機械学習講習会 2024

2024/06/24 - 2024/7/17

traP Kaggle班

@abap34

この資料について

この資料は 東京工業大学デジタル創作同好会 traP Kaggle班 で 2024年に実施した「機械学習講習会」の資料です

機械学習に初めて触れる学部一年生のメンバーが

1. **基本的な機械学習のアイデアを理解** して
2. **最終的にニューラルネットを実際の問題解決に使えるようになること**

を目指しています

(講習会自体については https://abap34.github.io/ml-lecture/supplement/preface.html をみてください)

目次

各種リンク・注意など

- この資料を管理しているレポジトリは https://github.com/abap34/ml-lecture　です
  - 誤りのご指摘などはこちらの Issue または https://twitter.com/abap34 までご連絡ください
- 補足資料なども含めてまとめたものを https://abap34.com/trap_ml_lecture.html から確認できます
- この資料のリンクにはサークルメンバー以外がアクセスできないものが含まれています (oj.abap34.com, dacq.abap34.com など)
  - オンラインジャッジは https://github.com/abap34/ml-lecture-judge
  - コンペプラットフォームは https://github.com/abap34/DacQ-v2 を動かしています
  - どちらもかなり未成熟ですが, 基本的なオンラインジャッジの機能とデータ分析コンペプラットフォームの機能を提供しています. これらをホストすることで同等の環境を構築することができます
- そのほか何かあれば https://twitter.com/abap34 までご連絡ください

謝辞

資料の公開にあたって 東京工業大学情報理工学院情報工学系博士後期課程の @YumizSui さん (大上研究室) と 前田航希さん (@Silviase, 岡崎研究室)に内容について多くの助言をいただきました

この場を借りてお礼申し上げます

機械学習講習会

[1]「学習」

2024/06/24

traP Kaggle班

はじめに

この講習会のゴール

✅ **機械学習の基本的なアイデアを理解して**

問題解決の手段として使えるようになる.

おしながき

第1回 ｜ 学習
第2回 ｜ 勾配降下法
第3回 ｜ 自動微分
第4回 ｜ ニューラルネットワークの構造
第5回 ｜ ニューラルネットワークの学習と評価
第6回 ｜ PyTorch による実装
第7回 ｜ 機械学習の応用, データ分析コンペ

この講習会で扱うこと・扱わないこと

機械学習は非常に広大な分野 ➪ 全7回ではちょっと限界がある

今回の講習会ではとくに**ディープラーニング**についてメインに扱います

- ツールを触るだけで原理は全然やらない
- 原理をやるだけで全然使えない

にならないようにどちらもバランス良くやります

最終的には...

- ✅ 機械学習の基本的なアイデアを説明できるようになる
- ✅ ライブラリに頼らず基本的なアルゴリズム, モデルを実装できるようになる
- ✅ PyTorch を使った基本的なニューラルネットの実装ができるようになる

使うプログラミング言語

Python を使います

慣れている人へ

→ Jupyter Notebook と numpy, matplotlib, scipy, PyTorch あたりのライブラリを使えるようにしておいてください

慣れていない人へ

→ https://abap34.github.io/ml-lecture/supplement/colab.html をみて Google Colaboratory の使い方を覚えておいてください

使うプログラミング言語や前提知識など

1.Pythonを使った初歩的なプログラミング

- if文, for文, 関数 など
- 外部パッケージの利用

  (そこまで高度なことは求めません **ググり力とかの方が大事**)

2.数学の初歩的な知識

- 基本的な行列の演算や操作 (積,転置など)
- 基本的な微分積分の知識 (偏微分など)

(1年前期の (線形代数)　+ (微分積分のさわり) くらい)

がんばりましょう

(ここだけの話機械学習はめちゃくちゃおもしろい)

全7回がんばりましょう！！

第一回: 学習

おしながき

今日の目標

機械学習の基本的な用語を整理して
「学習」ということばをきちんと説明できるようになる.

