

ロボットカード学ぶ 深層学習の基礎

© 2019 Preferred Networks, Inc. and UNIVERSITY of YAMANASHI, All Rights Reserved.

本テキスト・画像の無断転載・複製を固く禁じます。二次利用不可、商用利用不可、但し教育機関の利用は可能。

Chainer™は、株式会社Preferred Networksの日本国およびその他の国における商標または登録商標です。

本テキストについて

- 本テキストは、株式会社Preferred Networks（以下、PFN）と
国立大学法人山梨大学の共同研究として開発された教材をベー
スに、より間口を広くして、高等専門学校から大学学部レベル
の学生の方々に、深層学習に興味を持っていただく内容になっ
ています。
- 教育機関での授業や、個人の自主学習用など、深層学習を学び
たい方に学習機会を広く提供するため無料でご利用いただけま
す。ただし、商用利用はできませんのでご注意ください。
- 詳しくは利用規約（TermsOfService.md）をご参照ください。

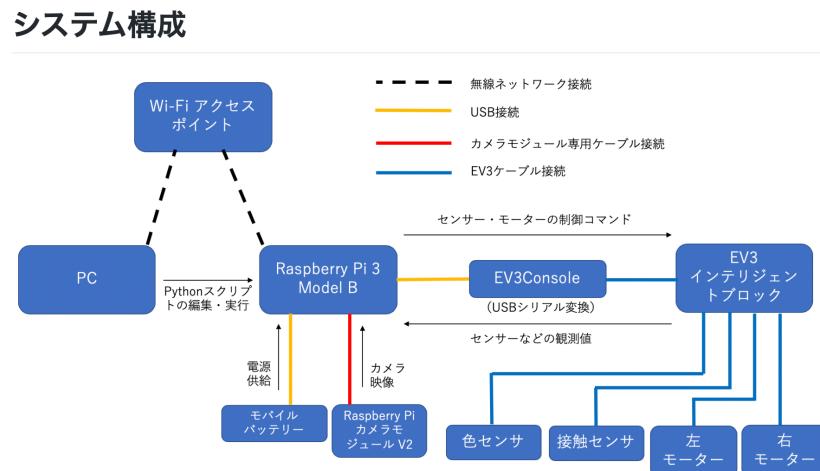
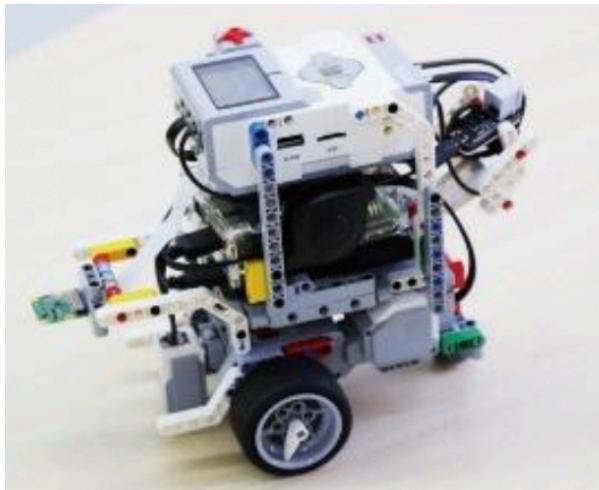
目次

- 環境の準備
- 教師あり学習
- ニューラルネットワークと深層学習
- Chainerの基礎
- Chainerによるロボット制御

環境の準備

環境構築方法について

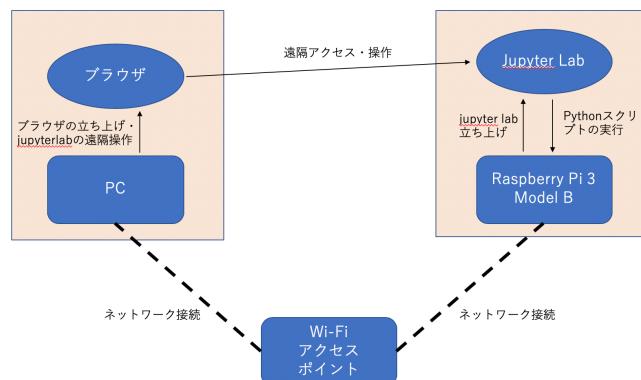
- 「Chainerによるロボット制御」の章ではRaspberry Pi 3とロボットカー（レゴ®マインドストーム® EV3）を連携させたシステムを利用します。



- 必要機材、環境構築などについてはこちらをご参照ください。
 - <https://github.com/pfnet-research/chainer-ev3>

プログラミングについて

- 「Chainerの基礎」と「Chainerによるロボット制御」ではPythonとNumPyの基礎を習得しているユーザーを前提で書かれています。未習得のユーザーは先にChainerチュートリアルなどによる学習を推奨します。
- プログラミングはRaspberry Pi上のJupyter LabにPCから遠隔でアクセスして行います。Jupyter Labの環境構築方法はGithubをご参照ください。



```
import os
import numpy as np
import chainer
import chainer.links as L
from chainer import optimizers
from chainer import serializers
from chainer import training
from chainer.training import extensions

class MLInference(object):
    def __init__(self):
        self.x = np.array([[-1.0, -1.0], [1.0, 1.0], [-1.0, 1.0], [1.0, -1.0], [0.0, 0.0]], dtype=np.float32)
        self.y = np.array([1, 1, -1, -1, 0], dtype=np.int32)
        with self.x.astype(np.float32):
            self.z = L.Linear(2, 200)
            self.a = F.relu(self.z)
            self.b = F.linear(self.a, 200, 1)
            self.c = F.linear(self.b, 1, 1)
        self.w = self.c[0][0]
        self.b = self.c[1][0]

    def generate_data(self, N=10000):
        self.x = np.random.uniform(-1.0, 1.0, size=(N, 2))
        self.y = np.random.randint(0, 2, size=N)

    def set_up_a_neural_network_to_train(self):
        self.model = L.Classifier(MLInference)
        self.model.use_cleargrads()
        self.model.compute_accuracy = True
        self.optimizer = optimizers.Adam()
        self.optimizer.setup(self.model)

    def train(self, n_epochs=1000, batch_size=100, learning_rate=0.001):
        self.set_up_a_neural_network_to_train()
        self.optimizer = optimizers.Adam(alpha=learning_rate)
        self.optimizer.setup(self.model)
        self.model.compute_accuracy = True
        self.optimizer.update(self.model)
```

EV3を制御するAPIについて

- ・「Chainerによるロボット制御」の章ではRaspberry Pi 3からレゴマインドストームEV3を制御するためのAPIを利用します。
- ・**chainer-ev3/workspace**以下に各APIの使い方を説明するサンプルコードを用意しています。実行方法については**chainer-ev3/README.md**をご参照ください。
 - ・タッチセンサーの値を取得する。
 - basic_get_touch_sensor_state.ipynb
 - ・カラーセンサーの値を取得する。
 - basic_get_color_sensor_intensity.ipynb
 - ・モーターを動かす。
 - basic_go_straight.ipynb, basic_go_around.ipynb
 - ・EV3のLCDに文字列を表示する。
 - basic_display_strings_on_lcd.ipynb
 - ・EV3の5つのボタン（ENTER, UP, DOWN, LEFT, RIGHT）の状態を検知する。
 - basic_button_click.ipynb

カメラモジュールを制御するAPIについて

- ・「Chainerによるロボット制御」の章ではRaspberry Piのカメラモジュールを制御するためのAPIを用意しています。
- ・**chainer-ev3/workspace**以下に各APIの使い方を説明するサンプルコードを用意しています。実行方法については**chainer-ev3/README.md**をご参照ください。
 - ・ カメラの画像を 1 秒毎に画面に表示する。
 - camera_show_image.ipynb
 - ・ カメラ画像を手動で取得し、ラベルを付けて保存する。
 - camera_labeled_image_logger.ipynb

コースデータについて

- ・「Chainerによるロボット制御」の課題2では紙に印刷した白黒のサーキットコースを利用します。
- ・**chainer-ev3/course/circuit.pdf**を印刷してください。
コピー用の上質紙、A0白黒印刷で動作確認を行っています。

照明条件について

- ・「Chainerによるロボット制御」では二値化したカメラ画像を利用するため、照明条件によってうまく動作しない場合があります。保存された画像を確認して影響をうけている場合は以下を試してください。
 - ・LED照明の場合は蛍光灯照明の部屋に変更して試す。
 - ・カメラAPI(VideoStream)のパラメータ**bin_threshold**を調整する。

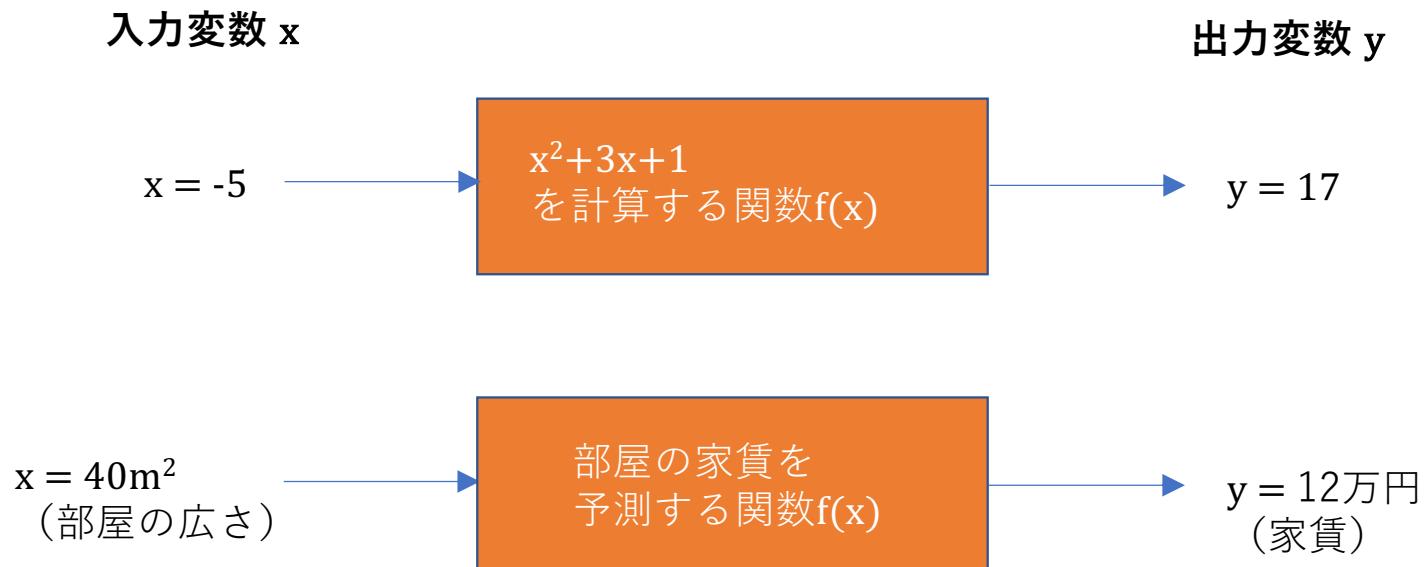
教師あり学習

本章でニューラルネットワークや深層学習に必要となる
教師あり学習の概念を中心に学ぼう。

プログラムを作る = 関数を作る

関数についてのイメージをつかもう。

関数のイメージ 「**入力変数**を与えると**出力変数**が出てくる箱」

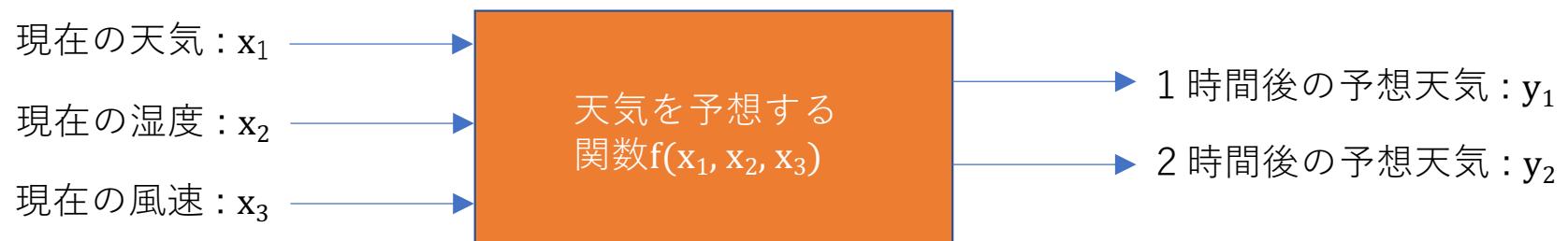


多次元の入力と出力を持つ関数

多次元の入出力変数を持つ関数のイメージをつかもう。

関数は**多次元（多変数）**の入力変数と出力変数を扱うこともできる。

例) 天気予報

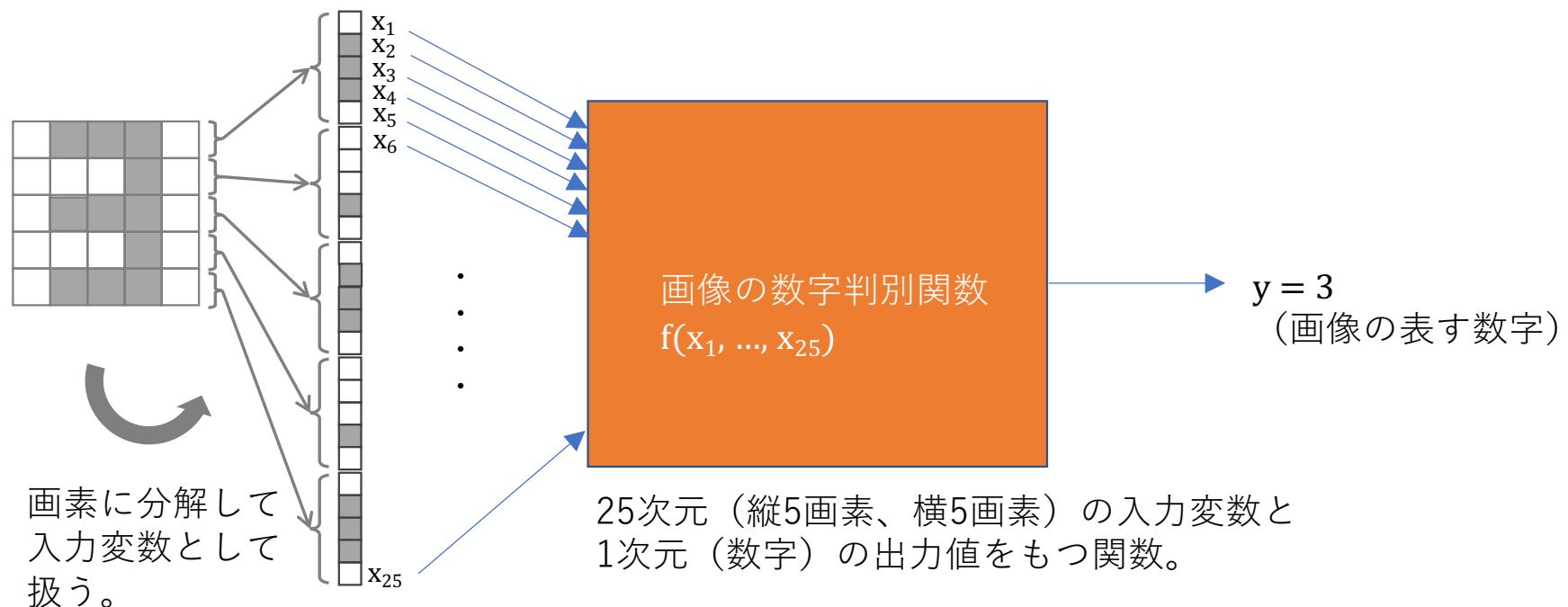


この関数は 3 次元の入力変数(x_1, x_2, x_3)、
2 次元の出力変数(y_1, y_2)を持つ。

高次元のデータを扱う関数

画像などの高次元データを関数で扱うイメージをつかもう。

画像などのデータ処理も高次元の入力変数を持つ関数として扱う。



関数の中身を作るためのアプローチ

関数の中身を作るためのルールベースと教師あり学習のプログラミングの違いを学ぼう。

- ルールベース
 - 定義された入出力に対して、関数の中身を人が考えて設計する。
(例：数式を書く、プログラムのロジックを書く、など)
 - 人が入力値に対応する出力値の法則性を記述できる場合に有効。
- 教師あり学習
 - 入出力の法則性を記述することが困難な場合に検討する方法の一つ。
 - データセット（作りたい関数の入出力値の例示の集合）から学習によって関数の近似表現を自動的に獲得する。
 - 獲得した関数によって未知の入力値に対する出力値を予測する。

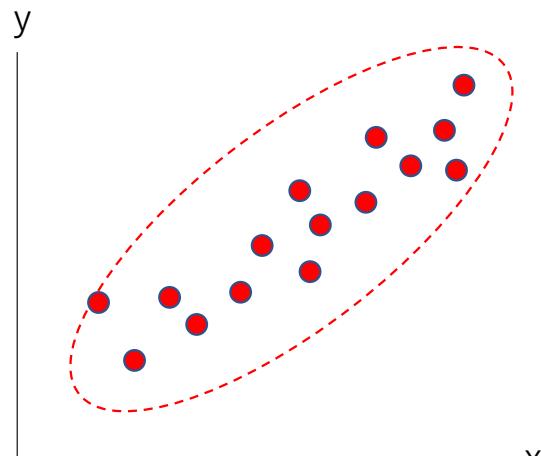
簡単な教師あり学習の例

1次関数を例に簡単な教師あり学習のイメージを覚えよう。

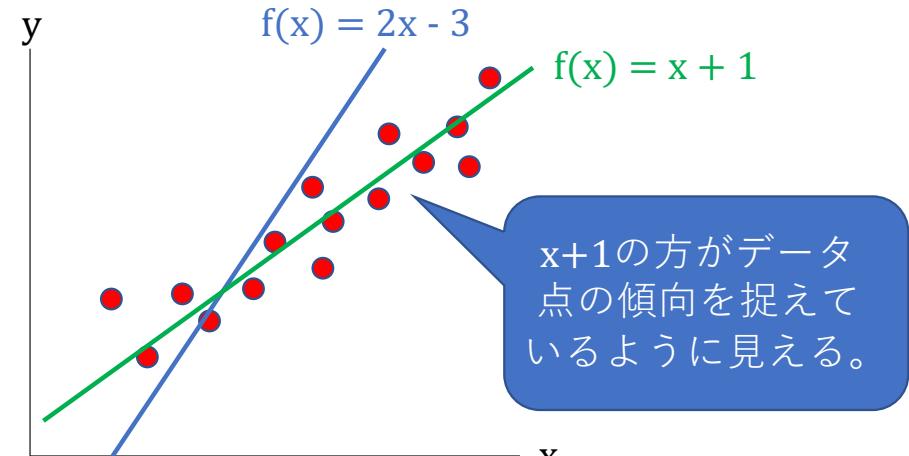
データセット D (グラフ中の赤い点) が与えられるとする。

その時に D を特徴づける関数 $y = f(x)$ を獲得したい。

$f(x) = wx + b$ と仮定したときに **パラメータ** w, b を求める。



データ点のプロット図

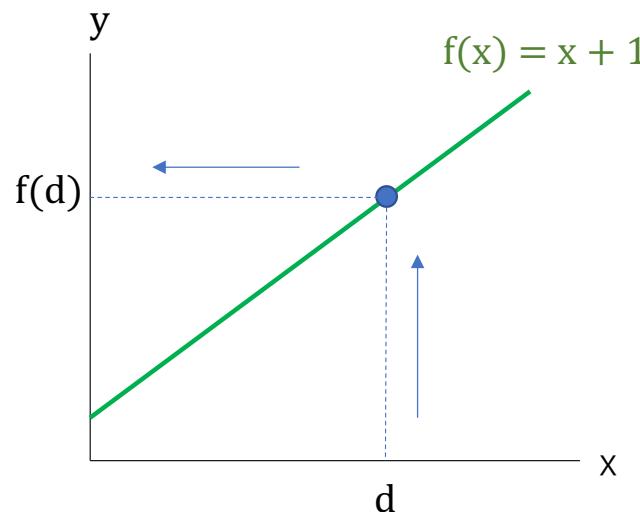


$w=2, b=-3$ と $w=1, b=1$ の場合の $f(x)$

簡単な教師あり学習の例

データを特徴づける関数を獲得することで、未知の入力値に対する出力値を予測できることを学ぼう。

D を特徴づける関数 $f(x) = x + 1$ を獲得したことで、未知の入力 d が与えられたときに、適切な出力値を $f(d)$ として予測できるようになる。



教師あり学習の目的

教師あり学習がどのような問題を解くもののか学ぼう。

教師あり学習ではパラメータを持つ関数を仮定し、与えられたデータセットを特徴づけるパラメータの値を探す。

例) 入力変数の次元数が大きい場合
(n 変数の1次関数の例)

$$f(x_{1..n}) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

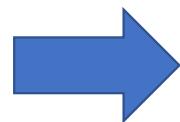
→ 調整すべきパラメータは $n+1$ 個

例) 複雑な関数を仮定する場合
(1変数の m 次式の例)

$$f(x) = w_1x + w_2x^2 + \dots + w_mx^m + b$$

→ 調整すべきパラメータは $m+1$ 個

高次元の入出力変数、または、複雑な関数を利用する場合に、調整すべきパラメータ数が増加する。



自動的に良いパラメータの組み合わせを効率よく探索する方法が必要となる。

教師あり学習：回帰問題と分類問題

教師あり学習でよく扱う問題について学ぼう。

・回帰問題

- ・出力変数が連続値である場合の問題。
- ・例
 - 電力需要の予測
 - 河川の水位予想



本テキストでは
回帰問題を中心
に説明する。

・分類問題

- ・出力変数がカテゴリ値である場合の問題。
- ・例
 - 画像分類
 - スパムメール分類

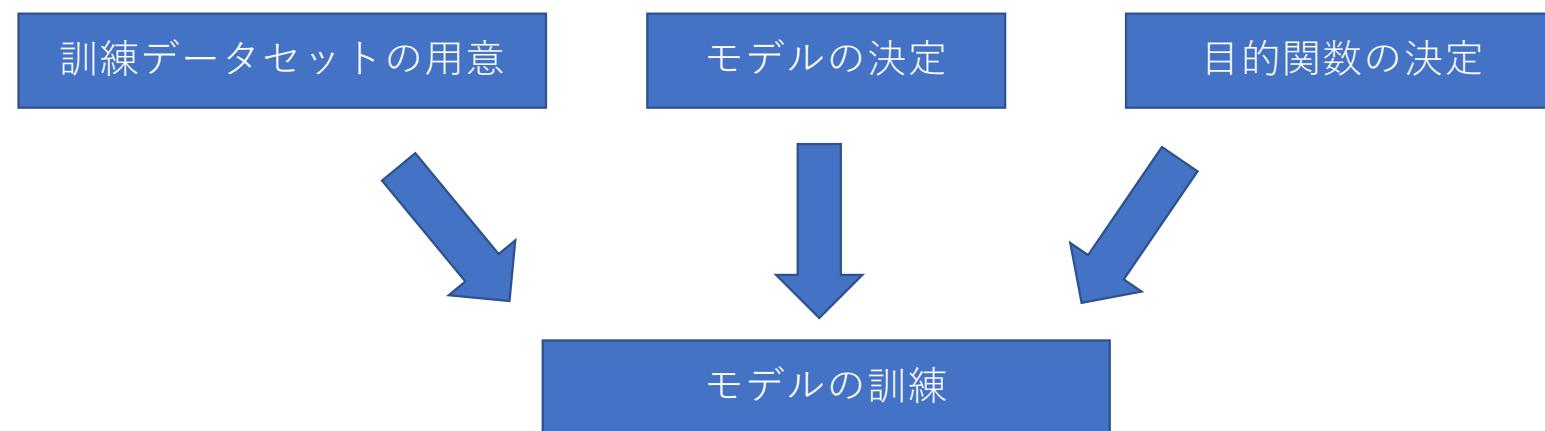
教師あり学習：モデル、訓練、推論

教師あり学習で頻出する基本的な用語を覚えよう。

- 教師あり学習（機械学習）では、データの背後にある法則を捉えて近似する関数 $f(x)$ のことを**モデル**と呼ぶ。
注釈：「モデル」という言葉が何を表すべきかについては、完全に統一的な意味で用いられている訳ではない。
- データの法則を説明できるパラメータを獲得させることを**訓練**と呼ぶ。訓練によって獲得されたパラメータを持つモデルを**訓練済みモデル**と呼ぶ。
- 訓練済みモデルを使って、未知の入力値が与えられた時の出力値を予測することを**推論**と呼ぶ。また、推論によって得られた出力値のことを**予測値**と呼ぶ。

