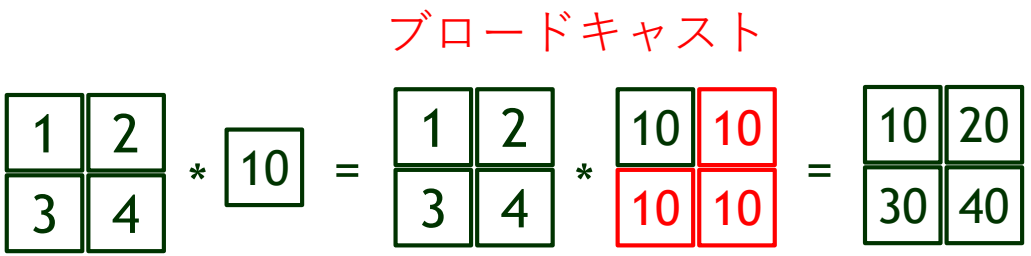
$\begin{pmatrix} 1\times -1 & 2\times 1 \\ 3\times 5 & 4\times 6 \end{pmatrix}$

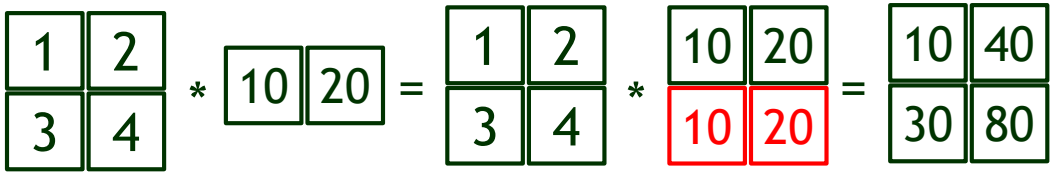
$\begin{pmatrix} 1\times -1+2\times 5 & 1\times 1+2\times 6 \\ 3\times -1+4\times 5 & 3\times 1+4\times 6 \end{pmatrix}$

torch.tensor：テンソル型の演算 (2)

- 配列の形状が異なる場合は、形状を揃えて要素毎の演算がされる(ブロードキャスト)。

```
>>> A = torch.tensor([[1, 2], [3, 4]])
>>> A*10
tensor([[10, 20],
        [30, 40]])
>>> A+10
tensor([[11, 12],
        [13, 14]])
>>> B = torch.tensor([10, 20])
>>> A*B
tensor([[10, 40],
        [30, 80]])
```





torch.tensor: 要素へのアクセス

- **Pythonリストの要素へのアクセスと同様にインデックスによりアクセス可能**
- **加えて、tensor型配列によるアクセスも可能**
  - インデックスの配列により指定番目の要素だけ取得
  - ブーリアンの配列によりTrueに対応する要素を取得

```
>>> X = torch.tensor([[10, 40], [15,
30], [1, 0]])
>>> X
tensor([[10, 40],
        [15, 30],
        [ 1 , 0]])
>>> X[0] #0行目
tensor([10, 40])
>>> X[0][1] #(0,1)の要素
tensor(40)
>>> X = X.flatten()
>>> X
tensor([10, 40, 15, 30,  1,  0])
>>> X[torch.tensor([0,2,4])]
tensor([10, 15,  1])
>>> X>20
tensor([False,  True, False,  True,
False, False])
>>> X[X>20]
tensor([40, 30])
```

Xを1次元配列に変換

tensor配列に対して不等号演算を行うと配列の各要素に対して不等号演算が行われた結果のブーリアンの配列となる

ネットワークの作成と計算

- **次の関数fの$f(1,2,3)$と$\nabla f(1,2,3)$の計算をしてみよう**

$$f(x_1, x_2, x_3) = (x_1 - 2x_2 - 1)^2 + (x_2 x_3 - 1)^2 + 1$$

- **解析的な答え**

$$f(1,2,3) = 42$$

$$\frac{\partial f}{\partial x_1} = 2(x_1 - 2x_2 - 1)$$

$$\frac{\partial f}{\partial x_2} = -4(x_1 - 2x_2 - 1) + 2(x_2 x_3 - 1)x_3$$

$$\frac{\partial f}{\partial x_3} = 2(x_2 x_3 - 1)x_2$$

$$\nabla f(1,2,3) = \left(\frac{\partial f}{\partial x_1}(1,2,3), \frac{\partial f}{\partial x_2}(1,2,3), \frac{\partial f}{\partial x_3}(1,2,3)\right) = (-8, 46, 20)$$

関数と勾配の計算

- 次の関数fの$f(1,2,3)$と$\nabla f(1,2,3)$の計算をしてみよう

$$f(x_1, x_2, x_3) = (x_1 - 2x_2 - 1)^2 + (x_2 x_3 - 1)^2 + 1$$

```
x1 = torch.tensor(1.0, requires_grad=True)
x2 = torch.tensor(2.0, requires_grad=True)
x3 = torch.tensor(3.0, requires_grad=True)

z = (x1 - 2*x2 - 1)**2 + (x2*x3 - 1)**2 + 1

print("z: ", z)
print("z: ", z.item())
z.backward()

print("x1: ", x1.grad)
print("x2: ", x2.grad)
print("x3: ", x3.grad)
```

$(x_1, x_2, x_3) = (1, 2, 3)$

$f(1,2,3)$の計算
計算結果をzに代入

zでテンソル型の結果が表示される。
z.item()とすることでzの結果が実数型で得られる

z.backward()でzに対する勾配の計算

x.gradで、zをxで偏微分した結果が得られる

関数と勾配の計算

- **計算結果**

```
z:  tensor(42., grad_fn=<AddBackward0>)
z:  42.0
x1:  tensor(-8.)
x2:  tensor(46.)
x3:  tensor(20.)
```