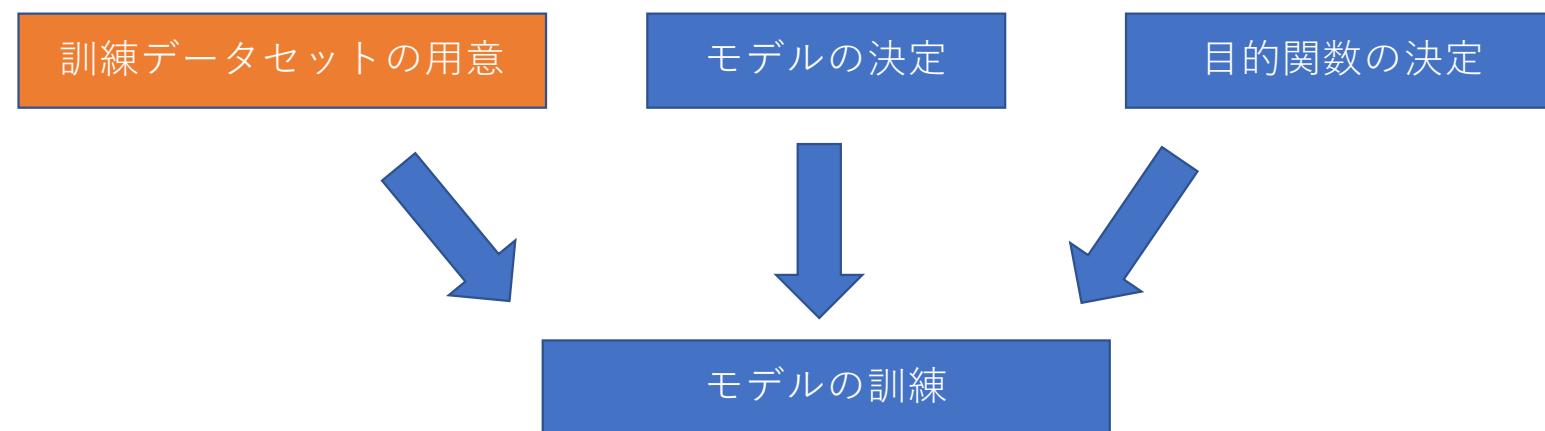
教師あり学習の流れ

回帰問題を中心にモデルを訓練する手順について見ていくこう。



教師あり学習の流れ

訓練データセットの集め方から見ていこう。



訓練データセット：定義

訓練データセットの定義と用語を覚えよう。

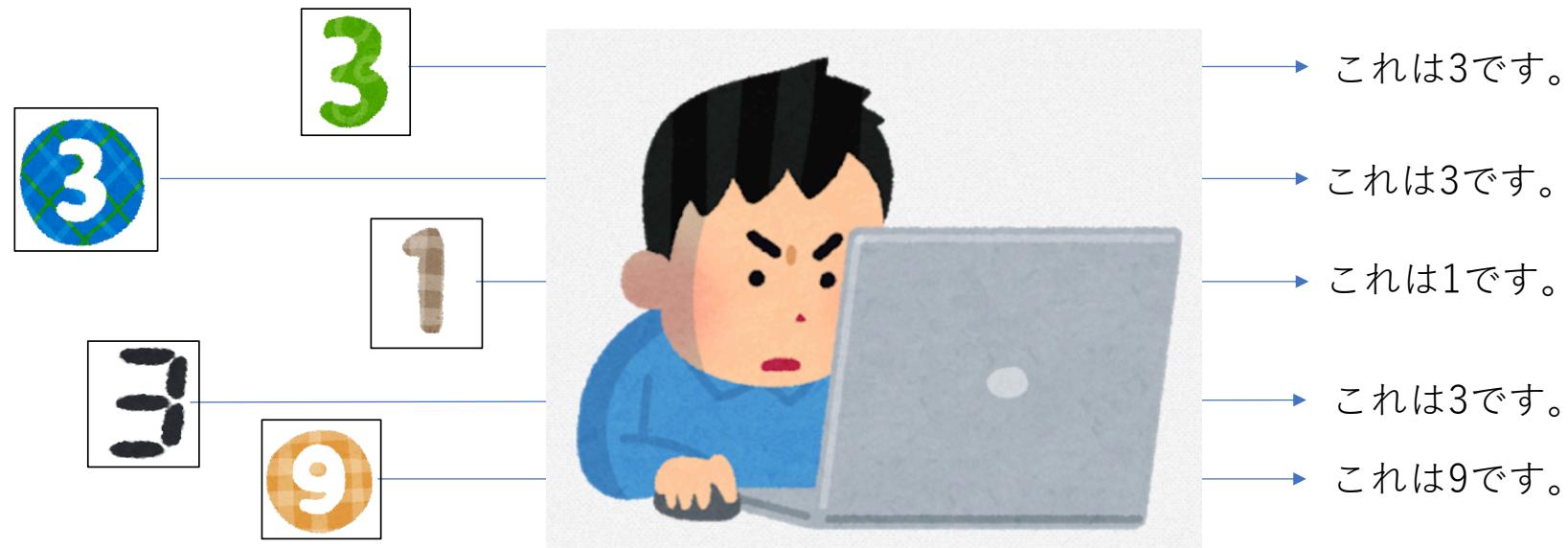
- 入力値 s とそれに対応する出力値 t の組 (s, t) を **データ点** と呼ぶ。
 - s と t はそれぞれ多次元の入出力値でも良い。
- 全部で N 個のデータ点があり、 k 番目のデータ点が (s_k, t_k) と表されるとき、訓練データセットは以下のように定義される。
$$D = \{(s_1, t_1), (s_2, t_2), \dots, (s_N, t_N)\}$$
- 訓練データセットの出力値は、モデルによって予測したい目標となるため、**目標値** または **正解ラベル** とも呼ぶ。

訓練データセット：作成方法の例 1

人間が手作業で訓練データを作る例を見ていこう。

入力値を見て人間が頑張って対応する目標値を与える。

例) 画像に映っている数字を答えるモデルの場合。



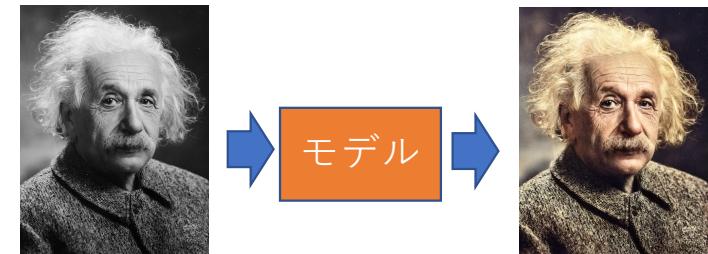
訓練データセット：作成方法の例 2

データを加工処理して自動的に訓練データセットを作る例を見ていこう。

目標値となるデータを加工して対応する入力値を作る。

例) モノクロ画像にそれらしい色をつけて
カラー画像へ変換するモデルの場合

1. カラー画像を大量に集める。
2. カラー画像に対してモノクロ画像に
変換する処理をかける。
(簡単な画像処理ができる)
3. 変換後のモノクロ画像と元のカラー
画像の組を訓練データ点の入力値と
目標値として扱う。



参考 : G. Larsson, et. al, "Learning Representations for Automatic Colorization", ECCV2016.

訓練データセット：作成方法の例 3

目標値を取得できるセンサーを使った場合の訓練データセットの作り方を見ていこう。

データ収集時にのみ目標値を観測できるセンサーを利用する。

例) カメラ画像のみから映っている物体の距離を予測するモデル。

1. カメラと深度を取得できるセンサー※を組み合わせてデータを集める。

※ 3Dスキャナや複眼カメラなど

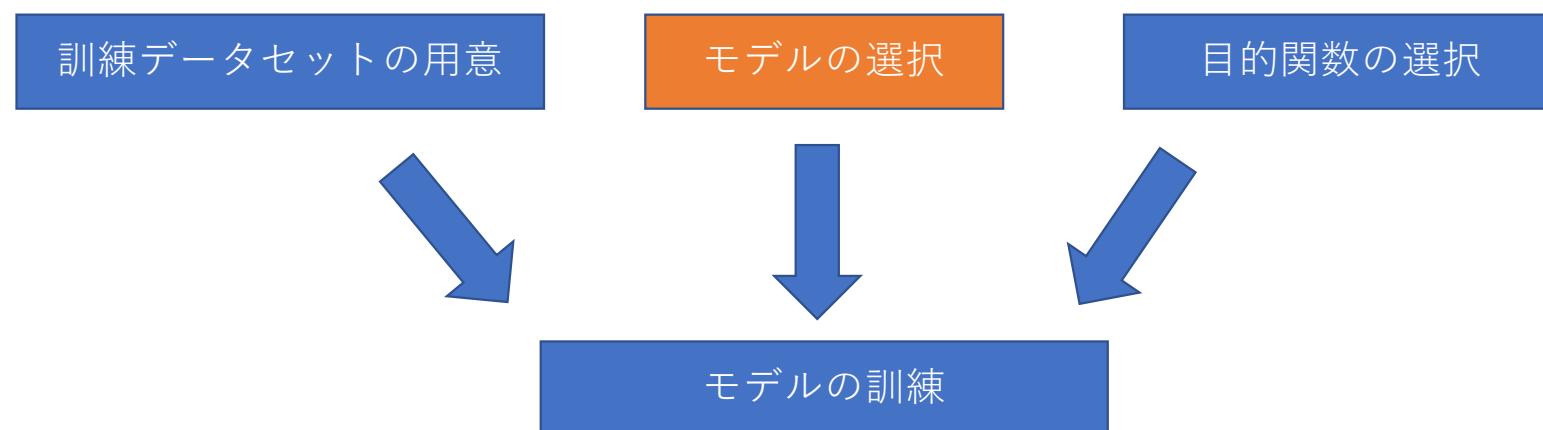
2. カメラ画像と深度情報の組を訓練データ点の入力値と目標値として扱う。



单眼カメラ画像を入力値として深度情報を推定できるモデルを作ることで、必要なセンサーの数が減り、製品のコストを下げるにつながる。

教師あり学習の流れ

データセットの法則性を捉えるためのモデルをどのようにして選択するか見ていこう。



モデル：線形変換

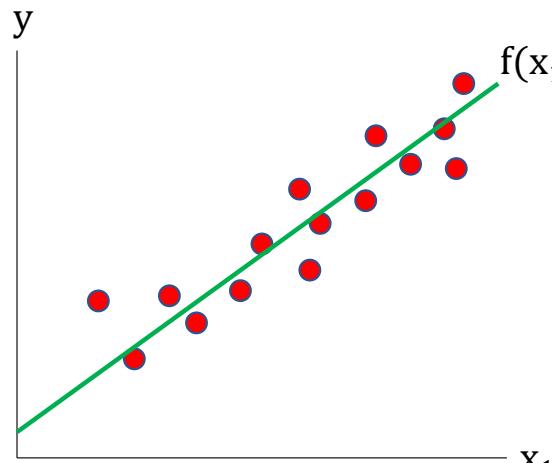
線形変換を行うモデルと非線形変換を行うモデルに大きく分けられる。まずは線形変換を行うモデルを覚えよう。

- 1次式のモデルのとき、入出力変数には**線形**の関係性があり、入力変数から出力変数へ変換を**線形変換**と呼ぶ。

注釈：厳密にはアフィン変換であるが、深層学習ではこれを線形変換と呼ぶ場合が多い。

入力変数が1次元の場合

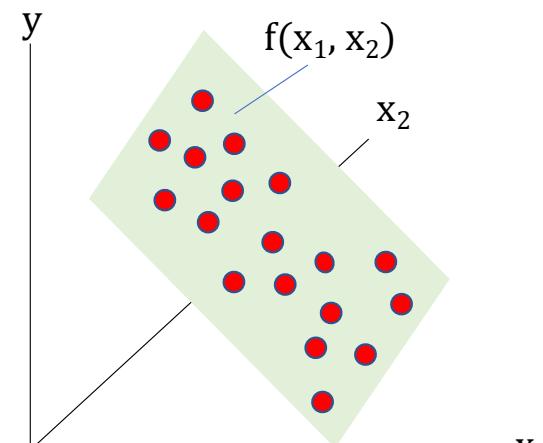
$$f(x_1) = w_1x_1 + b$$



モデルは2次元平面上の直線を表す。

入力変数が2次元の場合

$$f(x_1, x_2) = w_1x_1 + w_2x_2 + b$$



モデルは3次元空間上の平面を表す。

入力変数がn次元の場合

$$f(x_{1..n}) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

入力変数がn次元($n > 2$)以上の場合は
n次元の超平面を表す。

4次元以上の空間と超平面を
普通の人は認知できない。

モデル：多次元の入出力変数

多次元の入出力変数を持つ線形変換のモデルを見ていこう。

- 多次元の入力変数 $x_{1..n}$ から出力変数 $y_{1..m}$ への線形変換を表現するモデルは以下のように書く。

$$y_1 = w_{11}x_1 + w_{12}x_2 + \dots + w_{1n}x_n + b_1$$

$$y_2 = w_{21}x_1 + w_{22}x_2 + \dots + w_{2n}x_n + b_2$$

⋮

⋮

$$y_m = w_{m1}x_1 + w_{m2}x_2 + \dots + w_{mn}x_n + b_m$$

- 高次元の線形変換は行列を使って簡潔に記述することができる。

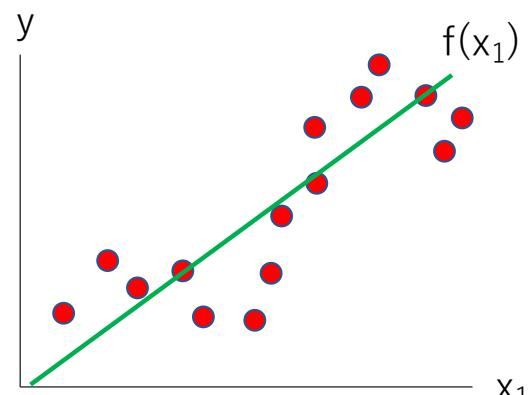
$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad \text{ここで} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & & & \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

モデル：非線形変換

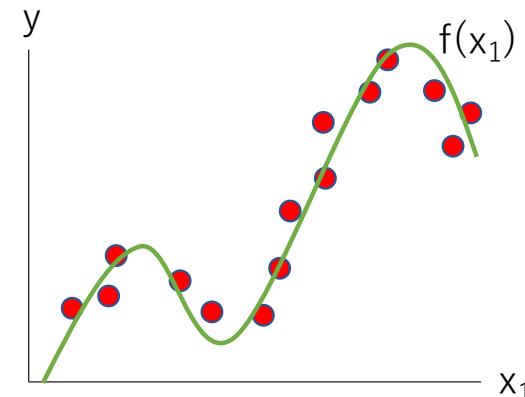
非線形変換を行うモデルは様々なデータの傾向を捉えられることを覚えよう。

- **非線形変換**を表現するモデルの概要

- 入出力変数に線形の関係性がないモデル
 - 例として、 x^2 , $\sin(x)$, $\log(x)$ などは非線形変換を行う関数
- 1次元の入力変数では曲線、2次元では曲面を表現することができる。



線形変換の場合、このデータの法則性を捉えることができない。



非線形変換を使えば法則性をうまく捉えることができる。

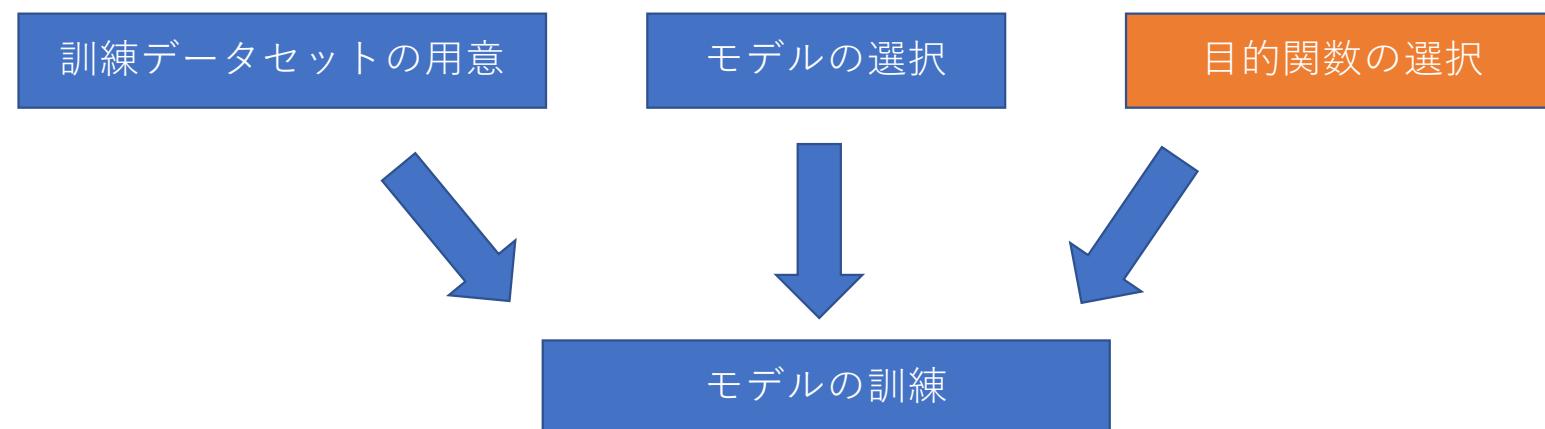
モデル：モデルの選択

モデルの表現力についての基本的な概念を学ぼう。

- モデルを選択することは、データセットの背景にある法則に仮説を立てることと同義である。
- モデルが多様な法則性を近似できることを**モデルの表現力**が高いと呼ぶ。当然、線形変換よりも非線形変換を行うモデルの方が表現力が高い。
- 常に表現力の高い複雑な非線形変換を選択すれば良いわけではないことに注意する。
 - 複雑なモデルほど訓練に必要なデータ点数を多く必要とする。
 - 現実の問題では線形変換の表現力で十分な場合も多い。

教師あり学習の流れ

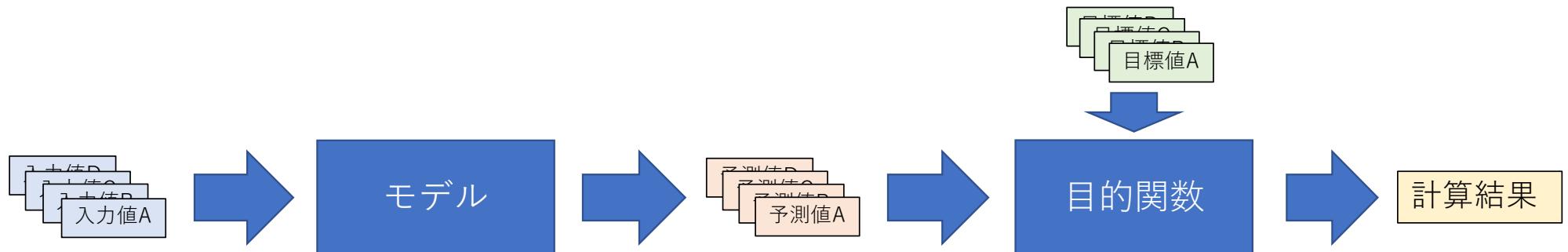
学習アルゴリズムが自動的に良いパラメータを見つけ出すための指標を与える目的関数について学ぼう。



目的関数：目的関数とは何か

目的関数の使い方のイメージをつかもう。

- ・モデルの良いパラメータを自動的に見つけるために
パラメータの良し悪しを表す定量的な指標が必要となる。
- ・教師あり学習では目標値が与えられるため、訓練中のモデルの
予測値が目標値にどれだけ近いかを**目的関数**で計算する。



- ・モデルの訓練時には、できるだけ目的関数の結果の良くなる
パラメータを目標に探索する。

目的関数：目的関数の選択

目的関数にはどのようなものがあるのか、その概要を学ぼう。

- 回帰問題の場合は**平均二乗誤差**や**平均絶対誤差**などの訓練データセットの目標値とのずれを計算する関数を利用する。
- 目的関数の選択にはデータセットに含まれる誤差などの性質を考慮する。
- 平均二乗誤差や平均絶対誤差は、モデルの予測値が全て目標値と一致するときに0となり、値が0に近いほど良い結果みなす。これらの目的関数の計算結果は**損失**とも呼ばれる。

目的関数：平均二乗誤差

回帰問題の目的関数としてよく使われる平均二乗誤差を学ぼう。

- 平均二乗誤差の定義

$$L = \frac{1}{N} \sum_{n=1}^N (t_n - y_n)^2 \quad N \text{は評価するデータ点の数}$$

- 計算例

入力データ	予測y	目標t
入力1	2.7	2.0
入力2	1.5	1.5
入力3	3.0	2.5
入力4	0.8	1.0
入力5	3.1	3.0
入力6	0.4	0.5

$$\begin{aligned} L &= \frac{1}{6} \{(2.0 - 2.7)^2 + (1.5 - 1.5)^2 + (2.5 - 3.0)^2 + \\ &\quad + (1.0 - 0.8)^2 + (3.0 - 3.1)^2 + (0.5 - 0.4)^2\} \\ &= \frac{1}{6} \{(-0.7)^2 + 0^2 + (-0.5)^2 + 0.8^2 + (-0.1)^2 + 0.1^2\} \\ &= \frac{1}{6} \times 1.4 \approx 2.33 \end{aligned}$$

目的関数：平均絶対誤差

平均絶対誤差について定義と例を通して学ぼう。

- 平均絶対誤差の定義

$$L = \frac{1}{N} \sum_{n=1}^N |tn - yn| \quad N \text{は評価するデータ点の数}$$

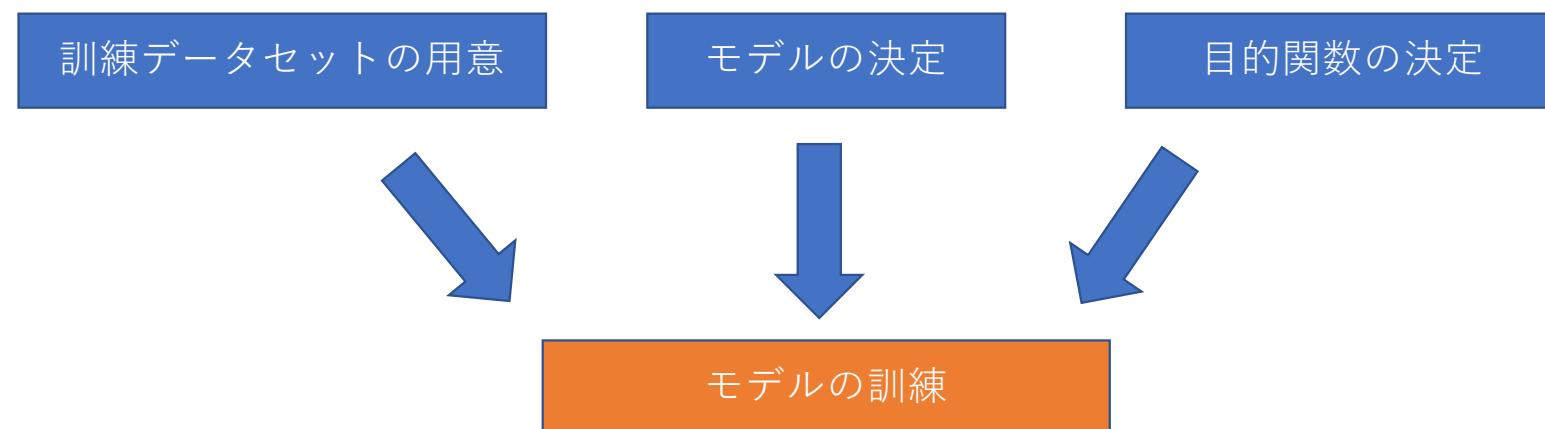
- 計算例

入力データ	予測y	目標t
入力1	2.7	2.0
入力2	1.5	1.5
入力3	3.0	2.5
入力4	0.8	1.0
入力5	3.1	3.0
入力6	0.4	0.5

$$\begin{aligned} L &= \frac{1}{6} \{ |2.0 - 2.7| + |1.5 - 1.5| + |2.5 - 3.0| + \\ &\quad + |1.0 - 0.8| + |3.0 - 3.1| + |0.5 - 0.4| \} \\ &= \frac{1}{6} \{ |-0.7| + |0| + |-0.5| + |0.2| + |-0.1| + |0.1| \} \\ &= \frac{1}{6} \times 1.6 \approx 2.67 \end{aligned}$$

モデルの訓練の流れ

モデルの訓練方法を簡単な例を使って学ぼう。



モデルの訓練：最適化手法

訓練を行うための最適化手法の基本的な概念を覚えよう。

- 損失を小さくするためにモデルの良いパラメータを求める手法を**最適化手法**と呼ぶ。
- 一部のモデルでは最適なパラメータを**解析的**に（式を解いて）求めることができる。
- モデルの複雑さの観点から、数値計算を使って現実的な時間で（最適とは限らないが）良いパラメータを探索する**数値的**な解法を用いる場合もある。

モデルの訓練：簡単な例

簡単なモデルと訓練データセットの問題例を見てみよう。

問題：以下が与えられた時、 L を最小にする w を求めたい。

- モデル： $y = wx$
- 目的関数： $L = \frac{1}{N} \sum_{n=1}^N (t_n - y_n)^2$
- 訓練データ：

入力値x	目標値t
1.0	2.0
2.5	5.1
3.0	6.2
5.0	9.6

目的関数 L をパラメータ w の関数として整理できる。

$$\begin{aligned} L &= \frac{1}{4} \{(2.0 - 1.0w)^2 + (5.1 - 2.5w)^2 + \\ &\quad (6.2 - 3.0w)^2 + (9.6 - 5.0w)^2\} \\ &= 10.3125w^2 - 40.675w + 40.1525 \end{aligned}$$

モデルに複雑な関数を採用する場合は L は更に複雑な関数となる。

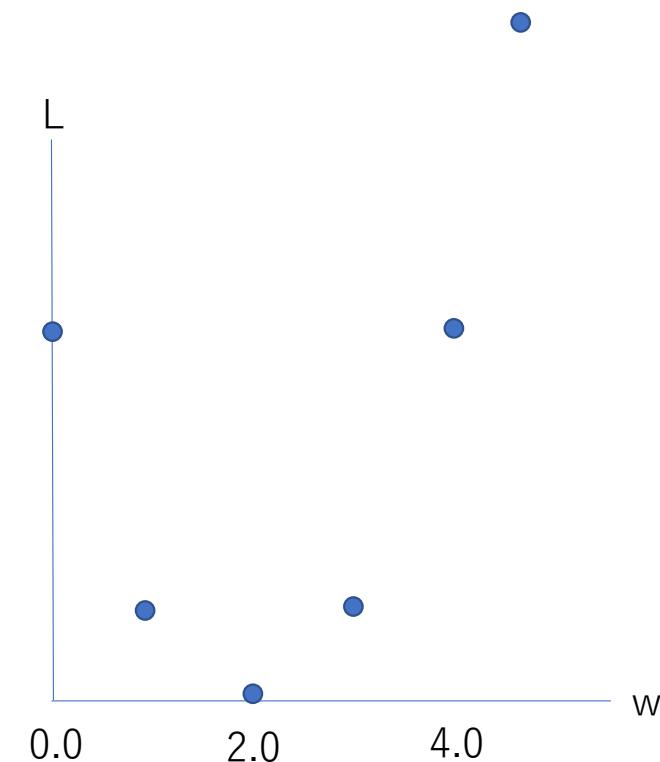
モデルの訓練：簡単な例

パラメータを変更したときに目的関数の値がどのように変わるか見てみよう。

- 様々な w での目的関数の計算を試すと

- $w = 0.0$ の場合 $L = 40.15$
- $w = 1.0$ の場合 $L = 9.79$
- $w = 2.0$ の場合 $L = 0.525$
- $w = 3.0$ の場合 $L = 10.94$
- $w = 4.0$ の場合 $L = 42.45$
- $w = 5.0$ の場合 $L = 94.59$

- モデルが複雑な場合にどのようにして目的関数の計算結果を小さくする w を効率よく発見できるか？



モデルの訓練：数値的解法

モデルを訓練する基本的な手法である勾配降下法の概要を学ぼう。

• 勾配降下法の概要

- 目的関数の**勾配**をヒントとしてパラメータを繰り返し更新する方法。

注釈) 勾配降下法は目的関数が微分可能な関数であるときに利用できることに注意する。

• 関数の勾配

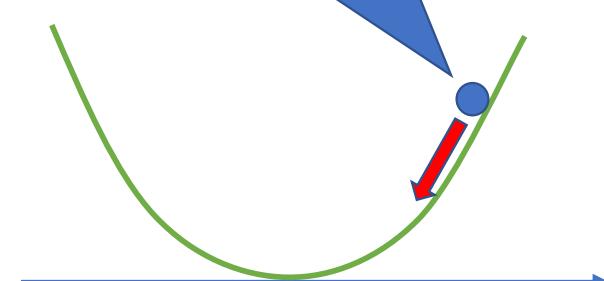
- イメージは

「関数のある点にボールを置いた場合に、
どの方向に、どのくらいの勢いで
転がっていくかを示す指標」

注釈) 2次関数の場合は関数の接線の傾きを勾配と呼ぶ。

- 勾配は目的関数の微分を
計算することで求めることができる。

ここにボールを置くと左側に向かって勢いよく転がっていく。



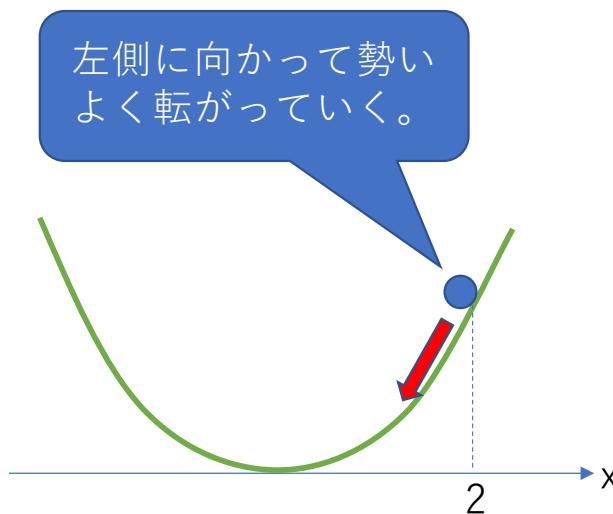
勾配のイメージ

モデルの訓練：数値的解法

関数の勾配を例を通して学ぼう。

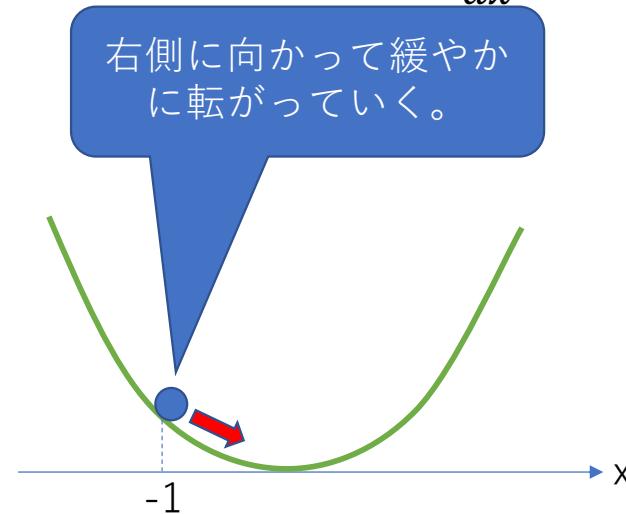
- 勾配計算の例

- 関数 $f(x)=x^2$ のとき、勾配は $f(x)$ の微分 $\frac{df(x)}{dx} = 2x$ を計算する。



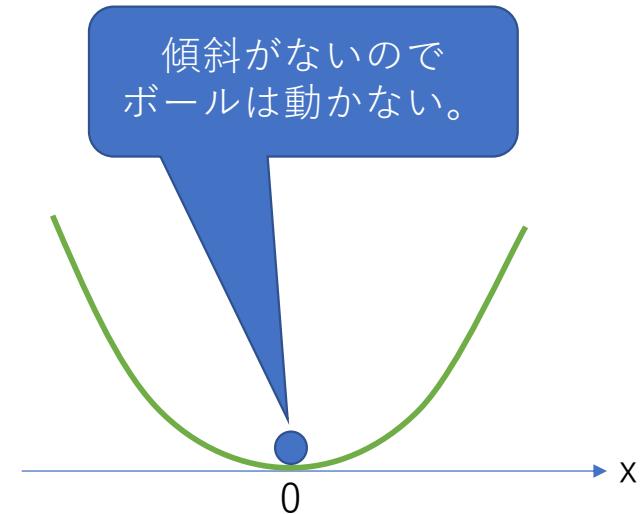
$x=2$ の場合の勾配は4

勾配が正の場合は左傾斜



$x=-1$ の場合の勾配は-2

勾配が負の場合は右傾斜



$x=0$ の場合の勾配は0

勾配が0の場合は傾斜なし

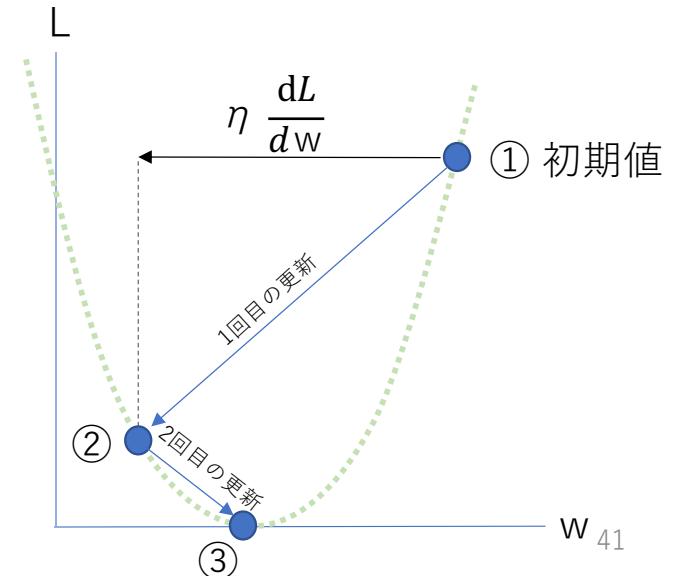
モデルの訓練：数値的解法

勾配降下法のアルゴリズムの概要について学ぼう。

- 勾配降下法でやりたいこと
 - 目的関数 L をパラメータ w の関数として、目的関数の谷を目指すように w を更新する。
 - w は適当な初期値から始め、 w の更新による L の変化量が十分に小さくなる（**収束**）まで繰り返す。

- パラメータ更新の概要
 - w での勾配 $\frac{dL}{dw}$ を計算する。
 - $w - \eta \frac{dL}{dw}$ を新しい w として更新する。

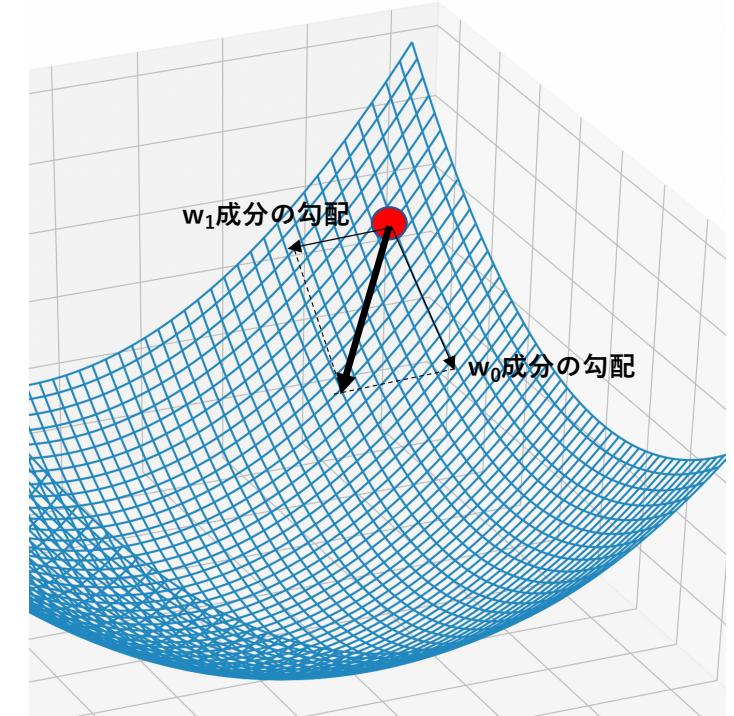
$\eta (>0)$ は**学習率**と呼ばれ、勾配による更新量を調整するために与える重要な定数。



モデルの訓練：数値的解法

パラメータが複数ある場合の勾配降下法の概要を学ぼう

- パラメータを2つ以上持つモデルの場合
 - 目的関数は高次元の複雑な形となる。
 - 各パラメータ成分の勾配は偏微分によって計算することができる。
 - w_k 成分の勾配を計算後、 w_k を勾配降下法の更新式で更新する。
 - 問題点
 - 非常に多くのパラメータを持つモデルの場合 勾配の計算に多くの時間が必要となる。
- 次章のニューラルネットワークでは効率の良い勾配計算方法が知られている。



モデルのパラメータが w_0 と w_1 の場合、目的関数は3次元空間に描写される。

モデルの訓練：勾配降下法の種類

勾配降下法には様々なバリエーションがあることを覚えよう。

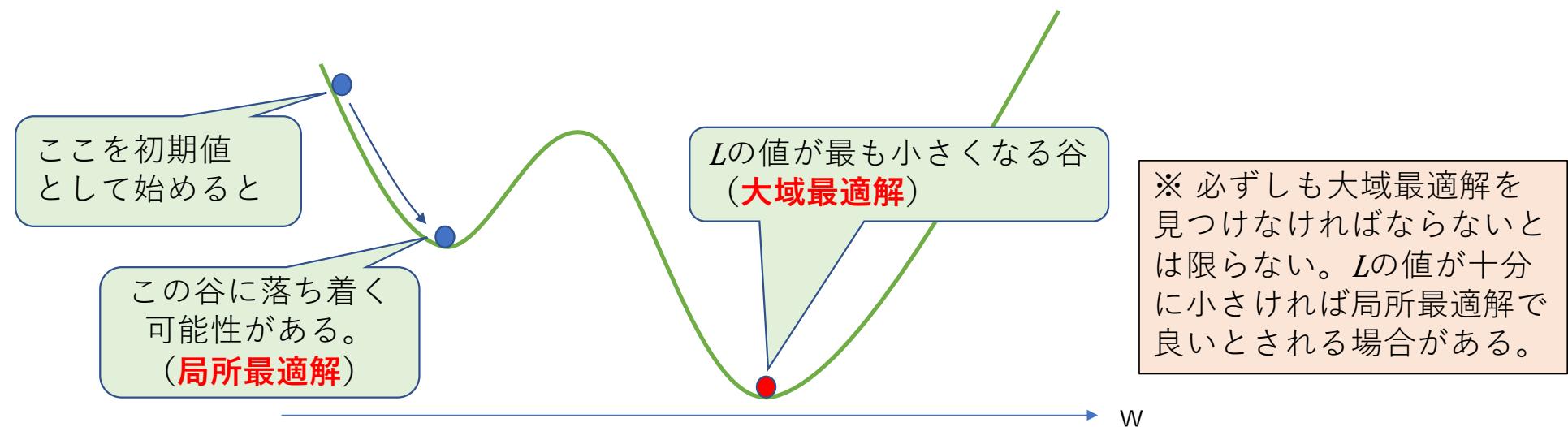
- 勾配降下法に基づく最適化手法の例
 - 確率的勾配降下法 (SGD) [文献]
 - MomentumSGD [文献]
 - AdaGrad [文献]
 - RMSprop [文献]
 - AdaDelta [文献]
 - Adam [文献]
 - ...
- これらの最適化手法を利用すれば必ず最適なパラメータを得られるわけではないことに注意。

パラメータの更新式などに
それぞれ工夫がある。

モデルの訓練：局所最適解

勾配降下法には局所最適解の問題があることを覚えよう。

- 目的関数 L に複数の谷がある場合、勾配降下法で解を求めようとすると最適ではない点で収束する場合がある。

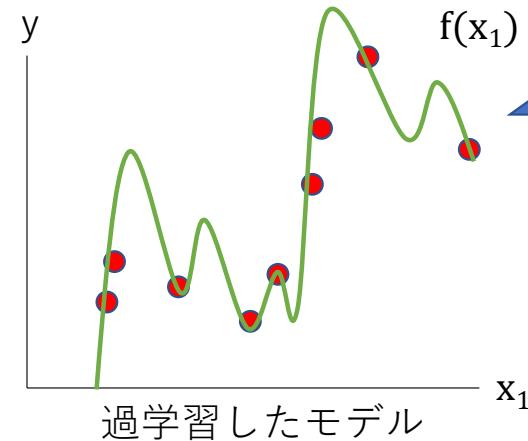
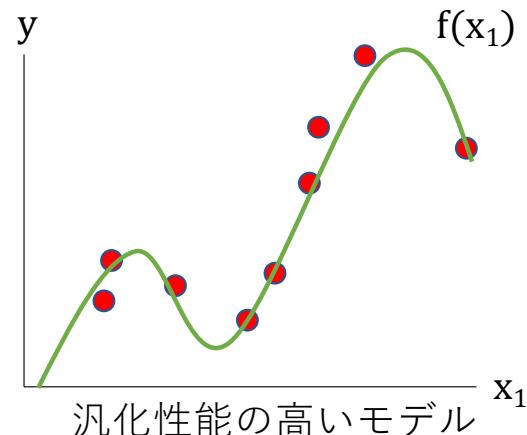


- 目的関数 L の値が十分に小さくなる解を見つけるために学習率や最適化手法の選択などが重要となる。

モデルの訓練：汎化性能と過学習

訓練データと同じ予測ができるようになるだけではなく、汎化性能を得ることも重要であることを学ぼう。

- 左図のモデルのように訓練データ点のカバーしていない範囲もそれらしい予測ができるることを**汎化性能**が高いと呼ぶ。
- 右図のモデルのように訓練データ点にのみ過剰に適合してしまうことを**過学習**と呼ぶ。

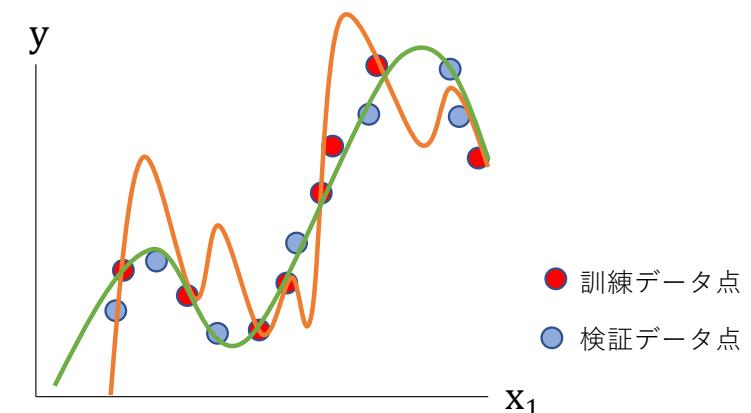


- 過学習はモデルの複雑さや次元数に対して訓練データ点の数が少ない場合などに起こる。

モデルの訓練：モデルの検証評価

過学習をしていないか確認するために検証による評価も必要であること学ぼう。

- ・モデルが過学習をしていないかを確かめるために**検証**による評価を訓練と同時にを行うことがある。
- ・用意したデータセットを訓練用と検証用のデータセットに分割する。
- ・検証データセットは訓練時のパラメータ更新には寄与せずに評価にのみ使い、訓練中のモデルの汎化性能を確かめる。



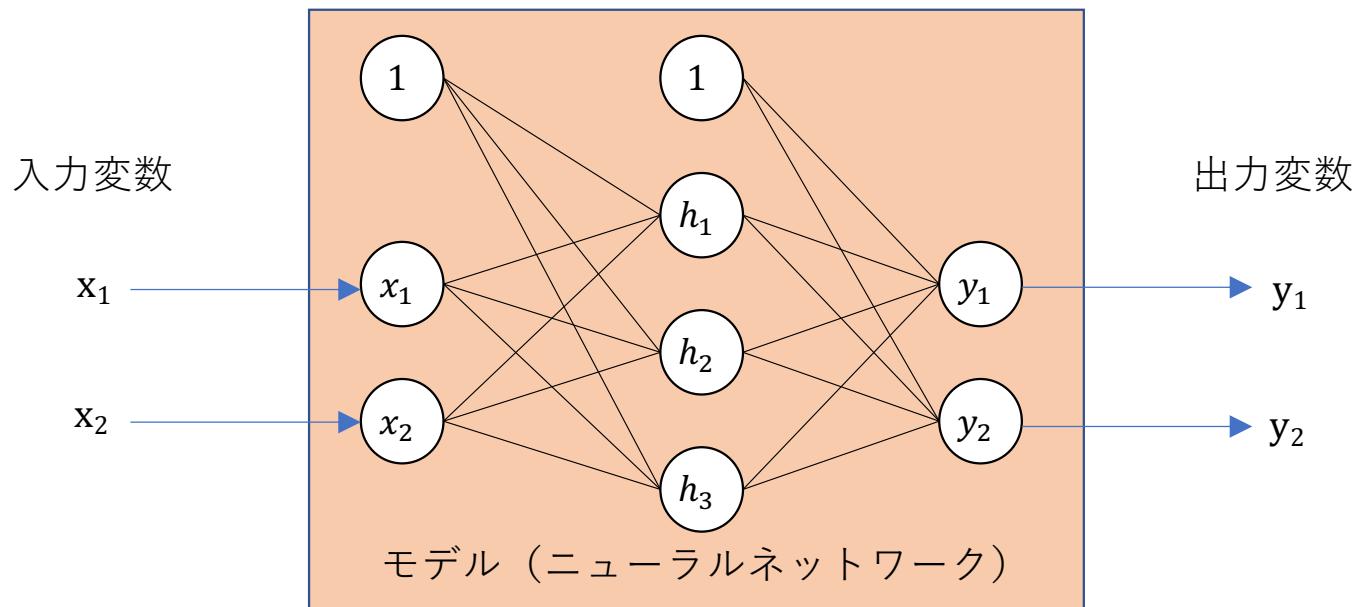
ニューラルネットワークと 深層学習

本章ではニューラルネットワークと深層学習の基礎を学ぶ。

ニューラルネットワーク

ニューラルネットワークはモデルの一つであることを学ぼう。

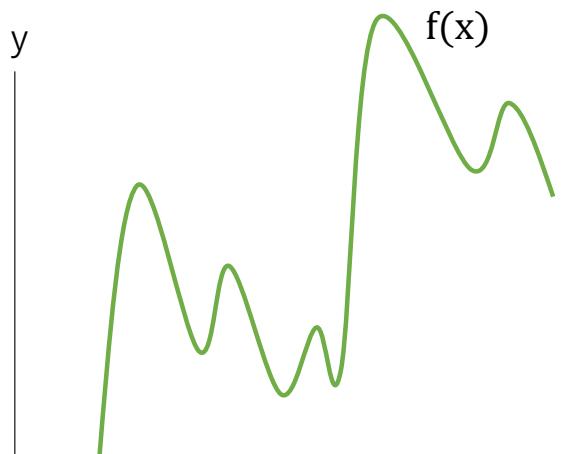
- ・**ニューラルネットワーク(NN)**は入力変数から出力変数への非線形変換を表現するモデルの一つである。



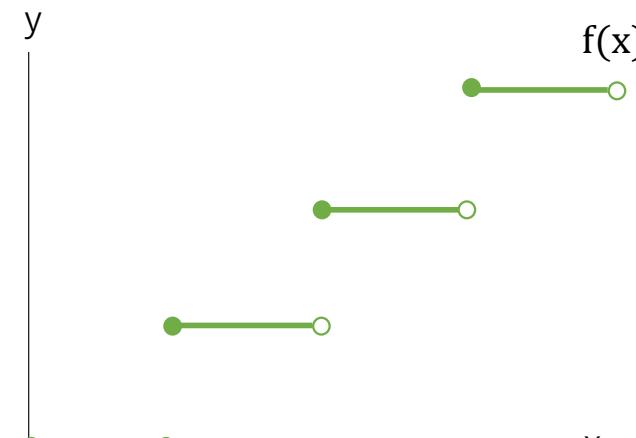
ニューラルネットワークの表現力

ニューラルネットワークの強力な表現力を示す定理について
知っておこう。

- ニューラルネットワークの普遍性定理の概要
 - 中間層一層で構成される全結合型のニューラルネットワークは任意の連続関数を任意の精度で近似することができる。
[G. Cybenko, 1989], [K. Hornik et, al., 1989]