解析的な計算結果と一致している

- **解析的な計算結果(2ページ前のスライド)**

$$f(1,2,3) = 42$$

$$\nabla f(1,2,3) = \left(\frac{\partial f}{\partial x_1}(1,2,3), \frac{\partial f}{\partial x_2}(1,2,3), \frac{\partial f}{\partial x_3}(1,2,3)\right) = (-8, 46, 20)$$

PyTorchにはこのように関数と勾配の計算を自動的に行う機能がある

線形回帰モデルの学習

- **教師データ$D = \{(1,2),(2,3),(3,5)\}$から最小二乗法により学習される直線の式$f(x) = ax + b$を求めよ**

$$L(a,b) = \sum_{(x,y)\in D} \big(y - f(x)\big)^2$$

$$L(a,b) = (2-a-b)^2 + (3-2a-b)^2 + (5-3a-b)^2$$

$$\frac{\partial L(a,b)}{\partial a} = -2(2-a-b) - 4(3-2a-b) - 6(5-3a-b) = -46 + 28a + 12b = 0$$

$$\frac{\partial L(a,b)}{\partial b} = -2(2-a-b) - 2(3-2a-b) - 2(5-3a-b) = -20 + 12a + 6b = 0$$

解くと、$a = \frac{3}{2}, b = \frac{1}{3}$。従って、$f(x) = \frac{3}{2}x + \frac{1}{3}$。

PyTorchを使った勾配降下法による最小二乗法

```
import torch

lr = 0.01
X = [1,2,3]
Y = [2,3,5]
a = 0.0
b = 0.0

for e in range(500):
  __a = torch.tensor(a, requires_grad=True)
  __b = torch.tensor(b, requires_grad=True)
  loss = torch.tensor(0.0)
  for i in range(len(X)):
    loss = loss + (Y[i] - __a*X[i]-__b)**2
  loss.backward()
  a = a - lr * __a.grad.item()
  b = b - lr * __b.grad.item()
  print("epoch: ", e, "a: ", a, "b: ", b, "loss: ", loss.item())
```

パラメータ__a, __bの設定
損失lossの初期化

各データの損失の累積を計算

損失に対する勾配を自動計算

勾配を使ってパラメータ更新

Pythonプログラミングで求めた回帰モデルをPyTorchで計算してみよう

$D = \{(1,2),(2,3),(3,5)\}$

$f(x) = ax + b$

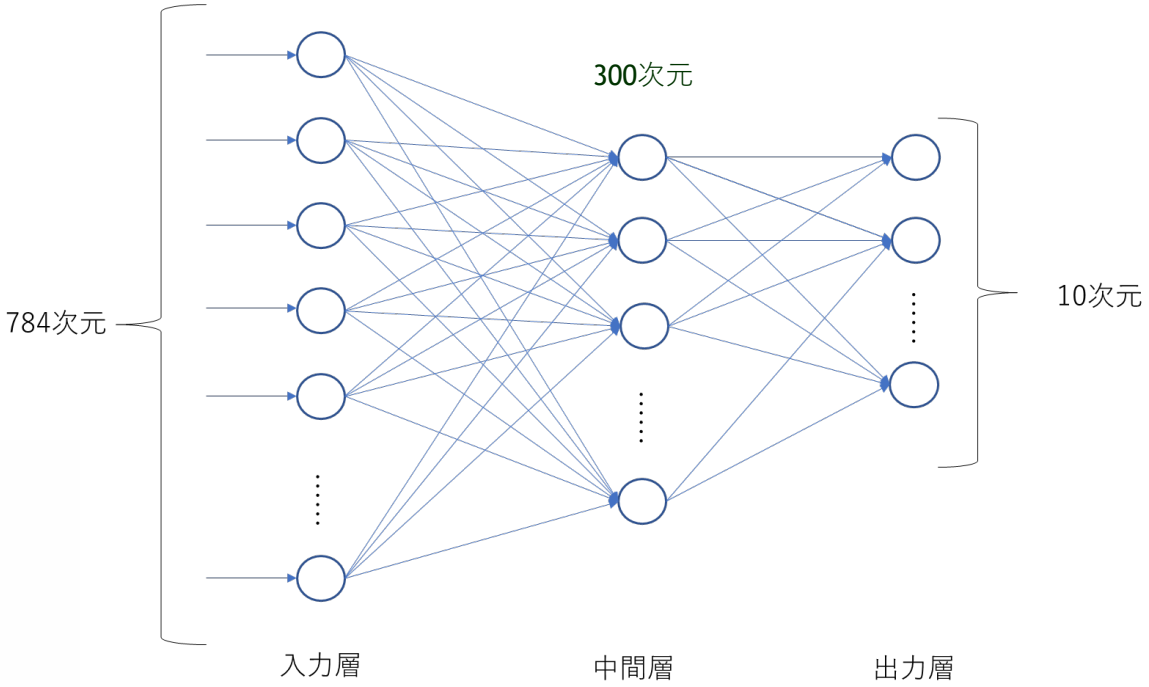
実行結果 (解析解 $a = \frac{3}{2}, b = \frac{1}{3}$)

```
epoch:  496 a:  1.4967026257514946 b:  0.34082910180091835 loss:  0.16669124364852905
epoch:  497 a:  1.4967263859510413 b:  0.34077503502368905 loss:  0.16669084131717682
epoch:  498 a:  1.4967499971389762 b:  0.34072136759757976 loss:  0.16669055819511414
epoch:  499 a:  1.4967734265327444 b:  0.34066808342933635 loss:  0.16669011116027832
```