連続関数の例



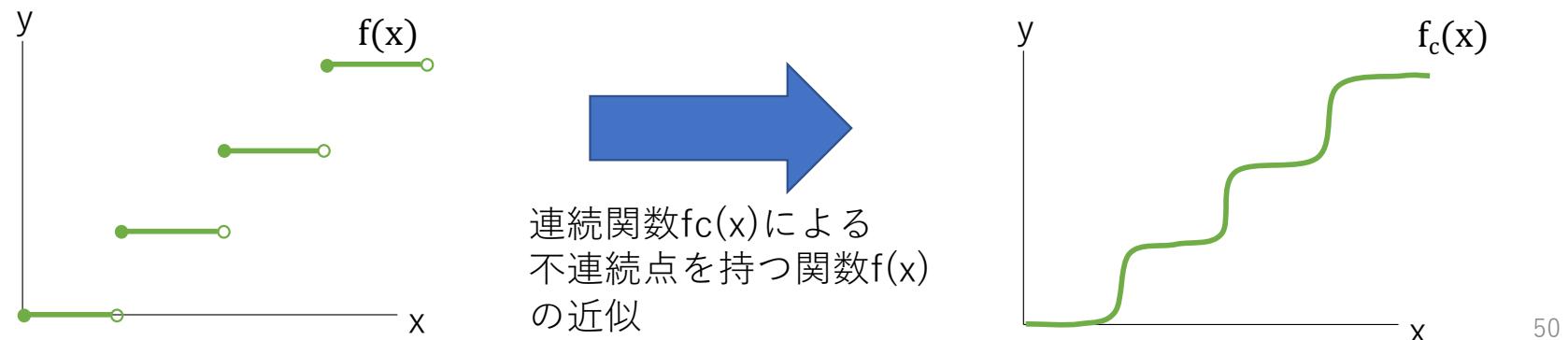
不連続点を持つ関数の例

ニューラルネットワークの表現力

普遍性定理の注意点について知っておこう。不連続点を持つ関数を学習したい場合でも連続関数による近似で良い場合があること覚えよう。

- 普遍性定理の注意点

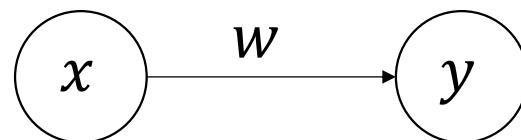
- 任意の関数と完全に同じ計算ができるのではなく、限りなくそれに近い関数（近似関数）を表現できることを示している。
- ニューラルネットワークの表現力について述べているもので、任意の関数を訓練により獲得できる意味ではない。
- 不連続点を持つ関数であっても連続関数による近似で実用上十分な場合も多い。



ニューラルネットワーク：基本構造

計算グラフの概念を学ぼう。

- ニューラルネットワークは**計算グラフ**と呼ばれる関数の図式表現で表される。
- $y = wx$ の計算グラフ



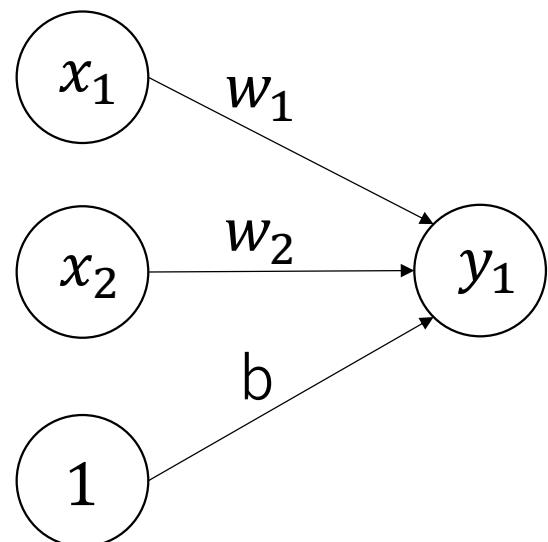
計算グラフ中の円を**ノード**と呼び、入出力変数や中間的な計算結果を格納する変数を表す。

ノード間を結合する矢印を**エッジ**と呼ぶ。エッジ上の記号wはノード間の**重み**を表す。始点となるノードの値xに重みwの積xwを終点のノードの値とする。

ニューラルネットワーク：基本構造

簡単な線形変換を行うモデルの計算グラフ表現を学ぼう。

$y_1 = w_1x_1 + w_2x_2 + b$ を表す計算グラフ
(2次元の入力変数の線形変換)



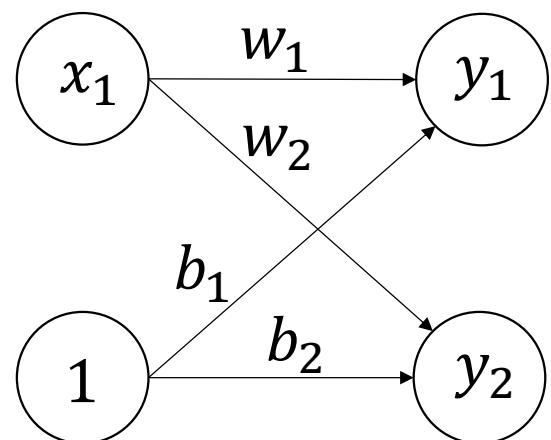
ノードに複数のエッジが入ってくる場合は、それぞれのエッジから入ってくる値の和を取る。

定数1のノードはバイアス項と呼ばれる。

ニューラルネットワーク：基本構造

簡単な線形変換を行うモデルの計算グラフ表現を学ぼう。

$y_1 = w_1x_1 + b_1$
 $y_2 = w_2x_1 + b_2$ を表す計算グラフ (2次元の出力変数への線形変換)



ノードから複数のエッジが出ていく場合は、
それぞれのエッジごとに重みを持つ。

ニューラルネットワーク：基本構造

多次元の入出力変数の線形変換を行う計算グラフを学ぼう。

入力層と出力層の概念について学ぼう。

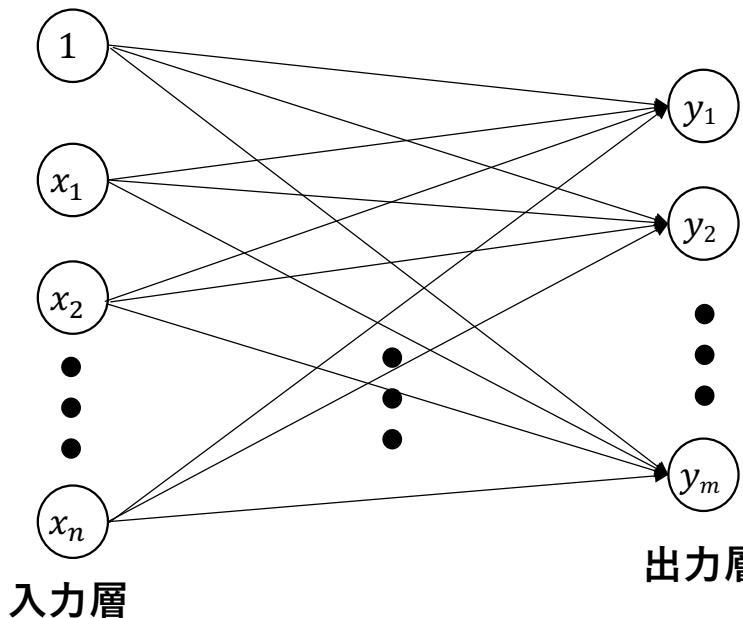
$$y_1 = w_{11}x_1 + w_{12}x_2 + \dots + w_{1n}x_n + b_1$$

$$y_2 = w_{21}x_1 + w_{22}x_2 + \dots + w_{2n}x_n + b_2$$

⋮

⋮

$$y_m = w_{m1}x_1 + w_{m2}x_2 + \dots + w_{mn}x_n + b_m$$



を表す計算グラフ

入力変数（出力変数）に対応するノードをまとめて入力層（出力層）と呼ぶ。

出力層 (計算グラフの各エッジの重みは省略)

ニューラルネットワーク：活性化関数

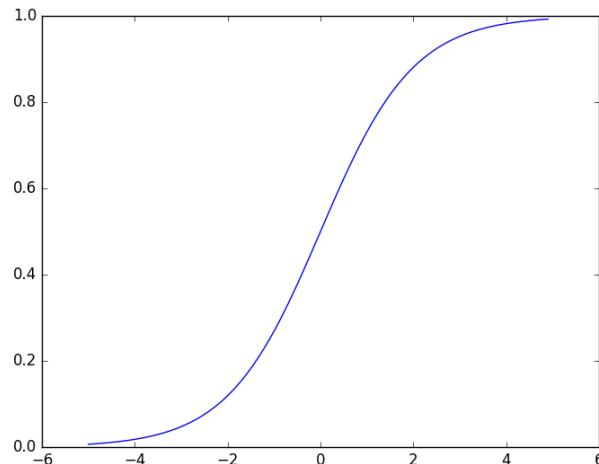
ニューラルネットワークが非線形変換を実現するために基本となる活性化関数について学ぼう。

- ニューラルネットワークでは非線形変換を表現するために、**活性化関数**を利用する。
- 活性化関数は勾配の計算のため微分可能な関数を利用する。
- 活性化関数 $a(u)$ の例
 - シグモイド関数 : $a(u) = \frac{1}{1+e^{-u}}$
 - ハイパボリックタンジェント : $a(u) = \tanh(u)$
 - ReLU : $a(u) = \max(0, u)$

ニューラルネットワーク：活性化関数

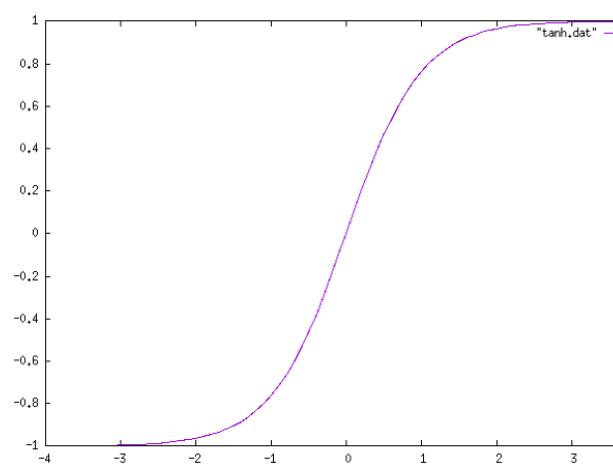
活性化関数のグラフと性質を学ぼう。

シグモイド関数
(ロジスティックシグモイド関数)



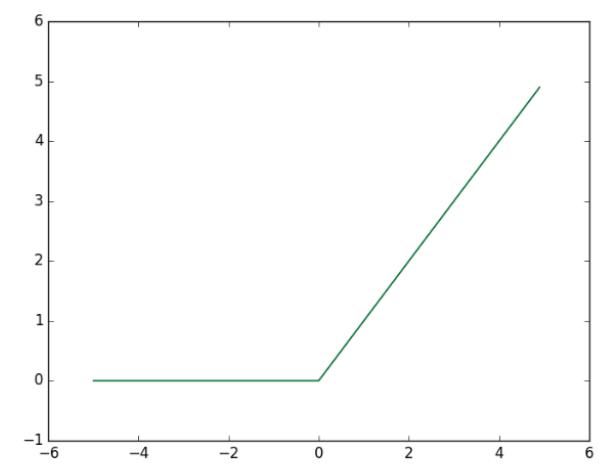
従来よく用いられてきた活性化関数。入力値を0から+1にマッピングする。

ハイパボリックタンジェント



シグモイド関数を線形変換した関数。原点を通ることでシグモイド関数よりも学習効率が良いとされる。

ReLU

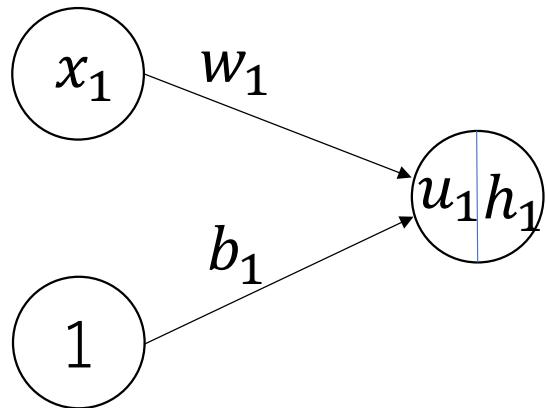


実装が簡単であり、後述の深いNNを利用する場合にシグモイド関数やtanhよりも学習効率が良い。

ニューラルネットワーク：活性化関数

活性化関数を加えた計算グラフを見てみよう。

- $h_1 = a(w_1x_1 + b_1)$ の計算グラフ



中に線の入っているノードでは活性化関数の適用を意味する。

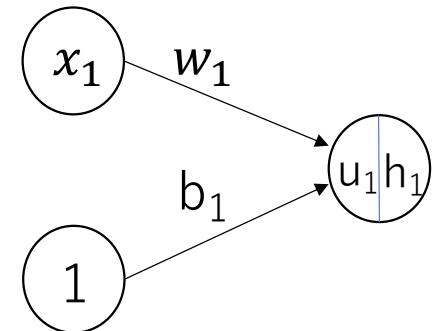
半円の左側・右側の変数 u と h はそれぞれ
 $u_1 = w_1x_1 + b_1$
 $h_1 = a(u_1)$
を表す。

半円の左側の変数は省略して図示されることが多い。
(特に次に説明する中間層のある場合)

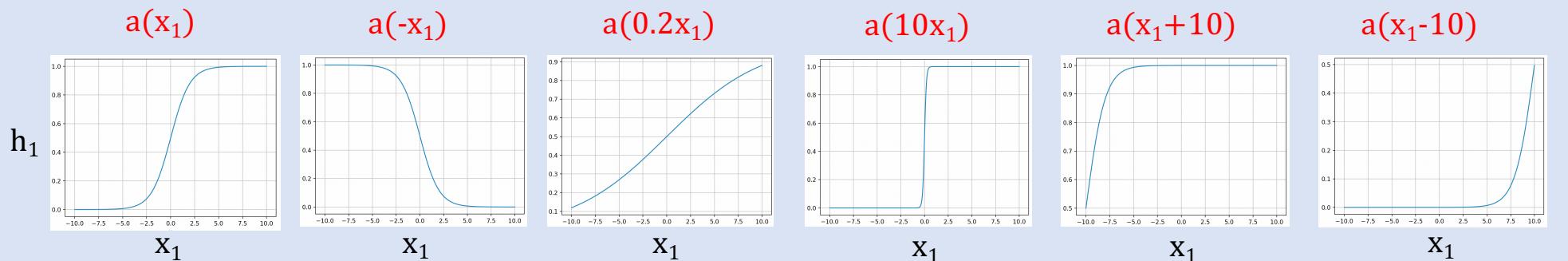
ニューラルネットワーク：活性化関数

パラメータを変更すると活性化関数の形がどのように変わるか見てみよう。

- $h_1 = a(w_1x_1 + b_1)$ はパラメータ w_1, b_1 を調整することで様々な非線形変換を表現できる。



活性化関数 $a(u)$ としてシグモイド関数を利用する場合

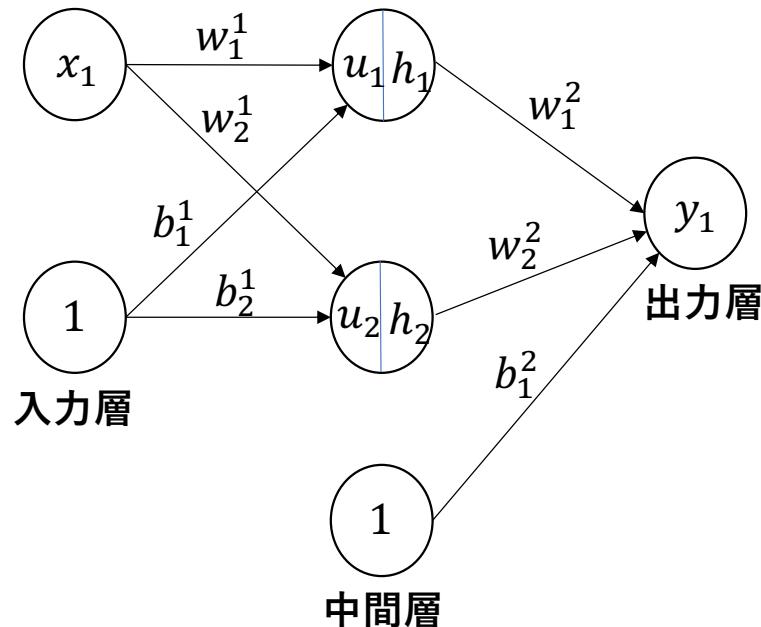


- NNでは活性化関数を通した様々な非線形変換を組み合わせることでより複雑な非線形変換を表現する。

ニューラルネットワーク：中間層

中間層を持つニューラルネットワークが表現する関数を学ぼう。

- 中間層（または隠れ層）のあるニューラルネットワーク
- 中間層のノードでは活性化関数が適用される。
※分類問題では出力層のノードにも活性化関数（ソフトマックス関数など）を適用する。



$$\begin{aligned} h_1 &= a(w_1^1 x_1 + b_1^1) \\ h_2 &= a(w_2^1 x_1 + b_2^1) \\ y_1 &= w_1^2 h_1 + w_2^2 h_2 + b_1^2 \\ &= \underline{w_1^2 a(w_1^1 x_1 + b_1^1) + w_2^2 a(w_2^1 x_1 + b_2^1) + b_1^2} \end{aligned}$$

中間層を導入することで
入れ子の複雑な関数を表現できる。

ニューラルネットワーク：全結合型

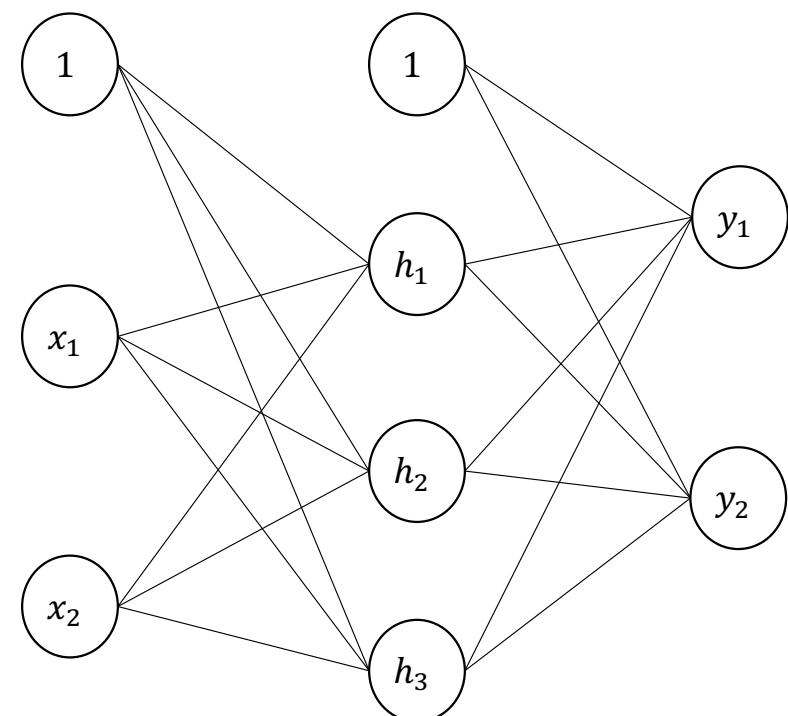
全結合型ニューラルネットワークを覚えよう。

- **全結合型ニューラルネットワーク**

- あるノード※が一つ前の層の全てのノードと結合しているニューラルネットワーク。

※入力層とバイアス項を除く

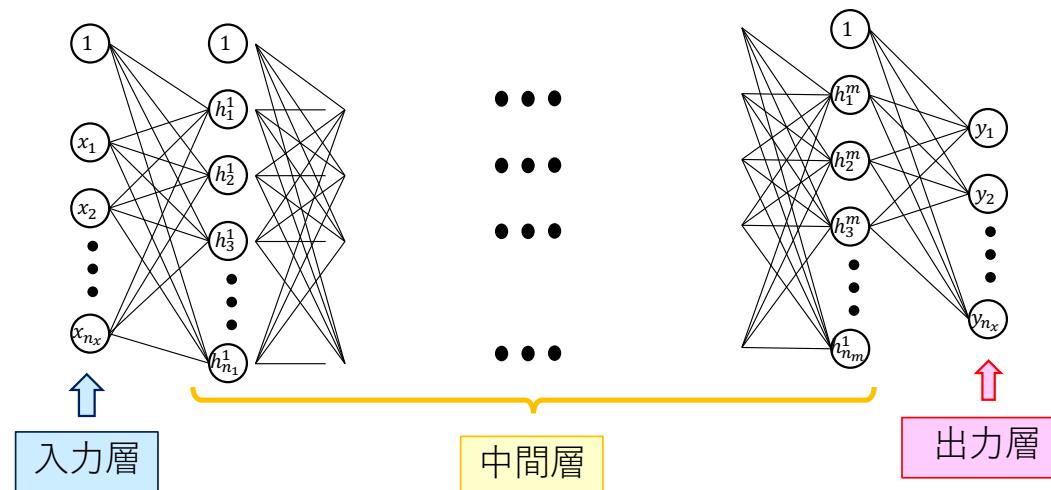
- 全結合型以外にも**畳み込み型**や**回帰型**などの種類があり、1つのネットワークに複数種類が混ざって現れることがある。



深層学習

深層学習とニューラルネットワークの関係を学ぼう。

- 中間層のたくさんある深いネットワークを
ディープニューラルネットワーク (DNN) と呼ぶ。



- 深層学習とはDNNを用いた機械学習の手法やその周辺の研究領域のこと。

層を深くする意味

層を深くすると何故良いとされるのか知っておこう。

- 直感的な理解
 - 非線形変換を階層的に何度も行うことで、より複雑な変換を実現する。
- 理論的な研究
 - 中間層 1 層の全結合型NNは任意の関数を近似する表現力を持っているが、非現実的な数のノードを必要とする場合がある。
 - 任意の連続関数を近似する時、多くの場合に深い層のモデルの方が必要なノード数（パラメータ数）が少なく済むことを説明する研究が行われている。[S. Liang and R. Srikant, 2017] など

注釈1) むやみに層を増やせばいいというわけではない。工夫なしに層を深くしすぎると訓練が進まなくなる現象が知られている。（勾配消失問題）

注釈2) 層が深い場合にシグモイド関数を活性化関数として利用すると上記の問題が起こりやすいことが知られている。

そのため、DNNではReLU関数などを活性化関数として利用した方が良いとされる。

ニューラルネットワークの訓練方法

ニューラルネットワークのモデルを訓練する手順を学ぼう。
教師あり学習の章で学んだ手順と同じであることを確認しよう。

1. 訓練データセットを作る
2. ネットワークモデルの決定
3. 目的関数の決定
4. 最適化手法の選択
5. ネットワークモデルの訓練

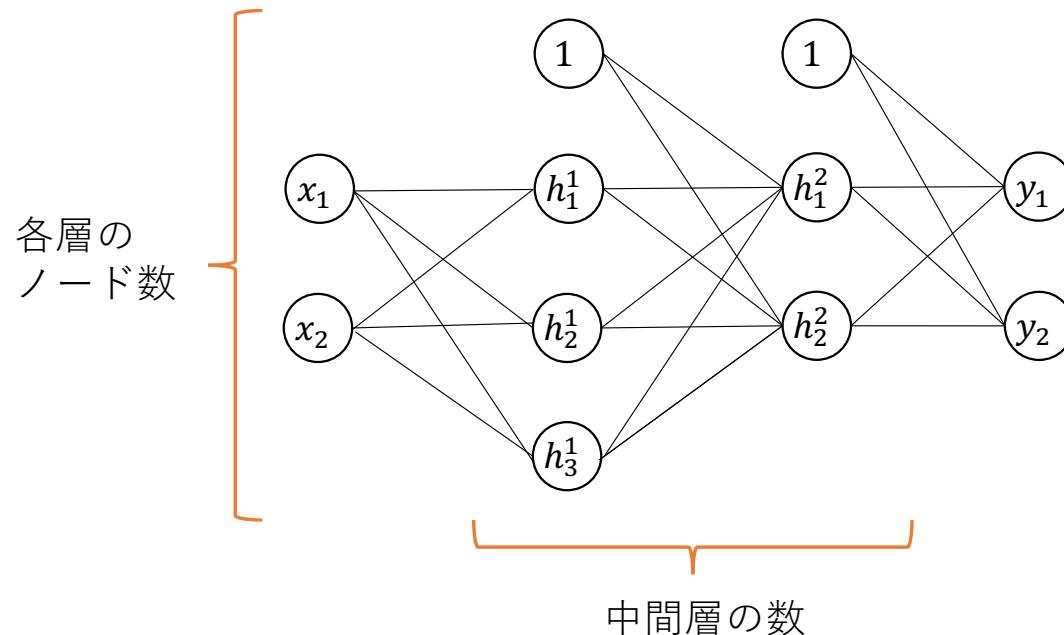
教師あり学習の章で
学んだことと同じ。

ネットワークモデルの決定

入出力変数が定義されているときに、全結合型のニューラルネットワークの構造を決める3つの要素を覚えよう。

- 全結合型ニューラルネットワークの構造を決める3つの要素

- 中間層の数
- 各層のノード数
- 活性化関数



モデルの訓練：訓練の手順

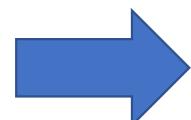
教師あり学習で学んだ訓練の手順をおさらいしよう

教師あり学習の章で学んだ訓練の流れ

準備：全てのパラメータをランダムな値で初期化する。

1. 各パラメータ成分の勾配を計算する。
2. 勾配から最適化手法を用いてパラメータを更新する。
3. 収束するまで1-2を繰り返す。

NNは多くのパラメータを持っているので、1. で各パラメータ成分の勾配を愚直に計算すると多くの時間を必要としてしまう。

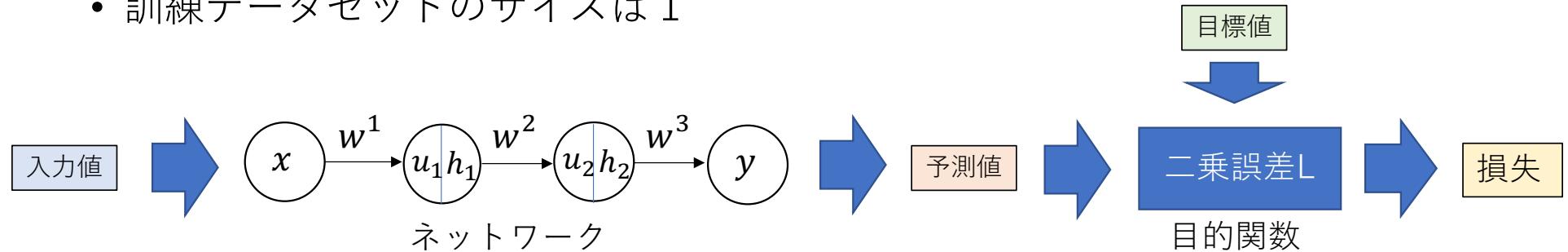


NNでは**誤差逆伝播法（バックプロパゲーション）**と呼ばれる勾配を効率よく計算できるアルゴリズムを利用できる。

モデルの訓練：誤差逆伝播法

誤差逆伝播の概要を簡単な例を通して学ぼう。

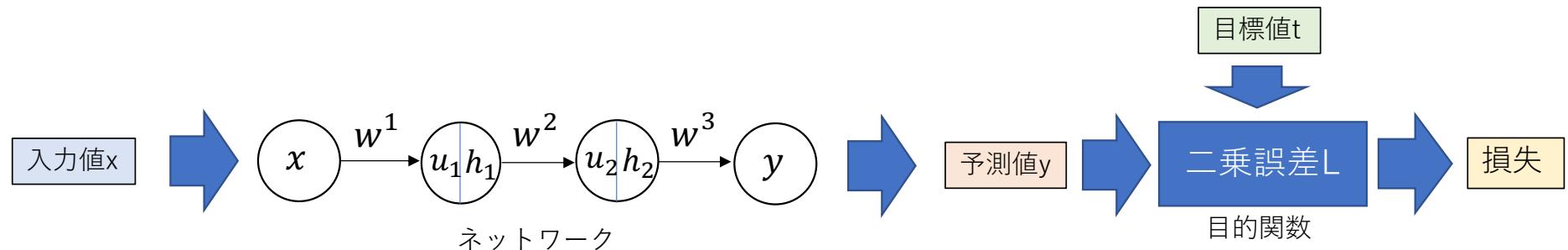
- 簡単な例
 - 中間層2、各層のノード数1のネットワーク
 - 活性化関数はシグモイド関数
 - 目的関数は平均二乗誤差
 - 訓練データセットのサイズは1



- 誤差逆伝播法の目的
 - 3つのパラメータ w^1, w^2, w^3 成分の勾配 $\frac{\partial L}{\partial w^1}, \frac{\partial L}{\partial w^2}, \frac{\partial L}{\partial w^3}$ を効率よく計算する。

モデルの訓練：誤差逆伝播法

誤差逆伝播による勾配計算の効率化のイメージを覚えよう。



- 微分の**連鎖律**を利用し、出力層側のパラメータから順番に計算する。

$$\frac{\partial L}{\partial w^3} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial w^3} = \underline{2(y - t)} \cdot h_2$$

計算結果を再利用

$$\frac{\partial L}{\partial w^2} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial u_2} \frac{\partial u_2}{\partial w_2} = \underline{2(y - t)} \cdot w^3 \cdot h_2(1 - h_2) \cdot h_1$$

計算結果を再利用

$$\frac{\partial L}{\partial w^1} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial u_2} \frac{\partial u_2}{\partial h_1} \frac{\partial h_1}{\partial u_1} \frac{\partial u_1}{\partial w_1} = \underline{2(y - t)} \cdot w^3 \cdot h_2(1 - h_2) \cdot w^2 \cdot h_1(1 - h_1) \cdot x$$

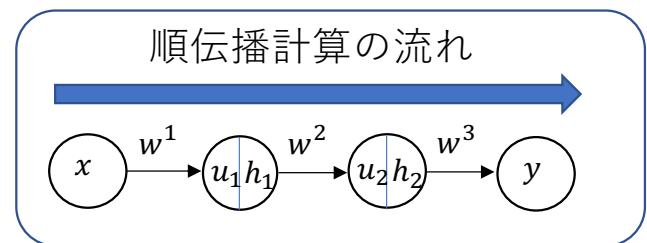
出力層側のパラメータ成分の勾配から計算することで効率よく計算できる。

モデルの訓練：パラメータ更新までの流れ

順伝播計算、逆伝播計算、パラメータ更新までの流れを覚えよう。

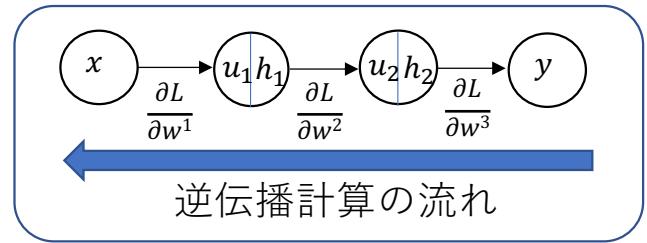
1. 順伝播計算

- NNに入力値 x を与えて予測値 y を計算する。
※中間層のノードの出力値（ h_1 や h_2 など）は逆伝播計算で利用するので覚えておく。



2. 逆伝播計算

- 出力層に近いパラメータ成分の勾配から順番に計算する。



3. パラメータの更新

- 最適化手法を用いてパラメータを更新する。
※パラメータの更新式は最適化手法によって異なる。

パラメータの更新

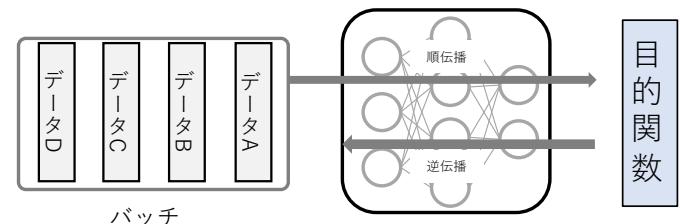
$$w^1 \leftarrow w^1 - \eta \frac{\partial L}{\partial w^1} \quad w^3 \leftarrow w^3 - \eta \frac{\partial L}{\partial w^3}$$
$$w^2 \leftarrow w^2 - \eta \frac{\partial L}{\partial w^2}$$

モデルの訓練：バッチ/ミニバッチ学習

バッチ学習とミニバッチ学習の違いを学ぼう。

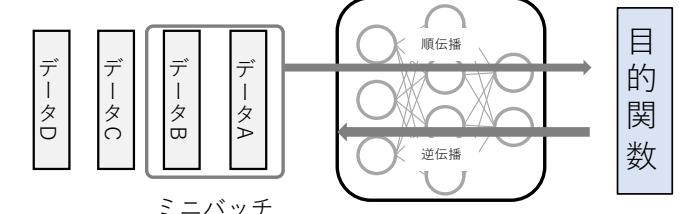
- バッチ学習

- 1回のパラメータ更新時に全ての訓練データセットを使う。



- ミニバッチ学習

- 訓練データセットから**ミニバッチ**と呼ばれる部分集合を作り出し、ミニバッチを用いて1回のパラメータ更新を行う。
- ミニバッチに含まれるデータ点数を**ミニバッチサイズ**を呼ぶ。
- ミニバッチは毎回異なるデータ点の組み合わせを使う。



モデルの訓練：訓練ループの単位

訓練ループに関する単位のイテレーションとエポックを覚えよう。

・イテレーション

- ・パラメータ更新の流れ（順伝播計算、逆伝播計算、パラメータ更新）を何回繰り返したかを表す単位。
- ・イテレーションを何度も繰り返す処理を**訓練ループ**と呼ぶ。

・エポック

- ・訓練データセット全体を何回繰り返して訓練に使ったかを表す単位。

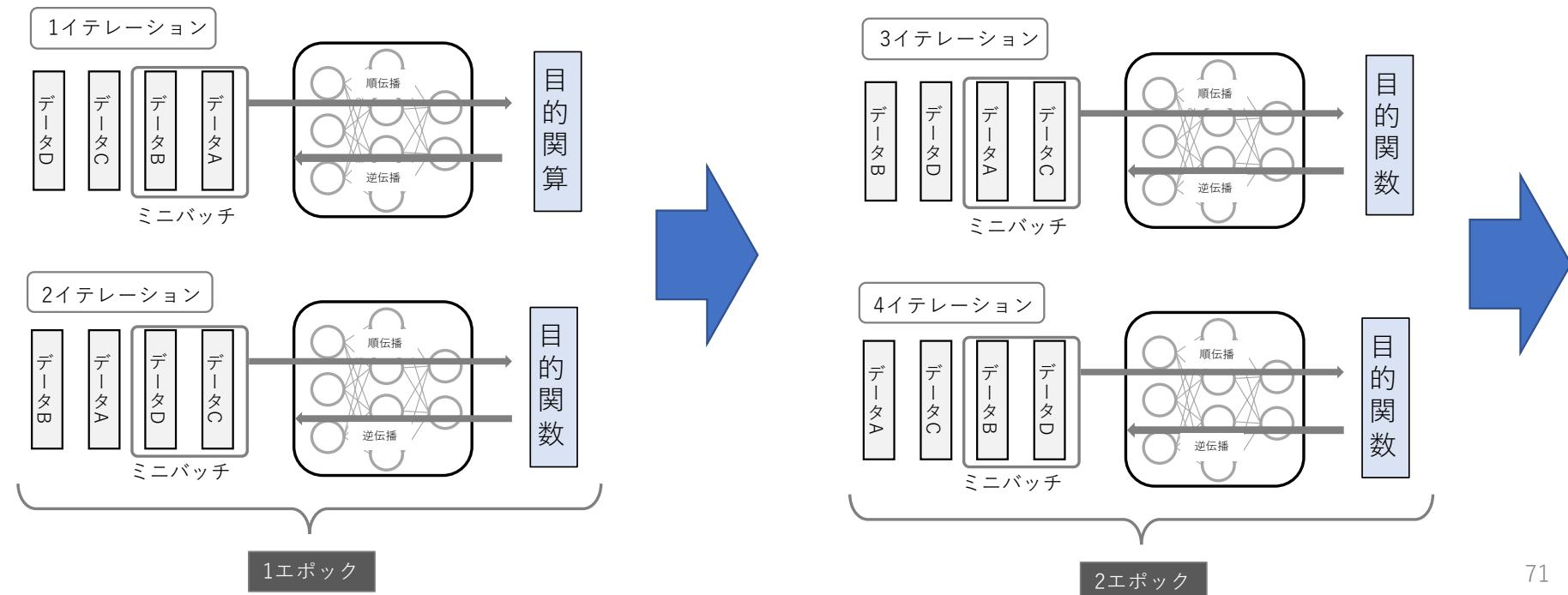
・例

- ・訓練データセットのサイズ100, ミニバッチサイズ10の時、1エポックは10イテレーションである。

モデルの訓練：ミニバッチ学習の例

例を通してミニバッチ学習の流れを覚えよう。

- 訓練データ数4, ミニバッチサイズ2の場合の例
 - 各エポックが始まる前にデータの順番のシャッフルを行う。



モデルの訓練：ハイパーパラメータ

ハイパーパラメータの意味とNNでのハイパーパラメータの例を覚えよう。

- NNの訓練時に人が設定する必要のある値の例
 - ミニバッチサイズ
 - 最大エポック数
 - 学習率
 - ネットワークの層の深さ
 - ネットワークの各層のノード数
- 人が訓練時に設定する値を**ハイパーパラメータ**と呼ぶ。
(モデルのパラメータと区別するため)
- 良いモデルを得るためににはハイパーパラメータの調整も重要である。

モデルの訓練：まとめ

これまで学んだNNの訓練の流れをおさらいをしよう。

1. 訓練の設定

- a. データセットの準備
- b. ネットワークの定義
- c. 目的関数の設定
- d. 最適化手法の選択

2. 訓練ループ

- a. ミニバッチの作成
- b. 順伝播計算
- c. 目的関数による損失の計算
- d. 逆伝播計算
- e. 最適化手法によるパラメータの更新

3. 訓練済みモデルの保存・推論

Chainerの基礎

この章では簡単な課題を通して、Chainerの使い方の基本を学ぼう。

ニューラルネットワークの訓練

1. 訓練の設定

- a. データセットの準備
- b. ネットワークの定義
- c. 目的関数の設定
- d. 最適化手法の選択

2. 訓練ループ

- a. ミニバッチの作成
- b. 順伝播計算
- c. 目的関数による損失の計算
- d. 逆伝播計算
- e. 最適化手法によるパラメータの更新

3. 訓練済みモデルの保存・推論



これらを間違えることなく実装するにはとても大変な作業。

Chainerの役割

1. 訓練の設定

- a. データセットの準備
- b. ネットワークの定義
- c. 目的関数の設定
- d. 最適化手法の選択

Chainerの提供する機能を組み合わせることで、直感的に訓練の設定を行うことができる。

2. 訓練ループ

- a. ミニバッチの作成
- b. 順伝播計算
- c. 目的関数による損失の計算
- d. 逆伝播計算
- e. 最適化手法によるパラメータの更新

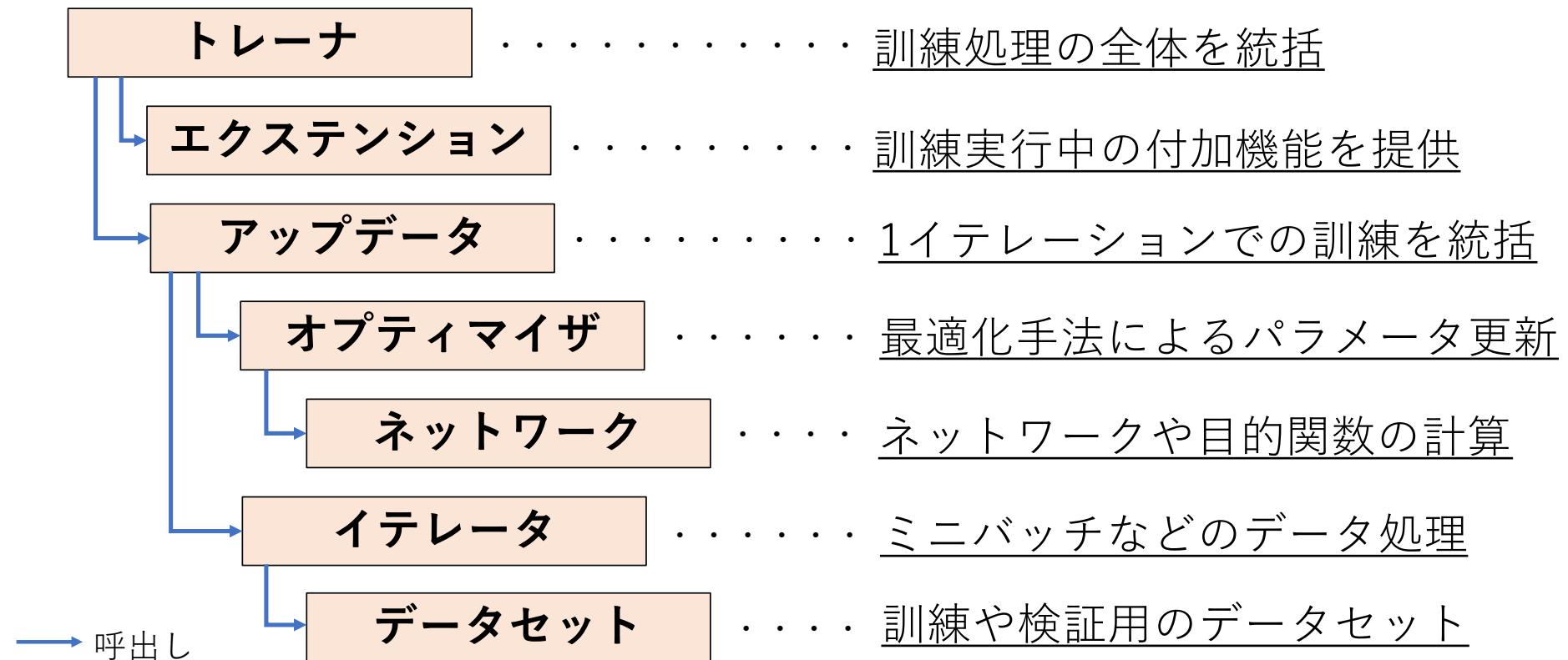
ユーザーが簡潔に記述できるようChainerでは**トレーナ**と呼ばれる仕組みを提供している※。

※トレーナを使わなくてもユーザーが自分で訓練ループを実装することもできる。

3. 訓練済みモデルの保存・推論

トレーナの全体像

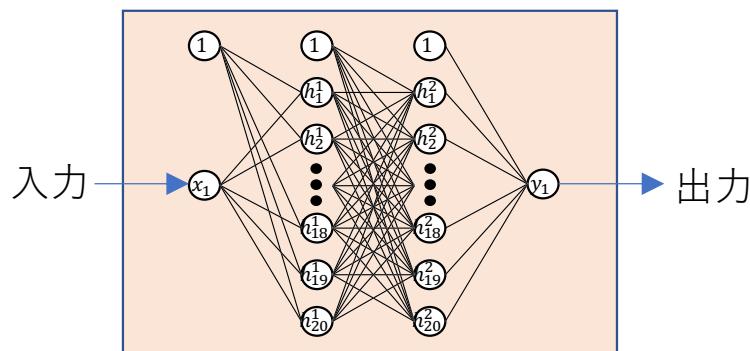
トレーナとトレーナが統括するオブジェクトの全体像を見てみよう。
各詳細はこの後の例題を通して学ぼう。



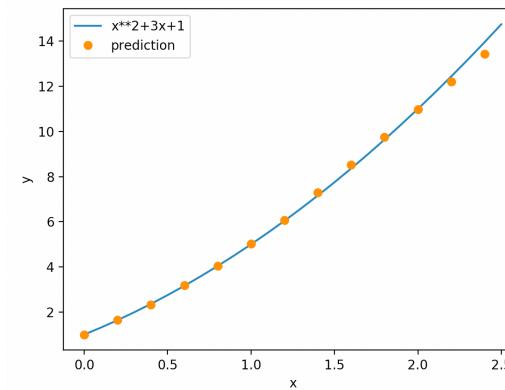
Chainerの使い方を理解するための課題

基本的な課題を通してChainerの使い方に慣れよう

課題：二次関数 x^2+3x+1 を近似するニューラルネットワークを訓練しよう。



x^2+3x+1 を近似する
ネットワーク



この課題の目的

この課題の目的を確認しよう

目的

- 簡単な課題を通してまずはChainerによるプログラミングの流れの身につける。

なぜ簡単な計算式の課題から始めるのか

- x^2+3x+1 を計算することで入力値と目標値の組を簡単に生成できる。
- 1次元の入出力値を扱うことで予測結果を直感的に確かめることができる。

サンプルプログラム

これから学ぶサンプルプログラムの概要を知っておこう

2つのサンプルプログラムを通してChainerの使い方を学ぶ。

- 訓練プログラム：**quadratic_func_trainer.ipynb**
 - 与えられたデータセットと設定を元にネットワークを訓練する。
 - 訓練したモデルを**regression_model/mychain.model**に保存する。
- 推論プログラム：**quadratic_func_evaluator.ipynb**
 - **regression_model/mychain.model**を読み込む。
 - 生成した入力値をネットワークに与え、出力値の予測と評価を行う。

訓練時の設定例

サンプルプログラムの訓練データセットを確認しよう

- 以下の訓練データセットと検証データセットの2種類を利用する。
- 目標値は $x^2 + 3x + 1$ を実際に計算して求める。

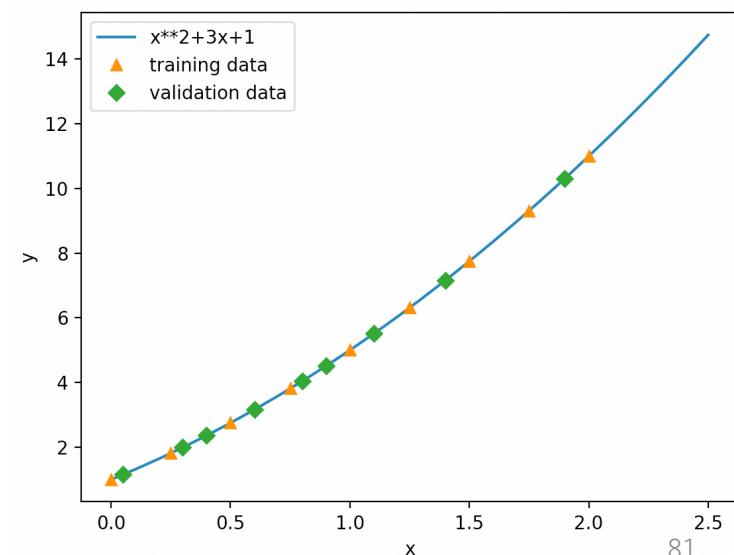
訓練データセット

入力値(x)	目標値(t)
0.00	1.00
0.25	1.81
0.50	2.75
0.75	3.81
1.00	5.00
1.25	6.31
1.50	7.75
1.75	9.31
2.00	11.00

検証データセット

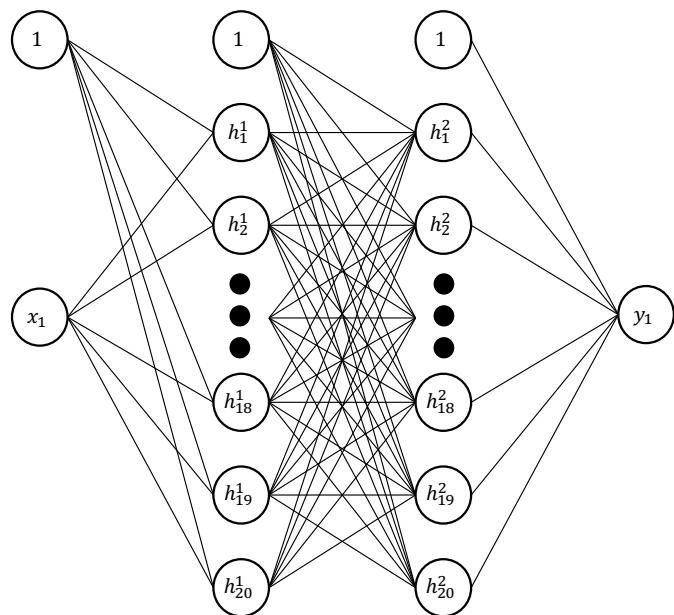
入力値(x)	目標値(t)
0.05	1.15
0.30	1.99
0.40	2.36
0.60	3.16
0.80	4.04
0.90	4.51
1.10	5.51
1.40	7.16
1.90	10.31