PYTORCHによる深層学習

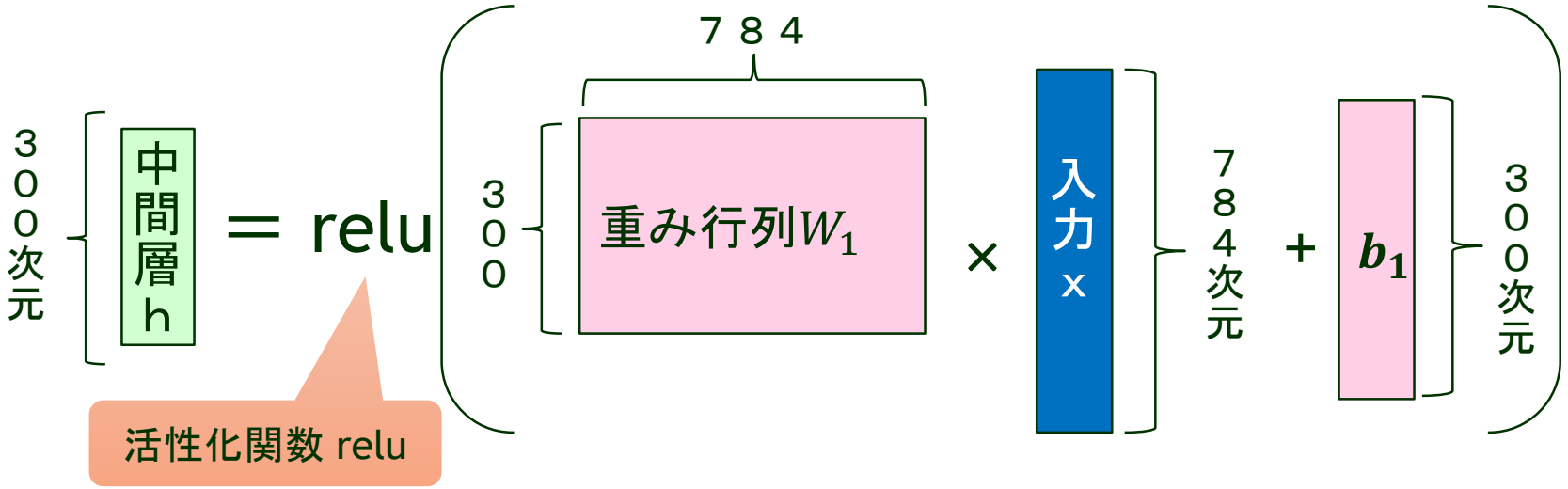
多層パーセプトロン

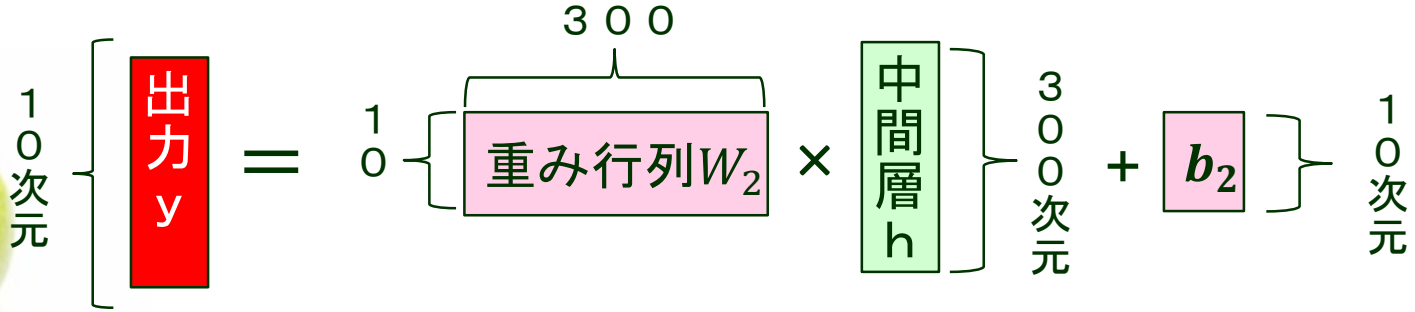
- 入力層(784次元)、中間層(300次元)、出力層(10次元)の3層のニューラルネットワークを作って白黒画像(28x28)の画像認識を行います



多層パーセプトロン

- 入力層(784次元)、中間層(300次元)、出力層(10次元)の3層のニューラルネットワークの計算





線形変換(全結合層) torch.nn.Linear

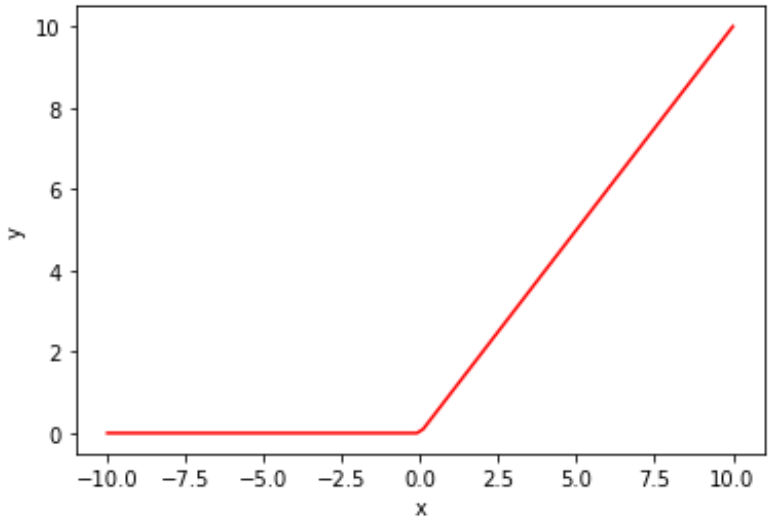
- torch.nn.Linear
  - 線形変換(全結合層)のモジュール
    - $\boldsymbol{y} = \mathrm{W}\boldsymbol{x} + \boldsymbol{b}$を計算する
    - W(行列)と$\boldsymbol{b}$(ベクトル)がパラメータ
  - `torch.nn.Linear(in_dim, out_dim)`
    - `in_dim`次元のベクトルから`out_dim`次元のベクトルに射影する線形変換を計算する層を作る
  - 例: `torch.nn.Linear(3, 4)`
    - 3次元ベクトルから4次元ベクトルに射影する線形変換