訓練/検証データセット
(training/validation data)
のプロット図



訓練時の設定例

サンプルプログラムのネットワークと目的関数を確認しよう



ネットワーク

- 入出力：1次元の実数値
- 中間層の数：2層
- 各中間層のノード数：256個
- 活性化関数：ReLU関数

目的関数

- 最小二乗誤差

$$L = \frac{1}{N} \sum_{n=1}^N (t_n - y_n)^2$$

訓練時の設定例

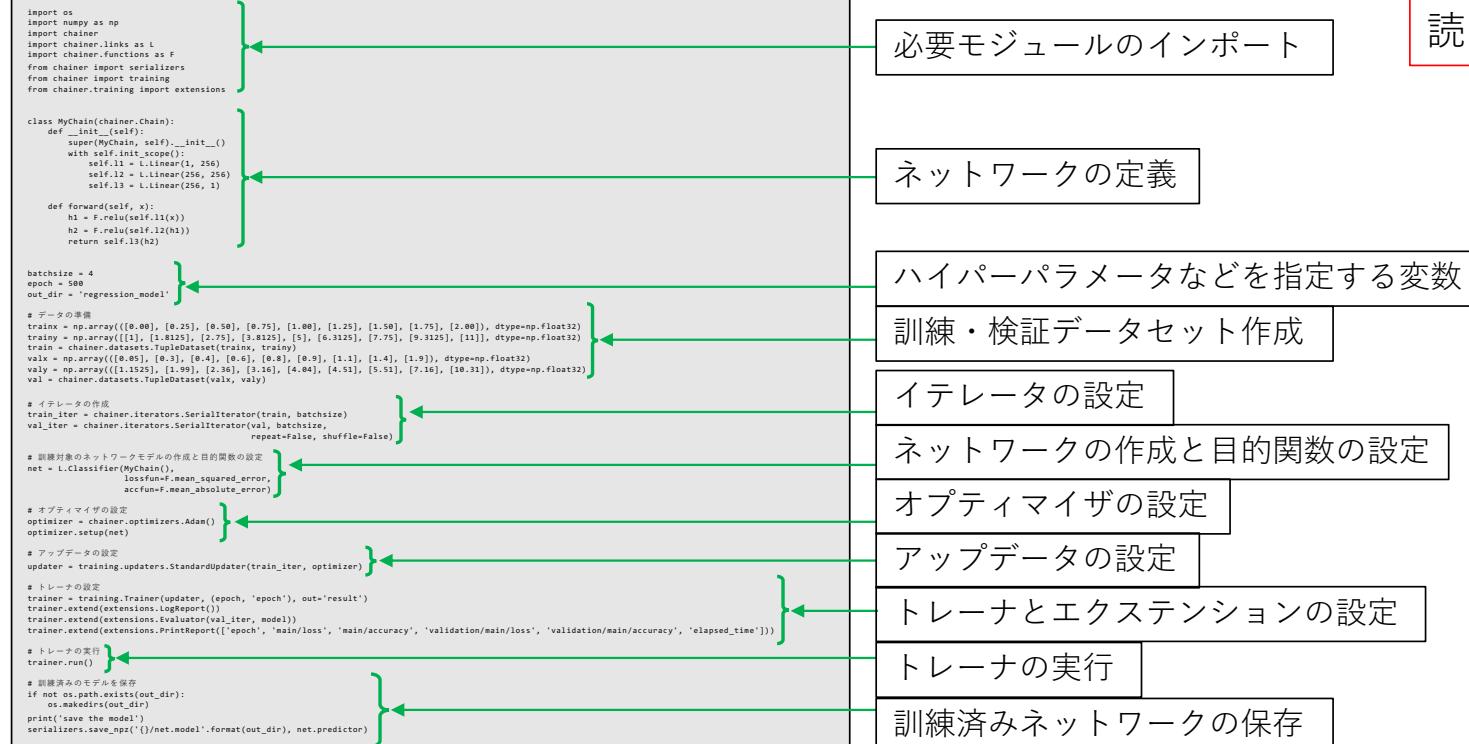
サンプルプログラムで扱うその他の設定を確認しよう

- その他の訓練時の設定
 - 最適化手法：Adam
 - 訓練ループの終了条件：500エポック
 - ミニバッチサイズ：4

プログラムの全体像

訓練プログラムの大枠をつかもう

quadratic_func_trainer.ipynb



※プログラムの詳細は
後に示すので、ここで
読む必要はない

プログラムの全体像

推論プログラムの大枠をつかもう

quadratic_func_evaluator.ipynb

```
import numpy as np
import chainer
import chainer.links as L
import chainer.functions as F

class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(1, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

model_path = 'regression_model/net.model'

# load the model
predictor = MyChain()
chainer.serializers.load_npz(model_path, predictor)

# 入力値の作成
x = np.array(([0.0], [0.2], [0.4], [0.6], [0.8], [1.0], [1.2],
              [1.4], [1.6], [1.8], [2.0], [2.2], [2.4]), dtype=np.float32)
# 推論の実行
result = predictor(x)
# 予測値の取り出し
y = result.data
# 予測値と比較するために目標値を作成
t = x*x + 3*x + 1
# 結果の表示
for i in range(len(x)):
    xi = x[i][0]
    yi = y[i][0]
    ti = t[i][0]
    print("入力値 x: {:.3f}, 予測値 y: {:.3f}, 目標値 t: {:.3f}, 誤差 |y-t|: {:.3f}".format(xi, yi, ti, abs(ti - yi)))
```

必要モジュールのインポート

ネットワークの定義

訓練済みネットワークの読み込み

入力値の生成と予測値の評価

※プログラムの詳細は
後に示すので、ここで
読む必要はない

プログラムを動かしてみよう

詳細に入る前に訓練プログラムを動かして実行結果を見てみよう

quadratic_func_trainer.ipynbの実行例

epoch	main/loss	main/accuracy	validation/main/loss	validation/main/accuracy	elapsed_time
1	33.8489	4.91271	19.3909	3.81443	0.123462
2	13.936	3.27884	9.77942	2.71267	0.204123
3	5.1449	1.94121	3.63521	1.65286	0.290507
4	2.58192	1.3674	0.731242	0.716014	0.422699
5	0.21762	0.348294	0.44894	0.61451	0.550904
(中略)					
496	0.00500993	0.0597903	0.00311359	0.0487931	71.0343
497	0.00279266	0.0403118	0.00232907	0.042887	71.1951
498	0.00511249	0.043298	0.00421055	0.0540388	71.3289
499	0.00479046	0.0395885	0.00278326	0.0459143	71.4386
500	0.00316206	0.0350491	0.0033597	0.0472345	71.6167
save the model					

各列の値の意味

- **epoch:** エポック数
- **main/loss:** 訓練データセットでの目的関数（平均二乗誤差）の値
- **main/accuracy:** 訓練データセットでの評価関数※（平均絶対誤差）の値
- **validation/main/loss:** 検証データセットでの目的関数の値
- **validation/main/accuracy:** 検証データセットでの評価関数※の値

※評価関数については後述する。

訓練済みモデルの保存先

regression_model	
Name	Last Modified
mychain.model	seconds ago

実行後に
regression_model/mychain.modelが生成されていることを確認しよう。

プログラムを動かしてみよう

次は推論プログラムを動かし、訓練したモデルで出力値を予測してみよう

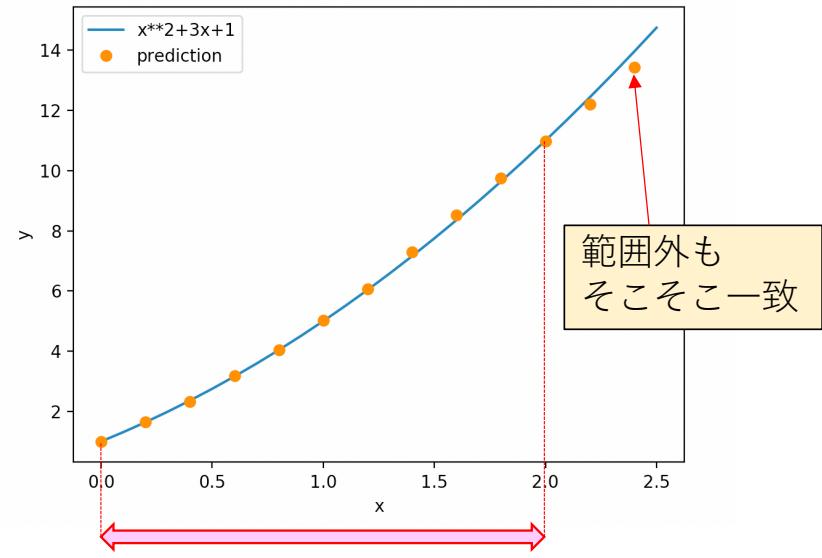
quadratic_func_evaluator.ipynbの実行例

```
入力値 x: 0.000, 予測値 y: 0.997, 目標値 t: 1.000, 誤差 |y-t|: 0.003
入力値 x: 0.200, 予測値 y: 1.650, 目標値 t: 1.640, 誤差 |y-t|: 0.010
入力値 x: 0.400, 予測値 y: 2.329, 目標値 t: 2.360, 誤差 |y-t|: 0.031
入力値 x: 0.600, 予測値 y: 3.181, 目標値 t: 3.160, 誤差 |y-t|: 0.021
入力値 x: 0.800, 予測値 y: 4.052, 目標値 t: 4.040, 誤差 |y-t|: 0.012
入力値 x: 1.000, 予測値 y: 5.020, 目標値 t: 5.000, 誤差 |y-t|: 0.020
入力値 x: 1.200, 予測値 y: 6.072, 目標値 t: 6.040, 誤差 |y-t|: 0.032
入力値 x: 1.400, 予測値 y: 7.287, 目標値 t: 7.160, 誤差 |y-t|: 0.127
入力値 x: 1.600, 予測値 y: 8.516, 目標値 t: 8.360, 誤差 |y-t|: 0.156
入力値 x: 1.800, 予測値 y: 9.744, 目標値 t: 9.640, 誤差 |y-t|: 0.104
入力値 x: 2.000, 予測値 y: 10.972, 目標値 t: 11.000, 誤差 |y-t|: 0.028
入力値 x: 2.200, 予測値 y: 12.201, 目標値 t: 12.440, 誤差 |y-t|: 0.239
入力値 x: 2.400, 予測値 y: 13.429, 目標値 t: 13.960, 誤差 |y-t|: 0.531
```

0.0から2.4の範囲で0.2刻みの入力値を生成する。

予測値と目標値は完全に一致しないが、
近い値となっていることが確認できる。

真の関数と訓練済みネットワークの
予測値のプロット図



訓練用サンプルプログラムの解説

訓練プログラムの詳細を学んでいこう

quadratic_func_trainer.ipynb

```
import os
import numpy as np
import chainer
import chainer.links as L
import chainer.functions as F
from chainer import serializers
from chainer import training
from chainer.training import extensions

class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(1, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

batchsize = 4
epoch = 500
out_dir = "regression_model"

# データの準備
trainx = np.array(([0.00], [0.25], [0.50], [0.75], [1.00], [1.25], [1.50], [1.75], [2.00]), dtype=np.float32)
trainy = np.array([1], [1.8125], [2.75], [3.8125], [5], [6.3125], [7.75], [9.3125], [11]), dtype=np.float32)
train = chainer.datasets.TupleDataset(trainx, trainy)
valx = np.array(([0.05], [0.3], [0.4], [0.6], [0.8], [0.9], [1.1], [1.4], [1.9]), dtype=np.float32)
valy = np.array([1.1525], [1.99], [2.36], [3.16], [4.04], [4.51], [5.51], [7.16], [10.31]), dtype=np.float32)
val = chainer.datasets.TupleDataset(valx, valy)

# イテレータの作成
train_iter = chainer iterators.SerialIterator(train, batchsize,
                                              repeat=False, shuffle=False)
val_iter = chainer iterators.SerialIterator(val, batchsize,
                                             repeat=False, shuffle=False)

# 訓練対象のネットワークの作成と目的関数の設定
net = L.Classifier(MyChain(),
                    lossfun=F.mean_squared_error,
                    accfun=F.mean_absolute_error)
optimizer = chainer optimizers.Adam()
optimizer.setup(net)

# アップデータの設定
updater = training.updaters.StandardUpdater(train_iter, optimizer)

# トレーナーの設定
trainer = training.Trainer(updater, (epoch, 'epoch'), out='result')
trainer.extend(extensions.LogReport())
trainer.extend(extensions.Evaluator(val_iter, model))
trainer.extend(extensions.PrintReport(['epoch', 'main/loss', 'main/accuracy', 'validation/main/loss', 'validation/main/accuracy', 'elapsed_time']))

# トレーナーの実行
trainer.run()

# 訓練済みモデルを保存
if not os.path.exists(out_dir):
    os.makedirs(out_dir)
print("save the model")
serializers.save_npz('{}/net.model'.format(out_dir), net.predictor)
```

The diagram illustrates the flow of the code in quadratic_func_trainer.ipynb. It starts with the '必要モジュールのインポート' (Import of necessary modules) block, which covers the first few lines of the code. This is followed by the 'ネットワークの定義' (Definition of network) block, which contains the class definition for MyChain. Next is the 'ハイパーパラメータなどを指定する変数' (Specify hyperparameters and other variables) block, containing variable assignments like batchsize, epoch, and out_dir. The '訓練・検証データセット作成' (Create training and validation dataset) block follows, showing the creation of TupleDataset objects for train and val. The 'イテレータの設定' (Set iterator) block is shown next, defining SerialIterators for both datasets. The 'ネットワークの作成と目的関数の設定' (Create network and set objective function) block covers the creation of the classifier and the setup of Adam optimizer. The 'オプティマイザの設定' (Set optimizer) block follows. The 'アップデータの設定' (Set updaters) block is shown, defining StandardUpdaters. The 'トレーナーとエクステンションの設定' (Set trainer and extensions) block covers the configuration of the Trainer object, including its updater, epoch count, and output directory. Finally, the 'トレーナーの実行' (Run trainer) block shows the call to trainer.run(). The '訓練済みネットワークの保存' (Save trained network) block at the bottom covers the final line of code where the trained network is saved as a NPZ file.

データセットの準備

データセット作成のためのTupleDatasetの使い方を覚えよう

- **chainer.datasets.TupleDataset** の使い方

- 入力値と目標値の 2 つの配列から、それらのタプル（組）を作成する。

- 使い方の例

```
x = np.array(([0.00], [0.25], [0.50]), dtype=np.float32)
t = np.array(([1.00], [1.81], [2.75]), dtype=np.float32)
tuple_dataset = chainer.datasets.TupleDataset(x, t)

print(tuple_dataset[1]) ← 2番目のデータを表示する。
print(len(tuple_dataset)) ← データセットの数を表示する。
```

実行結果

```
(array([0.25], dtype=float32), array([1.81], dtype=float32))
3
```

入力値(x)	目標値(t)
0.00	1.00
0.25	1.81
0.50	2.75

↓ ↓

入力値と目標値の組
(tuple_dataset)

(0.00, 1.00)
(0.25, 1.81)
(0.50, 2.75)

データセットの準備

Chainerでのデータセット作成のコードを見てみよう

quadratic_func_trainer.ipynb

```
# 訓練データセットtrain_datasetの作成
trainx = np.array(([0.00], [0.25], [0.50],
                  [0.75], [1.00], [1.25],
                  [1.50], [1.75], [2.00]), dtype=np.float32)
traint = np.array([[1], [1.81], [2.75],
                  [3.81], [5], [6.31],
                  [7.75], [9.31], [11.00]], dtype=np.float32) ←
train_dataset = chainer.datasets.TupleDataset(trainx, traint)

# 検証データセットval_datasetの作成
valx = np.array(([0.00], [0.25], [0.50],
                  [0.75], [1.00], [1.25],
                  [1.50], [1.75], [2.00]), dtype=np.float32)
valt = np.array([[1], [1.81], [2.75],
                  [3.81], [5], [6.31],
                  [7.75], [9.31], [11.00]], dtype=np.float32)
val_dataset = chainer.datasets.TupleDataset(valx, valt)
```

※この課題では目標値を以下のようにして生成しても良い。
traint = trainx**2 + 3*trainx + 1

イテレータの設定

イテレータの役割を覚えよう

- イテレータの主な役割
 - データセットからミニバッチを切り出す。
 - エポック毎でデータの順番をシャッフルする。
- **chainer.iterators.SerialIterator**
 - 引数
 - `dataset` : イテレータに割り当てるデータセット
 - `batch_size` : ミニバッチサイズ
 - `repeat` : `True`を指定するとデータを繰り返し読み出す。
 - `shuffle` : `True`を指定するとエポック毎にデータをシャッフルする。

通常、訓練用イテレータは`repeat/shuffle`を`True`、検証用イテレータは`False`とする。

イテレータの設定

作成したデータセットをイテレータに設定する方法を覚えよう

quadratic_func_trainer.ipynb

```
batchsize = 4 ← ミニバッチサイズを入れる変数。ここではミニバッチサイズ4とする。
```

```
# 訓練用データセットのイテレータtrain_iterを作成
train_iter = chainer.iterators.SerialIterator(train_dataset, batchsize,
                                              repeat=True, shuffle=True)
```

```
# 検証用データセットのイテレータval_iterを作成
val_iter = chainer.iterators.SerialIterator(val_dataset, batchsize,
                                             repeat=False, shuffle=False)
```

ネットワークの設定

Chainerでネットワークを定義するためには
リンクとファンクションの役割を学ぼう

リンク

パラメータを持つ関数

例：全結合型NNの線形変換の計算

モジュールのインポート方法

```
import chainer.links as L
```

ファンクション

パラメータを持たない関数

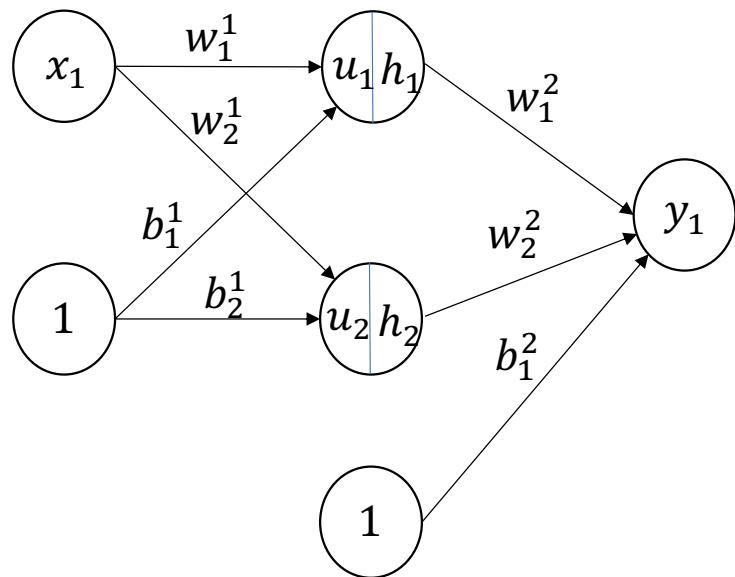
例：シグモイドやReLUなどの活性化関数の計算

モジュールのインポート方法

```
import chainer.functions as F
```

ネットワークの設定

ニューラルネットワークでの変換の過程を思い出そう



$$\begin{aligned} u_1 &= w_1^1 x_1 + b_1^1 \\ u_2 &= w_2^1 x_1 + b_2^1 \end{aligned}$$

$$\begin{aligned} h_1 &= a(u_1) \\ h_2 &= a(u_2) \end{aligned}$$

$$y_1 = w_1^2 h_1 + w_2^2 h_2 + b_1^2]$$

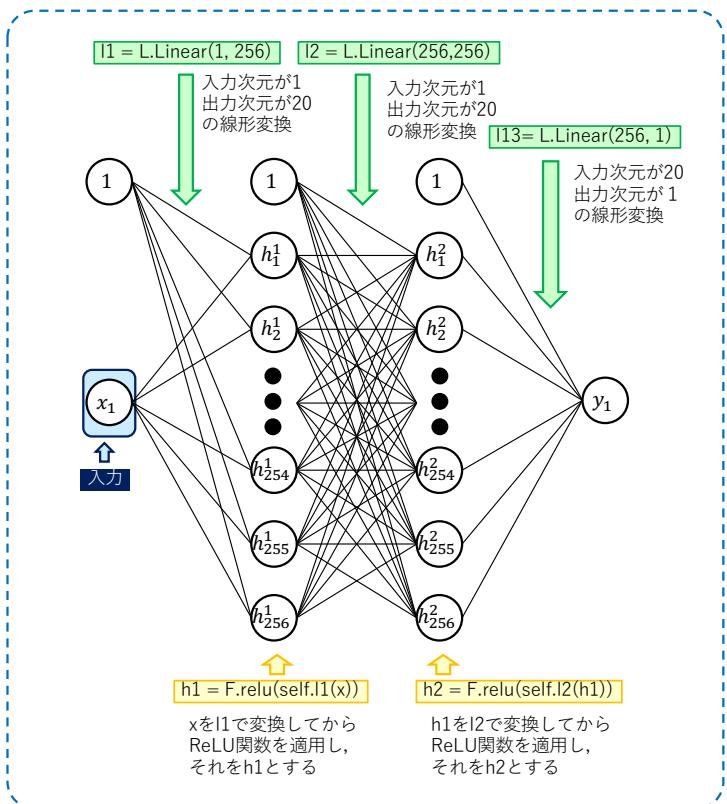
線形変換
↓
非線形変換
↓
線形変換

全結合型ニューラルネットでは

線形変換 → パラメータを持つ関数
非線形変換 → パラメータを持たない関数

ネットワークの設定

ネットワーク定義の方法を学ぼう



定義するネットワーク

quadratic_func_trainer.ipynb

```
class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(1, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

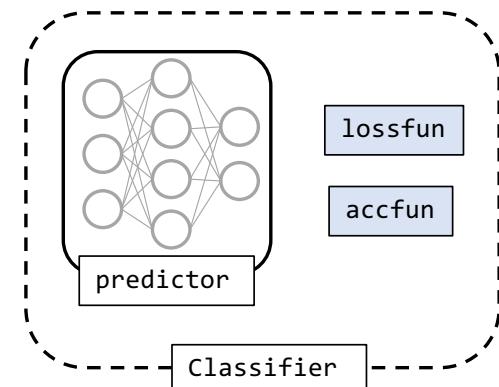
このスコープ中の
リンクの持つパラ
メータは最適化の
対象となる。

リンクとファンクション
を組み合わせて順伝播計
算を定義する。

目的関数の設定

Classifierによる目的関数の設定方法を学ぼう

- **chainer.links.Classifier**の使い方
 - 定義したネットワークをラップして、目的関数と評価関数※を付与する。
 - 引数
 - **predictor** : ラップされるネットワーク
 - **lossfun** : 目的関数
 - **accfun** : 評価関数



※ 目的関数と評価関数の違い

目的関数は損失を計算することによってネットワークのパラメータ更新に寄与するが、評価関数は評価値を計算するのみである。

目的関数の設定

目的関数を設定例を見てみよう

- ここでは定義済のネットワークMyChain()をClassifierへ渡し、平均二乗誤差を目的関数、平均絶対誤差を評価関数として付加する。

quadratic_func_trainer.ipynb

```
net = L.Classifier(MyChain(),  
                    lossfun=F.mean_squared_error,  
                    accfun=F.mean_absolute_error)
```

MyChain()を作成して渡す。

目的関数(lossfun)として平均二乗誤差(F.mean_squared_error)を設定。

評価関数(accfun)として平均絶対誤差(F.mean_absolute_error)を設定。

オプティマイザの設定

オプティマイザの役割と作成方法を学ぼう

- オプティマイザの役割
 - 選択した最適化手法を用いて、モデルのパラメータを更新を行う。
- `chainer.optimizers.Adam`の使用例

`quadratic_func_trainer.ipynb`

```
# オプティマイザの設定  
optimizer = chainer.optimizers.Adam() ← Adam※を選択してオプティマイザを作成。  
optimizer.setup(net)
```

最適化対象のネットワークを持たせる。

※ サンプルコードの`Adam()`のように引数を省略する場合は、その最適化手法のデフォルトのハイパーパラメータが設定される。学習率などを変更したい場合はオプティマイザ作成時に引数として与えることができる。

アップデータの設定

アップデータの役割と設定方法を学ぼう

- アップデータの役割
 - イテレータとオプティマイザを統括して、以下の計算を行う。
 1. 訓練データセットからミニバッチを取り出す
 2. 順伝播、損失、逆伝播を順番に計算
 3. オプティマイザを使ってパラメータを更新
- **chainer.training.updaters.StandardUpdater**の設定例

quadratic_func_trainer.ipynb