活性化関数 ReLU (Rectified Linear Unit) 関数

- **ReLU関数**
  - 出力の活性化(オン･オフ)を表す非線形関数

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

  - `torch.nn.functional.relu(x)`
    - xに対して、 $h(x)$の計算をする



データ MNIST

- MNIST
  - 手書き文字の認識を行うタスク
  - 画像に対して、どの数字が書かれているかを予測する。

  - 訓練データは60,000データ
  - テストデータは10,000データ
  - 各データは画像とラベルのペア
  - 各画像は28×28ピクセル
  - ラベルは0～9までのいずれかの数字。画像に対する正解の数字。

データの読み込み

- データ読み込みのためのプログラム

```
import numpy as np
import torch
import torch.nn.functional as F
import torchvision as tv

train_dataset = tv.datasets.MNIST(root="./", train=True,
                                  transform=tv.transforms.ToTensor(),
                                  download=True)
test_dataset = tv.datasets.MNIST(root="./", train=False,
                                 transform=tv.transforms.ToTensor(),
                                 download=True)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=100,
                                          shuffle=False)
```

ライブラリの読み込み

訓練データとテストデータの読み込み(初めて実行するときはデータをネットからダウンロードする)

訓練データとテストデータのミニバッチ処理
・ミニバッチサイズ=100
・データの順番をシャッフル

DataLoaderがしていること

- **ミニバッチ形式への変換**
  - もともとのデータ（1×28×28のリストと数値のペアのリスト）

```
[([[[0,0,…画像データ…,0,0]]], 5),
 ([[[0,0,…画像データ…,0,0]]], 2),
 …
 ([[[0,0,…画像データ…,0,0]]], 8)]
```

60000個(=データサイズ)

  - 変換後のデータ(1×28×28のリストと数値のリストのペア

```
([[[[0,0,…画像データ…,0,0]]],
  [[[0,0,…画像データ…,0,0]]],
  …
  [[[0,0,…画像データ…,0,0]]]],
 [5, 2, …, 8])
```

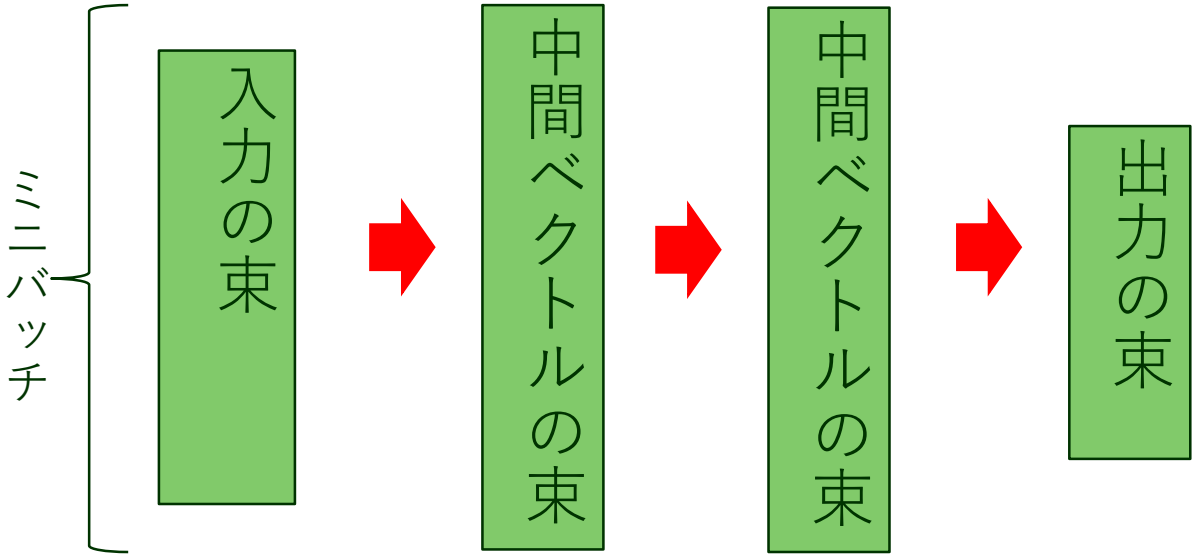
入力となる画像だけを集めたもの
(100個=ミニバッチサイズ)

出力となる数値だけを集めたもの
(100個=ミニバッチサイズ)

100個のミニバッチデータが600個続く

ミニバッチ

- ミニバッチ内のデータをまとめて左から右に流す(計算する)



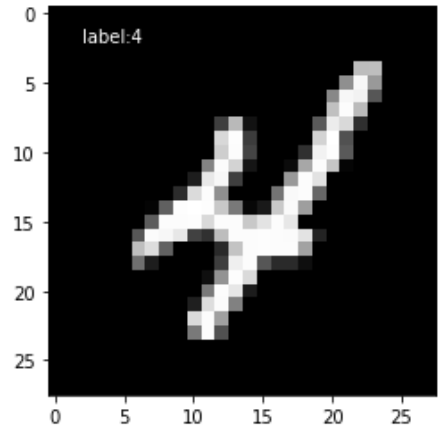
- まとめて計算できる行列演算が増える
  - 行列演算が得意なGPUにとって非常に効率の良い計算方法

(参考)どんな画像が入っているのか みてみよう

- 先程のプログラムの下に次のプログラムを追加して実行してみよう(ただし、コマンドプロンプトでpip3 install matplotlibの実行が必要)

```
import matplotlib.pyplot as plt
for i in range(10):
  print(train_dataset[i])
  plt.imshow(train_dataset[i][0][0], cmap="gray")
  txt = "label:"+str(train_dataset[i][1])
  plt.text(2, 2, txt, color="white")
  plt.show()
```

train_dataset[i][0][0]でi番目の画像(28x28の白黒)
train_dataset[i][1]でi番目の正解ラベル



・具体的なデータの中身が表示される
・このような画像が10枚表示される(左上の「label:4」はこの画像の正解ラベル)

(参考) カラー画像だったら、
train_dataset[i][0][0] 赤(R)の画像
train_dataset[i][0][1] 緑(G)の画像
train_dataset[i][0][2] 青(B)の画像

今回は白黒なので、train_dataset[i][0][0]だけ使う

コメント化

- **せっかく書いたプログラムを消すのはもったいないので、コメント化しておこう**
  - 「#」をいれるとそこから行末までは無視される(コメント化)

```
for i in range(10):
  print(train_dataset[i])
  plt.imshow(train_dataset[i][0][0], cmap="gray")
  txt = "label:"+str(train_dataset[i][1])
  plt.text(2, 2, txt, color="white")
  plt.show()
```

```
# for i in range(10):
#   print(train_dataset[i])
#   plt.imshow(train_dataset[i][0][0], cmap="gray")
#   txt = "label:"+str(train_dataset[i][1])
#   plt.text(2, 2, txt, color="white")
#   plt.show()
```

コメント化された部分は実行されない

プログラムをみやすくするために積極的にコメントをいれよう
(例) #初期設定
　　#モデルの定義
　　#ここから訓練

ネットワークの作成

- ネットワークの設定とパラメータ最適化の設定

```
l1 = torch.nn.Linear(784, 300)
l2 = torch.nn.Linear(300, 10)
params = list(l1.parameters())+ list(l2.parameters())
optimizer = torch.optim.Adam(params)

def mynet(x):
  h = F.relu(l1(x))
  y = l2(h)
  return y
```

l1とl2は線形変換の関数

l1/l2.parameters()でl1とl2に含まれるパラメータを取り出す

Adamというパラメータ最適化手法を使う。パラメータ更新に関わるパラメータ群paramsを渡す

ネットワーク全体の計算を行う関数 mynet
・xが入力、yが出力
・hはニューラルネットワーク前半(784次元→300次元への線形変換l1と活性化関数ReLUの適用)
・yはニューラルネットワーク後半(hに対して300次元→10次元への線形変換l2を適用)

ネットワークの学習

- **訓練データからの学習**

データ全体を10回学習する

train_loaderから100個ずつデータを受け取る
Imagesは画像データ100枚
labelsは正解ラベル100個

```
#学習
for e in range(10):
  loss = 0
  for images, labels in train_loader:
    images = images.view(-1, 28*28)
    optimizer.zero_grad()
    y = mynet(images)
    batchloss = F.cross_entropy(y, labels)
    batchloss.backward()
    optimizer.step()
    loss = loss + batchloss.item()
  print("epoch:", e, "loss:", loss)
```

(100×1x28×28)から(100×784)に変形

勾配を初期化

ネットワークの計算(ラベルの予測)

正解ラベルに対する出力の損失

誤差に対する勾配を計算

パラメータ更新

ミニバッチでの損失をlossに足す

ネットワークによる推論

- テストデータに対して学習したネットワークを評価する

```
#テスト
correct = 0
total = len(test_loader.dataset)
for images, labels in test_loader:
  images = images.view(-1, 28*28)
  y = mynet(images)
  pred_labels = y.max(dim=1)[1]
  correct = correct + (pred_labels == labels).sum()

print("correct:", correct.item())
print("total:", total)
print("accuracy:", correct.item()/total)
```

test_loaderから100個ずつデータを受け取る
Imagesは画像データ100枚
labelsは正解ラベル100個

(100×1x28×28)から(100×784)に変形

ネットワークの計算

ラベルの予測(最大値となるラベル)

100個のうち正解数がいくつか数える

correctは正解数
totalはテストデータのデータ数
accuracyは精度

実行結果

```
epoch: 0 loss: 195.63006672263145
epoch: 1 loss: 82.39563230797648
epoch: 2 loss: 54.87096862308681
epoch: 3 loss: 40.70517487358302
epoch: 4 loss: 31.048207819461823
epoch: 5 loss: 23.900350784882903
epoch: 6 loss: 19.02593113365583
epoch: 7 loss: 14.900030710035935
epoch: 8 loss: 12.13549662521109
epoch: 9 loss: 9.818495093728416
```

損失がだんだん減っていることがわかる

```
correct: 9798
total: 10000
accuracy: 0.9798
```

10000データ中、9798データの正解
(=97.98%の正解率)

- http://yann.lecun.com/exdb/mnist/ の結果と比較してみよう

PYTORCHのクラスライブラリ

torch.nn.functional

- **パラメータ(求めたい重み変数)を含まない関数はtorch.nn.functionalパッケージに用意されています**
  - 活性化関数
  - 損失関数など

torch.nn.functional

- 簡単な例

```
x1 = torch.tensor(1.5)
x2 = torch.tensor(0.75)
y = torch.nn.functional.relu(x1)
z = torch.nn.functional.mse_loss(x1, x2)
print("y: "+str(y))
print("z: "+str(z))
```

y: 1.5000
z: 0.5625

torch.nn

- パラメータ(求めたい重み変数)を含む関数はtorch.nnパッケージに用意されています。
- functionalとnnは機能的に似ていますが、違いはパラメータを含むか含まないかということになります。
- 従って、多くのモデルはnnを使って構成し、活性化関数などの簡単な関数はfunctionalで表現します。(使用例をみていればどちらを使えばよいかだいたいわかります)

torch.nn.Paremeter

- **PyTorchでの重み変数(パラメータ)を生成するクラス**
  - テンソル型を生成する際にRequires_grad=Trueとするだけでは、pytorchの中ではパラメータ(保存と更新の対象となる変数)として認識されない
  - nn.Parameter(x)とすることでxを中身とするパラメータを生成する

(ただし、通常使うことはあまり無い。カスタムレイヤーを作るときに必要)