```
# アップデータの作成  
updater = training.updaters.StandardUpdater(train_iter, optimizer)
```

↑
↑
設定したイテレータとオプティマイザを渡す

トレーナの設定

トレーナの役割と設定方法を学ぼう

- トレーナの役割
 - アップデータを受け取り、訓練全体の管理を行う。
 - トレーナの作成時に訓練の終了タイミングを指定する。
- `chainer.training.Trainer` の設定例

`quadratic_func_trainer.ipynb`

```
epoch = 500
```

訓練で実行するエポック数。

```
# トレーナの設定
```

```
trainer = training.Trainer(updater, (epoch, 'epoch'), out='result')
```

アップデータを登録

(整数、単位) のタプルで、
訓練終了の条件を指定する。
ここではエポックの単位で
500回繰り返して終了する。

訓練中のログファイル等
を保存するパスを指定。

エクステンションの設定

エクステンションの役割を覚えよう

- エクステンションの役割
 - トレーナの実行中にユーザーにとって便利な付加機能を提供する。
- サンプルコードで利用しているエクステンション
 - **LogReport**: 訓練中の損失や評価関数の値など集計を行う。
 - **Evaluator**: 検証用のイテレータとネットワークを渡し、
訓練中に検証によるモデルの評価を実行する。
 - **PrintReport**: **LogReport**で集計しているログを標準出力へ出力する。
出力する値をリストで渡して指定する。

quadratic_func_trainer.ipynb

```
trainer.extend(extensions.LogReport())
trainer.extend(extensions.Evaluator(val_iter, net))
trainer.extend(extensions.PrintReport(['epoch', 'main/loss',
    'main/accuracy', 'validation/main/loss', 'validation/main/accuracy',
    'elapsed_time']))
```

トレーナの実行

トレーナの実行方法を覚えよう

- トレーナの設定からトレーナ実行までの流れ

quadratic_func_trainer.ipynb

```
# トレーナの設定
trainer = training.Trainer(updater, (epoch, 'epoch'), out='result')

# エクステンションの設定
trainer.extend(extensions.LogReport())
trainer.extend(extensions.Evaluator(val_iter, net))
trainer.extend(extensions.PrintReport(['epoch', 'main/loss',
'main/accuracy', 'validation/main/loss', 'validation/main/accuracy',
'elapsed_time']))

# トレーナの実行
trainer.run() ← トレーナの設定時に指定した終了タイミングまで
                  訓練ループを繰り返し実行する。
```

モデルの保存

最後にモデルの保存方法を覚えよう

- 訓練済みモデルの保存には`chainer.serializers.save_npz`を利用する。

`quadratic_func_trainer.ipynb`

```
out_dir = 'regression_model'  
# 訓練済みのモデルを保存  
if not os.path.exists(out_dir):  
    os.makedirs(out_dir)  
print('save the model')  
serializers.save_npz('{}/mychain.model'.format(out_dir), net.predictor)
```

モデルの保存先ディレクトリパスを変数に設定する。

モデルの保存先ディレクトリが存在するか確認し、なければ作成する。

↑
保存先ディレクトリ/`net.model`へ保存。

↑
保存対象の訓練済みモデル※を指定する。

※ここではネットワークの構造は保存しない。
モデルのパラメータのみを保存している（後述）。

推論プログラムの解説

次は推論プログラムの詳細を学んでいこう

quadratic_func_evaluator.ipynb

```
import numpy as np
import chainer
import chainer.links as L
import chainer.functions as F
```

必要モジュールのインポート

```
class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(1, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

ネットワークの定義

```
model_path = 'regression_model/net.model'
# load the model
predictor = MyChain()
chainer.serializers.load_npz(model_path, predictor)
```

訓練済みネットワークの読み込み

```
# 入力値の生成
x = np.array(([0.0], [0.2], [0.4], [0.6], [0.8], [1.0], [1.2],
              [1.4], [1.6], [1.8], [2.0], [2.2], [2.4]), dtype=np.float32)
# 推論の実行
result = predictor(x)
# 予測値の取り出し
y = result.data
# 予測値と比較するために目標値を作成
t = x*x + 3*x + 1
# 結果の表示
for i in range(len(x)):
    xi = x[i][0]
    yi = y[i][0]
    ti = t[i][0]
    print("入力値 x: {0:6.3f}, 予測値 y: {1:6.3f}, 目標値 t: {2:6.3f}, 誤差 |y-t|: {3:6.3f}".format(xi, yi, ti, abs(ti - yi)))
```

入力値の生成と予測値の評価

訓練済みモデルの読み込み

ネットワークモデルの読み込み方法を学ぼう

- 保存した訓練済みモデルの持っている情報
 - 訓練済みモデルのパラメータのみ。
 - ネットワークの構造そのものは持っていないことに注意。
- 訓練済みモデルの読み込み手順
 1. 訓練時と同じネットワーク構造を作成する。
 2. 訓練済みモデルを読み込み、作成したネットワークにモデルのパラメータをコピーする。

訓練済みモデルの読み込み

訓練したモデルの読み込み方法を学ぼう

- 訓練済みモデルの読み込みには`chainer.serializers.load_npz`を利用する。

`quadratic_func_evaluator.ipynb`

```
model_path = 'regression_model/mychain.model' ← 読み込み対象のモデルのパス。
```

```
# load the model  
predictor = MyChain() ← 訓練時と同じネットワークを作成する。
```

```
chainer.serializers.load_npz(model_path, predictor) ← ネットワークに訓練済みモデルのパラメータをコピーする。
```

訓練済みモデルによる推論

サンプルプログラムを通して推論の方法を学ぼう

- サンプルプログラムでの推論の例

quadratic_func_evaluator.ipynb

```
# 入力値の作成
x = np.array(([0.0], [0.2], [0.4], [0.6], [0.8], [1.0], [1.2], [1.4],
[1.6], [1.8], [2.0], [2.2], [2.4]), dtype=np.float32)
# 推論の実行
result = predictor(x) ← ミニバッチでの入力を基本としているので、複数の入力値をそのまま
# 予測値の取り出し
y = result.data ← dataにNumPyの各入力値に対応する予測値が入っている。
# 予測値と比較するために目標値を作成
t = x*x + 3*x + 1
# 結果の表示
for i in range(len(x)):
    xi = x[i][0]
    yi = y[i][0] ← 結果の表示のためにスカラー値に変換
    ti = t[i][0]
    print("入力値 x: {0:6.3f}, 予測値 y: {1:6.3f}, 目標値 t: {2:6.3f}, 誤
差 |y-t|: {3:6.3f}".format(xi, yi, ti, abs(ti - yi)))
```

練習問題

ここで学んだ課題を応用した練習問題に取り組んでみよう

- **練習問題 1**

- 右のようにMyChainの活性化関数を取り除いた場合の予測値を確認してみよう。
- 結果をグラフにプロットするなどして、なぜそのような結果となったのか考察してみよう。

- **練習問題 2**

- 二次関数よりも複雑な関数のフィッティングをしてみよう。
- うまく予測ができない場合はハイパーパラメータを変更してみよう。

```
class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(1, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = self.l1(x)
        h2 = self.l2(h1)
        return self.l3(h2)
```

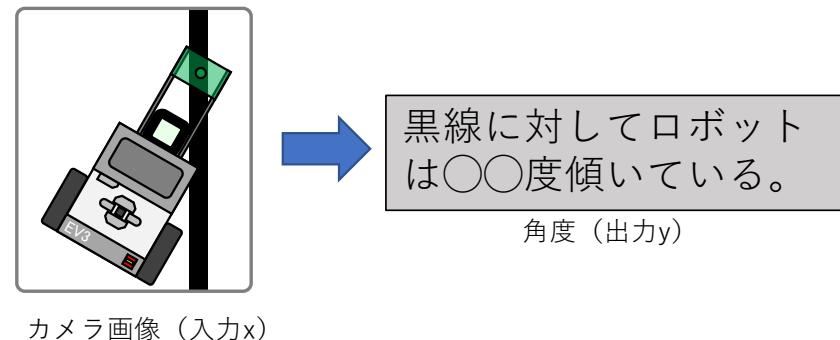
Chainerによるロボット制御

Chainerを使ったロボット制御の課題

これから取り組む2つの課題の概要を知ろう

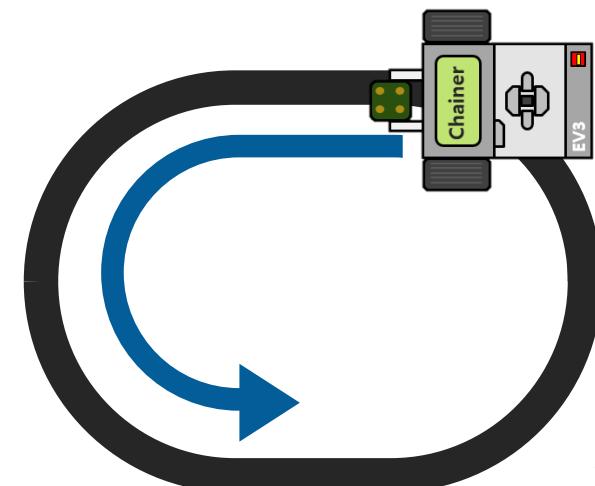
課題1. カメラ画像を使った ロボットの向きの推定

カメラ画像とそのときのEV3の向きの組からなるデータセットを作成し、画像からEV3の向き（角度）を推定するモデルを作る。



課題2. 学習ベースのライント レースプログラムの作成

ルールベースのライントレースを動かして、カメラ画像とその時の制御値の組のデータセットを作り、カメラでライントレース制御ができるモデルを作る。



課題 1：カメラ画像を使った ロボットの向きの推定

課題 1 の実行例

課題 1 で作成するプログラムの実行例を見てみよう

- ・ カメラを黒線に向けて傾けて撮影し、ロボットの向き（カメラ画像に映っている黒線の角度）の推定を行うモデルを作ろう。
- ・ 推定する角度は-30度から30度の連続値とする。

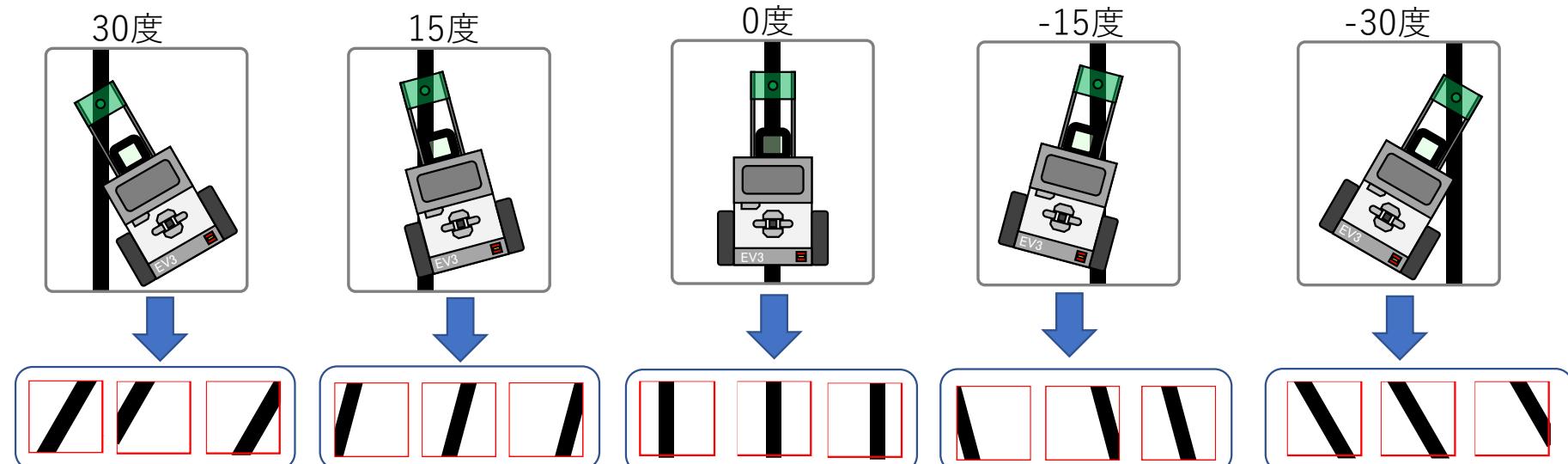


訓練データセットの作成方針

課題1で作成するデータセットの作成方針について学ぼう

訓練データセットの方針

- 5種類の目標値（-30度、-15度、0度、15度、30度）のデータセットを作成する。
- ロボットを左右に平行移動させながら撮影することでデータのバリエーションを増やそう。



訓練データセットの目標値にはない角度（20度、-5度など）も含めて予測ができるか実験により確かめる。

課題 1 で作成するプログラム

課題 1 で作成する 3 つのプログラムの概要を学ぼう

1. 訓練データセット作成プログラム

(angle_prediction_logger.ipynb)

- 黒線をカメラで撮影して、カメラ画像と角度の組のデータセットを作成する。

2. 訓練プログラム

(angle_prediction_trainer.ipynb)

- 作成したデータセットを用いて、カメラ画像から角度を予測するモデルを作成する。

3. 推論プログラム

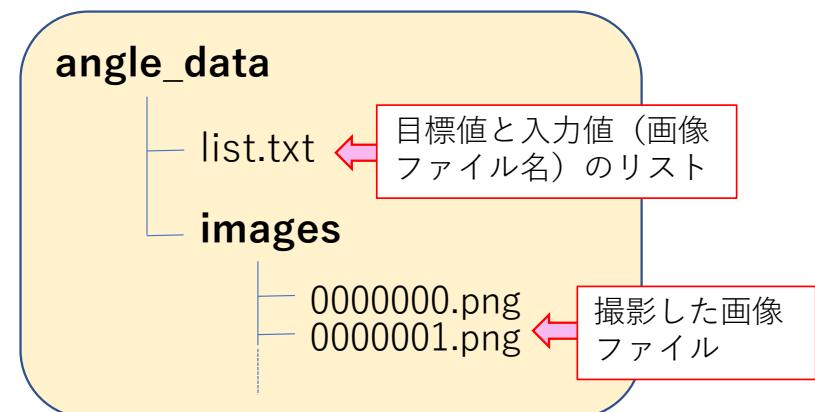
(angle_prediction_predictor.ipynb)

- カメラで黒線を撮影して、その角度をLCDに表示する。

訓練データセット作成プログラム

訓練データセット作成プログラムの実装方針を確認しよう

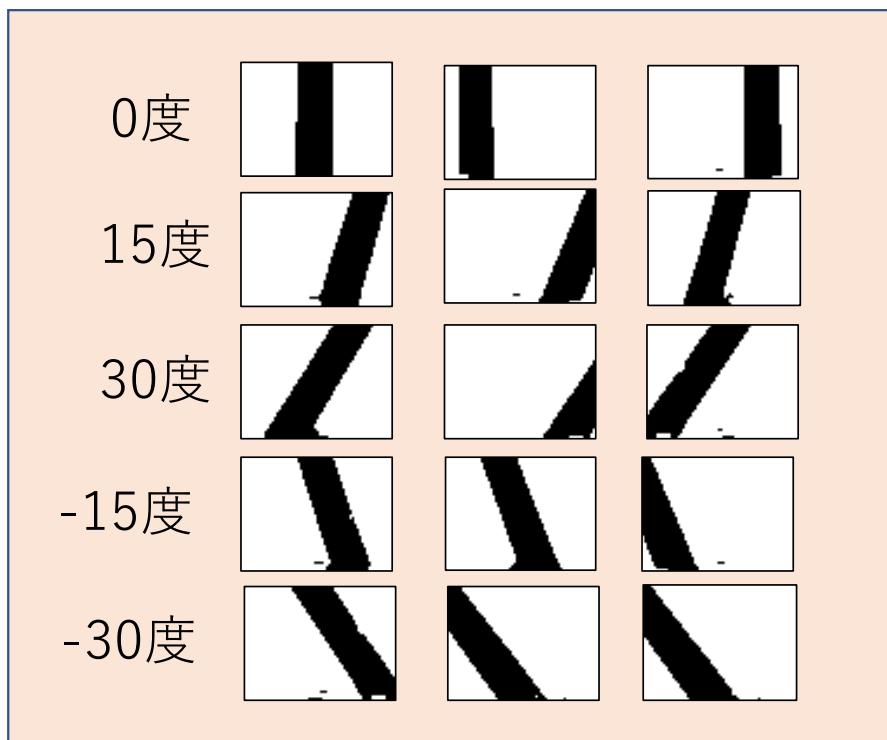
- 画像処理の基礎で学んだプログラム
「labeled_image_logger.ipynb」をベースにしよう。
- 次のボタンを押したらカメラ画像と対応する角度の組を保存する
ようにプログラムに修正しよう。
 - 「ENTER」 → 0度
 - 「UP」 → 15度
 - 「LEFT」 → 30度
 - 「DOWN」 → -15度
 - 「RIGHT」 → -30度
- loggerライブラリを使い、右のような
データセットを作成しよう。



訓練データセット作成プログラム

実際に作成するデータセットの例をみてみよう

撮影した実際の画像の例



list.txtの例

```
0000000.png 0  
0000001.png 0  
0000002.png 0  
0000003.png 0  
0000004.png 0  
0000005.png 0  
0000006.png 0  
0000007.png 0  
0000008.png 0  
0000009.png 0  
0000010.png 0  
0000011.png 0  
⋮
```

「目標値 画像ファイル名」のリスト

訓練データセット作成プログラム

データセット作成に関するその他の設定例を確認しよう

- カメラ画像の設定について
 - 解像度：横64画素 × 縦48画素
 - 色：白黒の2値画像

```
vs = VideoStream((64, 48), 10, colormode='binary').start()
```

- データセットについて
 - 訓練用と検証用の2つのデータセットを作成する。
 - データセットのサイズの目安
 - 訓練データセット：各角度の目標値毎に30から50個
 - 検証データセット：各角度の目標値毎に5個程度

サンプルプログラム

データセット作成プログラムのサンプルを見てみよう

```
import time
from IPython.display import Image, display
from lib.vstream import VideoStream
from lib.logger import Logger
from lib.ev3 import EV3

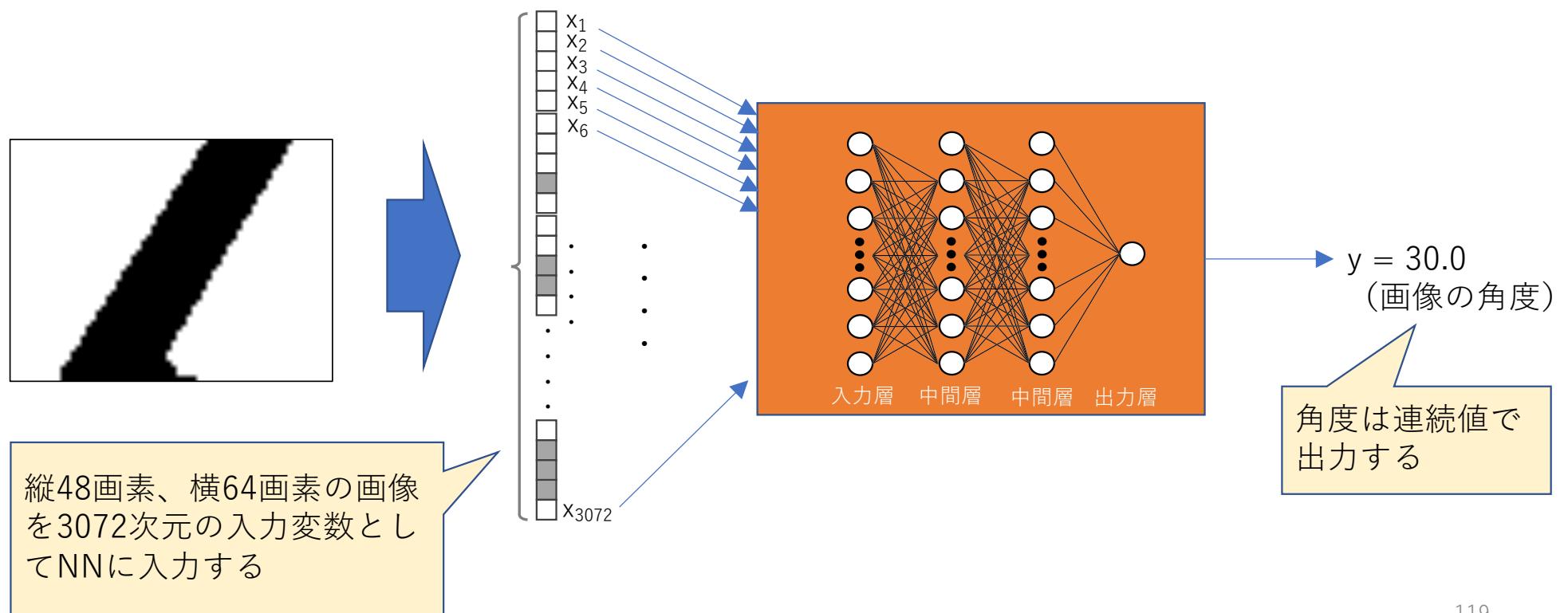
# Define the connection ports of the sensors and motors.
touch_port = EV3.PORT_2

ev3 = EV3()
ev3.sensor_config(touch_port, EV3.TOUCH_SENSOR)
lg = Logger(top_dir='angle_data')
vs = VideoStream((64, 48), 10, colormode='binary').start()
# Counter for each label
enter = up = left = down = right = 0
print('Ready')
while True:
    ev3.lcd_draw_string('enter: 0, {:02}'.format(enter), 0)
    ev3.lcd_draw_string('up : 15, {:02}'.format(up), 1)
    ev3.lcd_draw_string('left : 30, {:02}'.format(left), 2)
    ev3.lcd_draw_string('down : -15, {:02}'.format(down), 3)
    ev3.lcd_draw_string('right: -30, {:02}'.format(right), 4)
    if ev3.touch_sensor_is_pressed(touch_port):
        break
    if ev3.button_is_pressed(EV3.ENTER_BUTTON):
        label = 0
        enter += 1
    elif ev3.button_is_pressed(EV3.UP_BUTTON):
        label = 15
        up += 1
    elif ev3.button_is_pressed(EV3.LEFT_BUTTON):
        label = 30
        left += 1
    elif ev3.button_is_pressed(EV3.DOWN_BUTTON):
        label = -15
        down += 1
    elif ev3.button_is_pressed(EV3.RIGHT_BUTTON):
        label = -30
        right += 1
    else:
        continue
    image = vs.read()
    lg.write(image, label)
    image.save('out.png')
    display(Image('out.png'))
print('Finish')
ev3.close()
vs.stop()
lg.close()
```

訓練プログラム

作成するモデルを確認しよう

- 作成するモデルの例



訓練プログラム

この課題のネットワーク定義の例を確認しよう

- ネットワーク定義の例
 - 入力次元数 : 3072
 - 出力次元数 : 1
 - 中間層の数 : 2
 - 中間層のノード数 : 256
 - 活性化関数 : ReLU関数

```
class MyChain(chainer.Chain):
    def __init__(self):
        super(MyChain, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(3072, 256)
            self.l2 = L.Linear(256, 256)
            self.l3 = L.Linear(256, 1)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

訓練プログラム

画像のデータセットの読み込み方法を覚えよう

- **chainer.datasets.LabeledImageDataset**を利用した
画像データセットの読み込み方法
 - loggerを用いて作成したデータセットはChainerのAPIを使って直接
読み込むことができる。
 - 使用例

```
import os
from chainer.datasets import LabeledImageDataset  
  
(中略)  
  
train_dir = 'angle_data/20190520-130821'  
raw_train_dataset = LabeledImageDataset(os.path.join(train_dir, 'list.txt'),  
                                         os.path.join(train_dir, 'images'),  
                                         label_dtype=np.float32)
```

import os
from chainer.datasets import LabeledImageDataset

(中略)

train_dir = 'angle_data/20190520-130821' ← 訓練データセットのパス
raw_train_dataset = LabeledImageDataset(os.path.join(train_dir, 'list.txt'), ← list.txtのパスを
 os.path.join(train_dir, 'images'), ← 指定
 label_dtype=np.float32) ← imagesディレクトリへのパスを指定

目標値のデータ型を指定

訓練プログラム

画像データセットの前処理方法を覚えよう

- データセットの前処理の目的
 - 画像データは2次元配列として読み込まれるため、**1次元**の配列に変換する。
 - 画素値は**0**（黒）から**255**（白）までの整数値のため、**0.0**から**1.0**の実数値の範囲に収まるように変換する。
(このような前処理を**正規化**と呼ぶ。詳細は割愛。)
 - 目標値はスカラーとして読み込まれているため、**1次元**の長さ**1**の配列に変換する。

訓練プログラム

画像データセットの前処理方法を覚えよう

- **chainer.datasets.TransformDataset**を利用した
画像データセットの前処理方法

```
from chainer.datasets import TransformDataset
```

(中略)

```
# データの前処理
```

```
def preprocess(in_data):
```

```
    img, label = in_data
```

```
    img = img / 256. ← 画素値を0.0から1.0へ正規化
```

```
    img = img.reshape(3072) ← 64x48の2次元配列を長さ3072の1次元配列へ変換
```

```
    label = label.reshape(1) ← スカラー値を長さ1の1次元配列へ変換
```

```
    return img, label
```

(中略)

```
# 訓練データセットの前処理の設定
```

```
train_dataset = TransformDataset(raw_train_dataset, preprocess)
```

画素値を0.0から1.0へ正規化

64x48の2次元配列を長さ3072の1次元配列へ変換

スカラー値を長さ1の1次元配列へ変換

LabeledImageDatasetで読み込んだデータ
セットをラップして前処理preprocessを登録。

訓練プログラム

訓練に関するその他の設定例を確認しよう

- ミニバッチサイズ
 - 100
- 最大エポック数
 - 200
- 目的関数
 - 平均二乗誤差
- 評価関数
 - 平均絶対誤差
- 最適化手法
 - Adam
- モデルの保存先
 - `angle_prediction/mychain.model`

これらの設定を参考にして訓練プログラムを作成してみよう。

サンプルプログラム

訓練プログラムの実装例を見てみよう

```
import os
import numpy as np
from PIL import Image
import chainer
from chainer.datasets import LabeledImageDataset
from chainer.datasets import TransformDataset
import chainer.links as L
import chainer.functions as F
from chainer import serializers
from chainer import training
from chainer.training import extensions

# Network definition
class MyChain(chainer.Chain):

    def __init__(self):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(3072, 256) # 3072(64*48) -> 256 units
            self.l2 = L.Linear(256, 256) # 256 units -> 256 units
            self.l3 = L.Linear(256, 1) # 256 units -> 1

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

    # Preprocess data
    def preprocess(self, in_data):
        img, label = in_data
        img = img / 256. # normalization
        img = img.reshape(3072) # 2-dim (64,48) -> 1-dim (3072)
        label = label.reshape(1) # scalar -> 1-dim
        return img, label

input_dir = 'angle_data/20190520-130821'
out_dir = 'angle_model'
batchsize = 100
epoch = 200

# Set up a neural network to train
model = L.Classifier(MLP(),
                     lossfun=F.mean_squared_error,
                     accfun=F.mean_absolute_error)
```

```
# Setup an optimizer
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)

# Load the dataset
dataset = LabeledImageDataset(os.path.join(input_dir, 'list.txt'),
                               os.path.join(input_dir, 'images'),
                               label_dtype=np.float32)

# Set a dataset
train = TransformDataset(dataset, preprocess)

# Set an iterator
train_iter = chainer iterators.SerialIterator(train, batchsize)

# Setup an Updater
updater = training.updaters.StandardUpdater(train_iter, optimizer)

# Setup a Trainer
trainer = training.Trainer(updater, (epoch, 'epoch'), out='result')
trainer.extend(extensions.LogReport())
trainer.extend(extensions.PrintReport(['epoch', 'main/loss', 'main/accuracy', 'elapsed_time']))

# Run the Trainer
trainer.run()

# Save the model, the optimizer and config.
if not os.path.exists(out_dir):
    os.makedirs(out_dir)
print('save the model')
serializers.save_npz('{}/mychain.model'.format(out_dir), model.predictor)
```

推論プログラム

推論プログラムの流れを確認しよう

角度推定の流れ

1. EV3関連の初期化を行う
2. 保存した訓練済みモデルを読み込む
3. VideoStreamの初期化を行う
4. 次のループをタッチセンサーが押されるまで繰り返す。
 - A) カメラから画像を読み込む
 - B) モデルに画像を入力するために、訓練時と同じ前処理を行う。
 - C) モデルに入力値を入れて、予測値（推定角度）を得る。
 - D) 予測値をEV3のLCDに表示

推論プログラム

推論による角度予測のループの実装例を見ていこう

角度予測ループの実装例

```
while True:  
    if ev3.touch_sensor_is_pressed(touch_port): break  
  
    im = vs.read() ← カメラ画像の読み込み  
    im = np.asarray(im, dtype=np.float32) ← NumPy形式へ変換  
  
    x = im / 255. ← 0.0-1.0へ正規化  
    x = x.reshape(1, 3072) ← 入力画像を(ミニバッチサイズ, 画素数)へ変換  
    y = model.predictor(x) ← モデルへ入力、予測値を計算  
    angle = y.data[0, 0] ← 表示のために推定角度をスカラー値として取り出す。  
                           (0バッチ目の0次元目のデータ)  
  
    print("predicted angle = {}".format(angle))  
    ev3.lcd_draw_string('angle={}'.format(angle), 0) ← LCDに推定角度を表示
```

サンプルプログラム

推論プログラムの実装例をみてみよう

```
import time
import numpy as np

import chainer
from chainer import configuration
import chainer.links as L
import chainer.functions as F
from chainer import serializers

from lib.ev3 import EV3
from lib.vstream import VideoStream

touch_port = EV3.PORT_2
lmotor_port = EV3.PORT_B
rmotor_port = EV3.PORT_C

# Network definition
class MyChain(chainer.Chain):

    def __init__(self):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(3072, 256) # 300(20*15) -> 256 units
            self.l2 = L.Linear(256, 256) # 256 units -> 256 units
            self.l3 = L.Linear(256, 1) # 256 units -> 1

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

# Set up a neural network of trained model
model = MyChain()

# Load the model
serializers.load_npz('angle_model/mychain.model', model)

# Run VideoStream by setting image_size and fps
vs = VideoStream(resolution=(64, 48),
                  framerate=10,
                  colormode='binary').start()
```

```
ev3 = EV3()
ev3.sensor_config(touch_port, EV3.TOUCH_SENSOR)

# Enable evaluation mode for faster inference.
with configuration.using_config('train', False), chainer.using_config('enable_backprop', False):
    while True:
        # Break this loop when the touch sensor was pressed.
        if ev3.touch_sensor_is_pressed(touch_port):
            break
        im = vs.read() # Get a current image in PIL format.
        im = np.asarray(im, dtype=np.float32) # Convert to numpy array.
        x = im / 255. # Normalization
        x = x.reshape(1, 3072) # (64, 48) -> (1, 3072)
        y = model.predictor(x) # Predict steer value from x.
        angle = y.data[0, 0]
        print("predicted angle = {}".format(angle))
        ev3.lcd_draw_string('angle={}'.format(angle), 0)

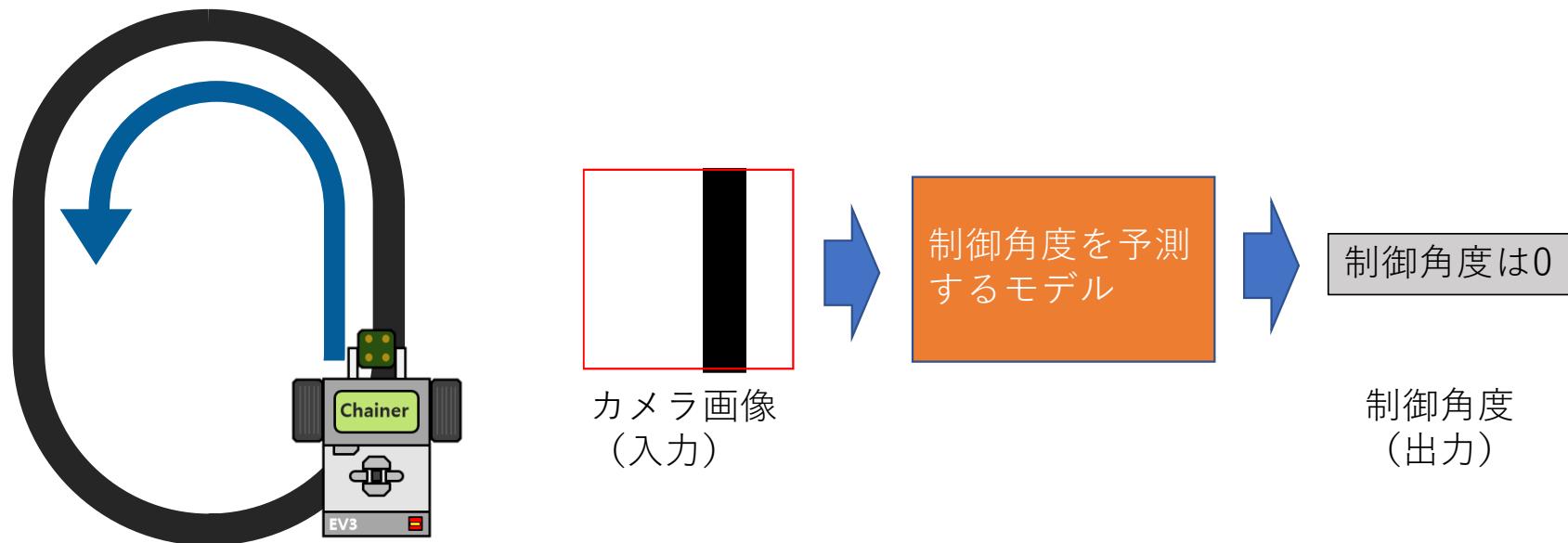
vs.stop()
ev3.close()
```

課題2：学習ベースのライン トレースプログラム

課題 2 の目標

課題 2 で作成するプログラムの目標を確認しよう

- ・ カメラ画像によるライントレースプログラムを開発する。
- ・ 入力値はカメラ画像、目標値は制御角度。制御速度は10で固定。
- ・ 訓練データセットはルールベースのライントレースを動かしながら作成する。→ ルールベース制御をカメラを使って模倣するモデルを作る。



課題 2 で作成するプログラム

課題 2 で作成する 3 つのプログラムの概要を学ぼう

1. 訓練データセット作成プログラム

(ml_linetrace_logger.ipynb)

1. ルールベースのライントレースプログラムを動かしながら、カメラ画像（入力値）と制御角度（目標値）の組を取得する。

2. 訓練プログラム

(ml_linetrace_trainer.ipynb)

- 作成したデータセットを用いて、カメラ画像から制御角度を予測するモデルを訓練する。

3. カメラベースのライントレースプログラム

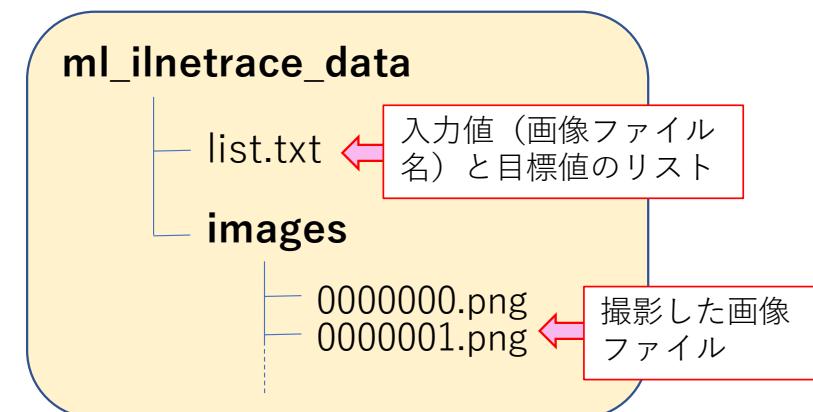
(ml_linetrace_controller.ipynb)

- 訓練したモデルを使って、カメラ画像から制御角度を予測し、実際にモーターを動かしてライン上を走らせる。

訓練データセット作成プログラム

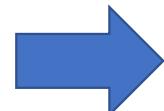
訓練データセット作成プログラムの実装方針を確認しよう

- ルールベース制御のライントレースプログラム
「rule_linetrace_controller.ipynb」をベースに実装しよう。
- 入力値はカメラ画像、目標値は制御角度（整数値）を組とする
データセットを作成する。
- VideoStreamとLoggerを
ルールベース制御のプログラムへ
組み込み、右のようなデータセットを作ろう。

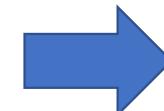


訓練データセット作成プログラム

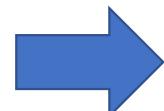
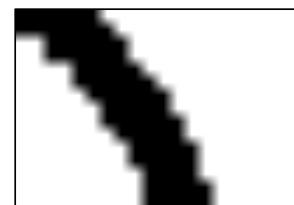
実際に作成するデータセットの例をみてみよう



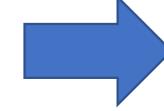
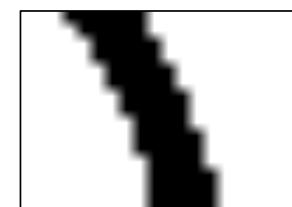
制御角度は0



制御角度は9



制御角度は-49



制御角度は-40

訓練データセット作成プログラム

データセット作成に関するその他の設定例を確認しよう

- カメラ画像の設定について
 - 解像度：横20画素 × 縦15画素
 - 色：白黒の2値画像
- モーターの制御値について
 - 制御角度：-100から100までの整数値として記録
 - 制御速度：常に10で等速とするので記録する必要はない
- データセットについて
 - 訓練用と検証用の2つのデータセットを作成する。
 - データセットのサイズの目安
 - 訓練データセット：約2000から3000組
 - 検証データセット：約300から500組

VideoStreamの設定例

```
vs = VideoStream((20, 15), 10, colormode='binary').start()
```

サンプルプログラム

データセット作成プログラムのサンプルを見てみよう

```
from lib.ev3 import EV3
from lib.vstream import VideoStream
from lib.logger import Logger

# センサーとモーターの通信ポートの定義
touch_port = EV3.PORT_2
color_port = EV3.PORT_3
lmotor_port = EV3.PORT_B
rmotor_port = EV3.PORT_C

ev3 = EV3()
ev3.motor_config(lmotor_port, EV3.LARGE_MOTOR)
ev3.motor_config(rmotor_port, EV3.LARGE_MOTOR)
ev3.sensor_config(touch_port, EV3.TOUCH_SENSOR)
ev3.sensor_config(color_port, EV3.COLOR_SENSOR)

# 白面の反射値のキャリブレーション
print("Press the touch sensor to measure light intensity on WHITE.\n")
ev3.lcd_draw_string('Calibrate white', 0)
while not ev3.touch_sensor_is_pressed(touch_port):
    pass
white = ev3.color_sensor_get_reflect(color_port)
print("WHITE light intensity: {}".format(white))
ev3.lcd_draw_string('White = {}'.format(white), 1)

while ev3.touch_sensor_is_pressed(touch_port):
    pass

# 黒面の反射値のキャリブレーション
print("Press the touch sensor to measure light intensity on BLACK.\n")
ev3.lcd_draw_string('Calibrate black', 0)
while not ev3.touch_sensor_is_pressed(touch_port):
    pass
black = ev3.color_sensor_get_reflect(color_port)
print("BLACK light intensity: {}".format(black))
ev3.lcd_draw_string('Black = {}'.format(black), 2)

vs = VideoStream(resolution=(20,15), framerate=10, colormode='binary').start()
lg = Logger(top_dir='ml_linetrace_data')
```

```
ev3.lcd_draw_string('Push to start.', 0)
while ev3.touch_sensor_is_pressed(touch_port):
    pass

ev3.lcd_draw_string('Go!', 0)

# 制御ループ
midpoint = (white - black) / 2 + black
while True:
    # タッチセンサーを押したら終了
    if ev3.touch_sensor_is_pressed(touch_port):
        break
    # P制御で制御角度を計算
    color = ev3.color_sensor_get_reflect(color_port)
    error = color - midpoint
    steer = 1.6 * error
    # EV3に制御角度を送る
    ev3.motor_steer(lmotor_port, rmotor_port, 10, int(steer))
    # (画像、制御角度) の組をログに書き込む。
    lg.write(vs.read(), int(steer))

# 終了処理
ev3.motor_steer(lmotor_port, rmotor_port, 0, 0)
lg.close()
vs.stop()
ev3.close()
```

訓練プログラム

訓練プログラムの実装方針を確認しよう

- 課題 1 で作成した「angle_prediction_trainer.ipynb」 とほぼ同じ実装を使用しよう。
- データセットサイズが大きいので、最大エポック数を30程度に設定して実行する。
- モデルの保存先は`ml_linetrace_model/mychain.model`とする。

サンプルプログラム

訓練プログラムの実装例を見てみよう

```
import os
import numpy as np
from PIL import Image
import chainer
from chainer.datasets import LabeledImageDataset
from chainer.datasets import TransformDataset
import chainer.links as L
import chainer.functions as F
from chainer import serializers
from chainer import training
from chainer.training import extensions

# ネットワークの定義
class MLP(chainer.Chain):

    def __init__(self):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(300, 256) # 300(20*15) -> 256 units
            self.l2 = L.Linear(256, 256) # 256 units -> 256 units
            self.l3 = L.Linear(256, 1) # 256 units -> 1

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

# データの前処理
def preprocess(in_data):
    img, label = in_data
    img = img / 256. # 正規化
    img = img.reshape(300) # 2-dim (20,15) -> 1-dim (300)
    label = label.reshape(1) # scalar -> 1-dim
    return img, label
```

```
input_dir = 'ml_linetrace_data/20190520-114406'
out_dir = 'ml_linetrace_model'
batchsize = 100
epoch = 100

# ネットワークと目的関数の設定
model = L.Classifier(MLP(),
                      lossfun=F.mean_squared_error,
                      accfun=F.mean_absolute_error)

# オプティマイザの設定
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)

# データセットの読み込み
dataset = LabeledImageDataset(os.path.join(input_dir, 'list.txt'),
                               os.path.join(input_dir, 'images'),
                               label_dtype=np.float32)

# 訓練と検証データセットに分割
threshold = np.int32(len(dataset) * 0.8)
train = TransformDataset(dataset[0:threshold], preprocess)
val = TransformDataset(dataset[threshold:], preprocess)

# Sイテレータの設定
train_iter = chainer.iterators.SerialIterator(train, batchsize)
val_iter = chainer.iterators.SerialIterator(val, batchsize,
                                            repeat=False, shuffle=False)

# アップデータの設定
updater = training.updaters.StandardUpdater(train_iter, optimizer)

# トレーナの設定
trainer = training.Trainer(updater, (epoch, 'epoch'), out='result')
trainer.extend(extensions.LogReport())
trainer.extend(extensions.Evaluator(val_iter, model))
trainer.extend(extensions.PrintReport(['epoch', 'main/loss', 'main/accuracy', 'validation/main/loss',
                                       'validation/main/accuracy', 'elapsed_time']))

# トレーナの実行
trainer.run()

# モデルの保存
if not os.path.exists(out_dir):
    os.makedirs(out_dir)
print('save the model')
serializers.save_npz('{}/mychain.model'.format(out_dir), model.predictor)
```

ライントレースプログラム

カメラベースのライントレースプログラムの流れを確認しよう

ライントレースプログラムの流れ

1. EV3関連の初期化を行う。
2. 保存した訓練済みモデルを読み込む。
3. `VideoStream`の初期化を行う。
4. 次のループをタッチセンサーが押されるまで繰り返す。

「angle_prediction_predictor.ipynb」
と同じ処理で良い。

- A) カメラから画像を読み込む。
- B) モデルに画像を入力するために、訓練時と同じ前処理を行う。
- C) モデルに入力値を入れて、予測値（推定制御角値）を得る。
- D) 以下のようにして予測値steerを使いEV3を制御する。

```
ev3.motor_steer(lmotor_port, rmotor_port, 10, int(steer))
```

サンプルプログラム

ライントレースプログラムの実装例をみてみよう

```
import chainer
from chainer import configuration
import chainer.links as L
import chainer.functions as F
from chainer import serializers

from lib.ev3 import EV3
from lib.vstream import VideoStream

# モーターとセンサーの通信ポートを定義
touch_port = EV3.PORT_2
lmotor_port = EV3.PORT_B
rmotor_port = EV3.PORT_C

# ネットワーク定義
class MLP(chainer.Chain):

    def __init__(self):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(300, 256) # 300(20*15) -> 256 units
            self.l2 = L.Linear(256, 256) # 256 units -> 256 units
            self.l3 = L.Linear(256, 1) # 256 units -> 1

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

```
ev3 = EV3()
ev3.enable_watchdog_task()
ev3.motor_config(lmotor_port, EV3.LARGE_MOTOR)
ev3.motor_config(rmotor_port, EV3.LARGE_MOTOR)
ev3.sensor_config(touch_port, EV3.TOUCH_SENSOR)

print("Push the touch sensor to start the linetracer")
while not ev3.touch_sensor_is_pressed(touch_port):
    pass

# タッチセンサーがリリースされたことを確認
while ev3.touch_sensor_is_pressed(touch_port):
    pass

# 推論を速く行うための設定
with configuration.using_config('train', False), chainer.using_config('enable_backprop', False):
    while True:
        # タッチセンサーが押されたらループを抜ける
        if ev3.touch_sensor_is_pressed(touch_port):
            break
        im = vs.read() # PIL形式で画像を読み込み
        im = np.asarray(im, dtype=np.float32) # NumPy形式に変換
        x = im / 255. # 正規化
        x = x.reshape(1, 300) # (20, 15) -> (1, 300)
        y = model.predictor(x) # steer値を予測
        steer = y.data[0, 0]
        print("predicted steer = {}".format(steer))
        ev3.motor_steer(lmotor_port, rmotor_port, 10, int(steer))

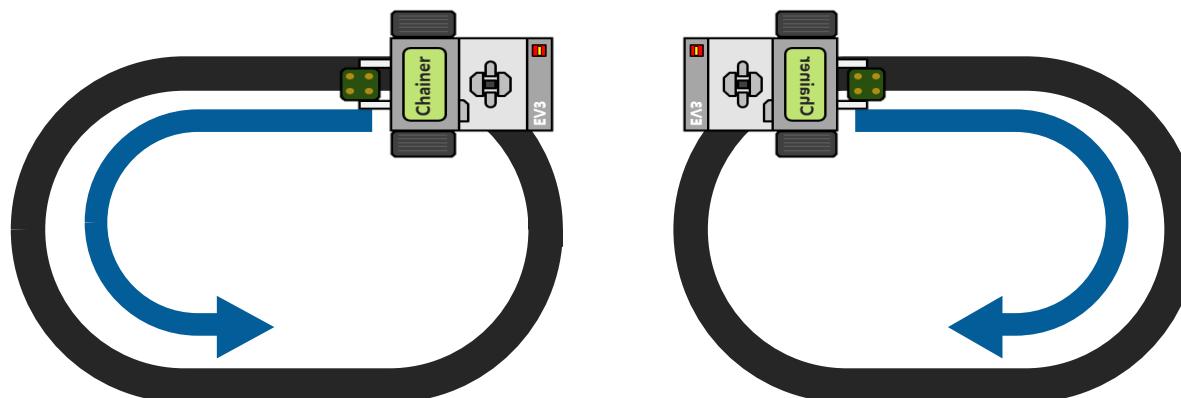
    vs.stop()
    ev3.motor_steer(lmotor_port, rmotor_port, 0, 0)
    ev3.close()
```

練習問題

課題2を応用した練習問題に取り組んでみよう

- 練習問題：双方向を走るモデルの作成

- 「ml_linetrace_logger.ipynb」の制御値計算を修正して、黒線の内側を右回り・左回りしたときのデータをそれぞれ作る。
- 右回り・左回りのデータを一つの訓練データセットとしてマージし、訓練を行う。
- 訓練したモデルで黒線の内側を右回り・左回りの双方向を走るか確認を行う。



解答は
「ml_linetrace_trainer_bi.ipynb」

おわりに

◆謝辞

本テキストの作成に当たり、東京農工大学の博士課程の宮下恵さんには、2017年のPFN インターンシップ期間中にレゴマインドストームEV3とRaspberry Pi 3を連携したシステムの開発に取り組んで頂きました、この時の知見は今回の教材開発のベースとなっています。
また、山梨大学の日置友梨さんには、本テキストの動作検証にご協力いただきました。 厚く御礼を申し上げ、感謝の意を表します。

◆執筆者

株式会社Preferred Networks 丸山史郎
国立大学法人山梨大学 牧野浩二、西崎博光

◆著作権・商標権・利用権

© 2019 Preferred Networks, Inc. and UNIVERSITY of YAMANASHI, All Rights Reserved.

本テキスト・画像の無断転載・複製を固く禁じます。

二次利用不可、商用利用不可、但し教育機関の利用は可能。

Chainer™は、株式会社Preferred Networksの日本国およびその他の国における商標または登録商標です。