  - 例: 2×3の重み行列

```
W = torch.nn.Parameter(torch.tensor([[1.,2.,3.],
                                     [4.,5.,6.]]))
```

↓

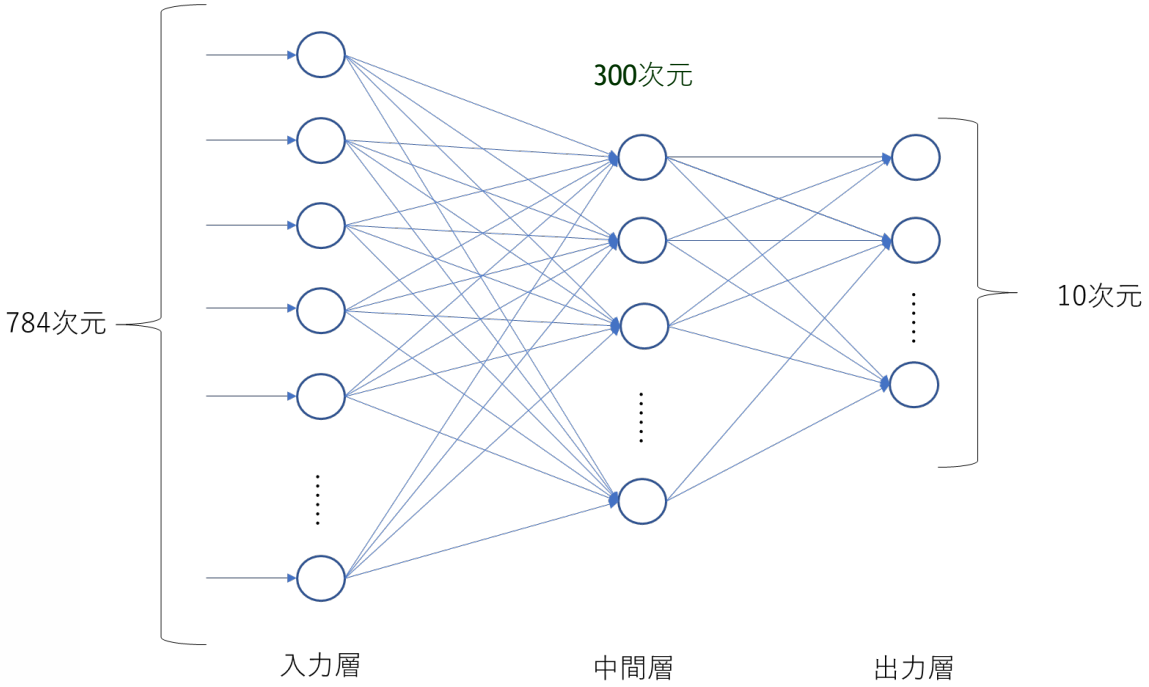
```
Parameter containing:
tensor([[1., 2., 3.],
        [4., 5., 6.]], requires_grad=True)
```

torch.nn.Module

- **Moduleクラス**
  - ネットワーク全体の構成(モデル)を与えるクラス
    - torch.nnやtorch.nn.functionalをパーツとして全体を構成し、そのネットワークが表現する合成関数を計算する
    - 合成関数に含まれるパラメータを学習する

Module

- 入力層(784次元)、中間層(300次元)、出力層(10次元)の3層のニューラルネットワークを作って白黒画像(28x28)の画像認識を行います



Module

- **例: 入力層(784次元)、中間層(300次元)、出力層(10次元)の3層のニューラルネットワーク (活性化関数はReLU)**

```
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet,self).__init__()
        self.l1 = torch.nn.Linear(784,300)
        self.l2 = torch.nn.Linear(300,10)
    def forward(self,x):
        h = torch.nn.functional.relu(self.l1(x))
        y = torch.nn.functional.relu(self.l2(h))
        return y
```

__init__()関数でネットワークの要素を定義

forward計算のときにforward()が呼ばれる。同時にネットワークを構築

optim

- **optimクラス**
  - 最適化ツール
  - 様々な最適化アルゴリズムを選択して利用できる
  - 最初に次の定義をして最適化クラスを選択、初期化

```
model = MyNet()
optimizer = torch.optim.SGD(model.parameters())
```

optimizerを選択し
パラメータをセット

  - パラメータ更新時

```
optimizer.zero_grad()
loss = model(x)
loss.backward()
optimizer.step()
```

勾配の初期化

パラメータを更新

PYTORCHクラスを用いたMNIST

データの読み込み

- データ読み込みのためのプログラム(以前のMNIST用プログラムと同じ)

```
import numpy as np
import torch
import torch.nn.functional as F
import torchvision as tv

train_dataset = tv.datasets.MNIST(root="./", train=True,
                                  transform=tv.transforms.ToTensor(),
                                  download=True)
test_dataset = tv.datasets.MNIST(root="./", train=False,
                                 transform=tv.transforms.ToTensor(),
                                 download=True)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=100,
                                          shuffle=False)
```

ライブラリの読み込み

訓練データとテストデータの読み込み(初めて実行するときはデータをネットからダウンロードする)

訓練データとテストデータのミニバッチ処理
・ミニバッチサイズ=100
・データの順番をシャッフル

グローバル変数の定義

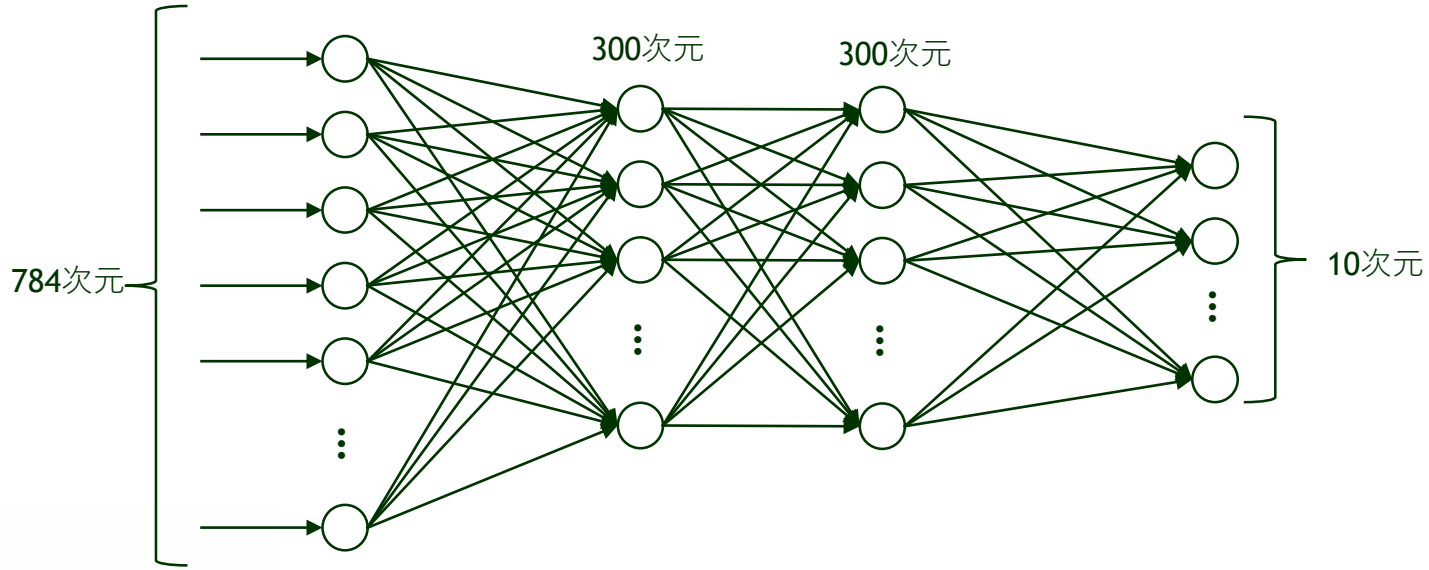
- グローバル変数を定義しておきます

```
MODELNAME = "mnist.model"
EPOCH = 10
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```

GPUが利用可能ならDEVICE=”cuda”
CPUを利用するのならDEVICE=”cpu”

モデルの定義

- **手書き文字の認識を行うモデル**
  - 入力層(784次元)、中間層1(300次元)、中間層2(300次元)、出力層(10次元)の中間層2層のネットワーク



  - 全結合層のみによるモデル
  - torch.nn.Conv2dを使用し、入力データの形式を少し変更すれば、畳込みニューラルネットワーク(CNN)も可能

モデルの定義

- **手書き文字の認識を行うモデルを構築**
  - 入力層(784次元)、中間層1(300次元)、中間層2(300次元)、出力層(10次元)の中間層2層のネットワーク

```
class MNIST(torch.nn.Module):
  def __init__(self):
    super(MNIST,self).__init__()
    self.l1 = torch.nn.Linear(784, 300)
    self.l2 = torch.nn.Linear(300, 300)
    self.l3 = torch.nn.Linear(300, 10)
  def forward(self,x):
    h = F.relu(self.l1(x))
    h = F.relu(self.l2(h))
    y = self.l3(h)
    return y
```

__init__()関数でネットワークの要素を定義

forward計算のときにforward()が呼ばれる。同時にネットワークを構築

モデルの定義　__init__メソッド

- **__init__メソッド**
  - 入力層の次元数: 768 (= ピクセル数)
  - 第1中間層の次元数: 300
  - 第2中間層の次元数: 300
  - 出力次元数: 10 (= 多クラス分類のクラス数)
  - ニューラルネットワークを定義する手順
    - 親クラスのコンストラクタを呼ぶ
    - __init__(): 以下に使用するネットワークを定義
  - torch.nn.Linear: 全結合層
    - torch.nn.Linearの第一引数: 全結合層の入力次元数
    - torch.nn.Linearの第二引数: 全結合層の出力次元数
    - 全結合層以外にも多くのネットワークが使用できます (https://pytorch.org/docs/stable/nn.html 参照)。

モデルの定義 forwardメソッド

- forwardメソッド
  - ネットワークの結合を定義
  - 784次元のベクトル(28×28ピクセルの画像)を受け取る
  - 10次元のベクトルを返す
  - l1が784次元(入力層)を中間次元(300次元)に圧縮
  - l2はl1の出力を受け取り、中間ベクトル(300次元)を生成
  - l3は中間ベクトルを受け取り、10次元のベクトル(出力層)に変換
    - MNISTの分類が10クラス(0～9の文字)であるため、10次元のベクトルを出力
  - L1, l2層に対しては、活性化関数のrelu関数

モデルの訓練

```
def train():
  model = MNIST().to(DEVICE) #モデルを定義してDEVICEにのせる
  optimizer = torch.optim.Adam(model.parameters()) #最適化にAdamを使う
  for epoch in range(EPOCH): #EPOCH回最適化を行う
    loss = 0
    for images, labels in train_loader:
      images = images.view(-1, 28*28).to(DEVICE)
      labels = labels.to(DEVICE)
      optimizer.zero_grad() #勾配の初期化
      y = model(images) #forward計算
      batchloss = F.cross_entropy(y, labels) #損失の計算
      batchloss.backward() #逆伝搬の計算
      optimizer.step() #重み変数の更新
      loss = loss + batchloss.item()

    print("epoch", epoch, ": loss", loss)
  torch.save(model.state_dict(), MODELNAME)
```

データをミニバッチサイズに切り出す

(100×1×28×28)から(100×784)に変形

ミニバッチ内の処理

モデルをファイルに保存

テスト

```
def test():
  total = len(test_loader.dataset)
  correct = 0
  model = MNIST().to(DEVICE)
  model.load_state_dict(torch.load(MODELNAME))
  model.eval()
  for images, labels in test_loader:
    images = images.view(-1, 28*28).to(DEVICE)
    labels = labels.to(DEVICE)
    y = model(images) #forward計算
    pred_labels = y.max(dim=1)[1]
    correct = correct + (pred_labels == labels).sum()
  print("correct:", correct.item())
  print("total:", total)
  print("accuracy:", (correct.item() / float(total)))
```

ファイルに保存したモデルをロード

テストデータに対してループ

最大値の次元IDを得る

正解率を計算

訓練・テスト

```
train()
#test()
```

訓練・テストのうちやりたくないこと
に対してコメントアウトする

Pythonでは#がコメントを表す記号

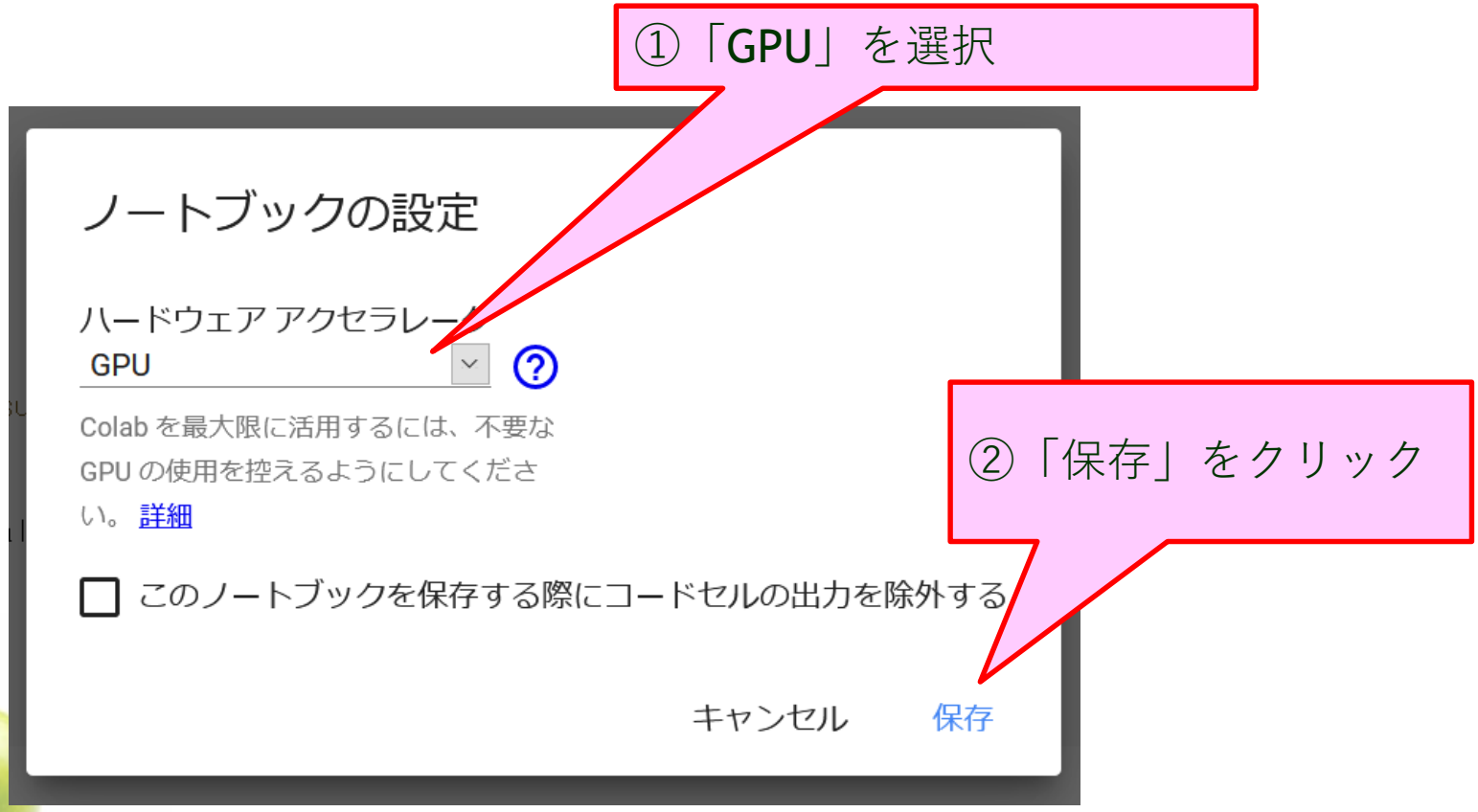
- train()を実行して訓練してみよう
- test()を実行して精度を測ってみよう
- 各データに対し、正解ラベルと予測ラベルをそれぞれ表示させてみよう
- http://yann.lecun.com/exdb/mnist/ の結果と比較してみよう

GPUを使ってみよう

- Google CoraboratoryではGPUを利用することができます

GPUを使ってみよう

- GPUの設定ができたらMNISTの学習を行ってみよう



まとめ

- Pythonのクラスについて学んだ
- テンソル型について勉強した
- クラスを用いたPyTorchの使い方を学んだ
- GPUを用いた深層学習を行った

オプション課題

CIFAR-10の画像分類

- CIFAR-10というデータセットに対して、学習と推論を行い、その精度評価の実験を行いましょう。
- CIFAR-10は10クラスの画像分類データであり、

```
train_dataset = tv.datasets.CIFAR10(root="./", train=True,
                                    transform=tv.transforms.ToTensor(),
                                    download=True)
test_dataset = tv.datasets.CIFAR10(root="./", train=False,
                                   transform=tv.transforms.ToTensor(),
                                   download=True)
```

を実行することによりMNIST同様、データが得られます。各画像は3色×32pixel×32pixelとなっています。

より高い精度の実現を目指して、モデル設計を行いましょう。