機械学習 or AI？

- AI(人工知能)
  「人間っぽい知能」を実現しようとする分野・あるいは知能そのもの
- 機械学習(Machine Learning, ML)
  様々な情報から「学習」をして動作するアルゴリズム
  人工知能の一つのかたちと見られることが多い

⬇ つまり？

機械学習 or AI？

機械学習 で **人工知能** を実現

(↔ **スーパーカー** で **爆速移動** を実現)

ここでは一つの定義を紹介しましたが, 実際この二つの言葉に明確に定義や合意があるわけではないです.
手法を厳密に分類してもあまり嬉しいことはないと思いますが, とりあえずこの講習会ではこういう形で整理してみることにします.

学習ってなに？

- 機械学習(Machine Learning, ML)
  様々な情報から「**学習**」をして動作するアルゴリズム

↑ **学習**って何？

今日のテーマ:

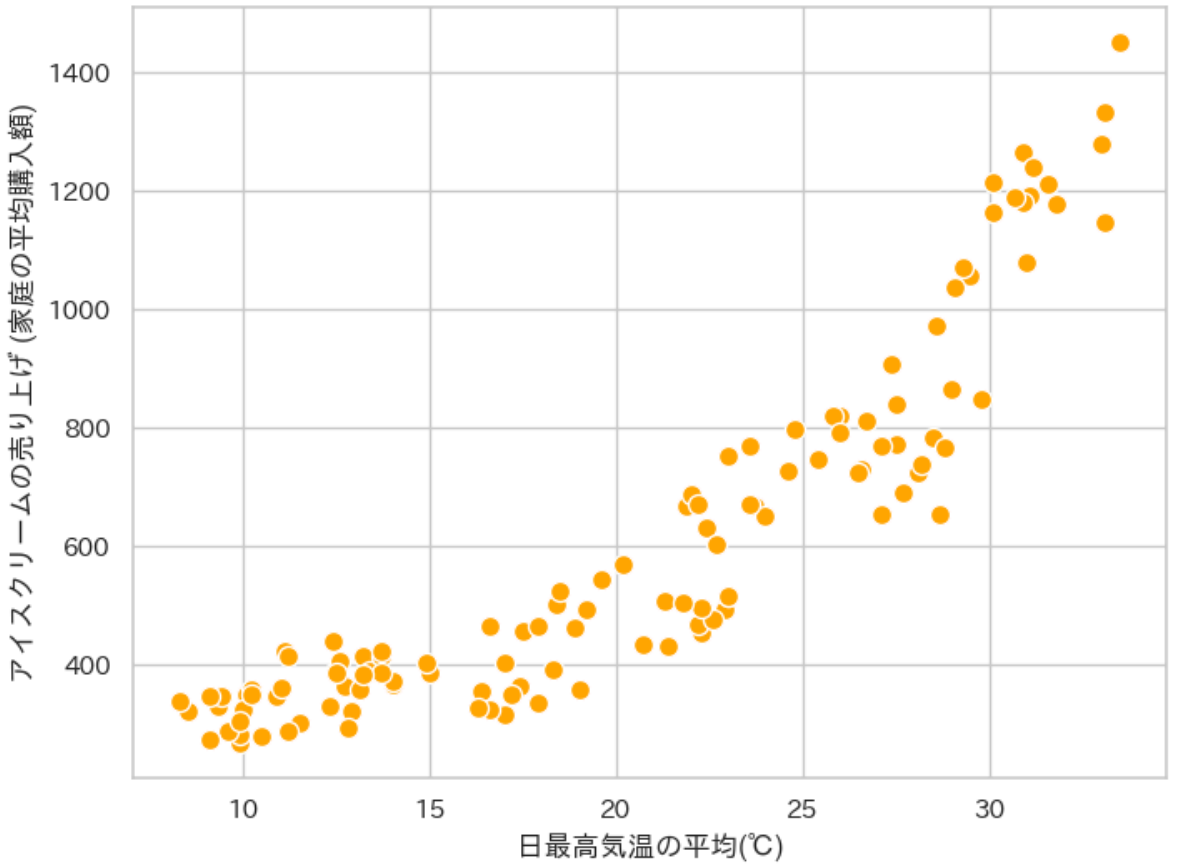
✅ 「学習」を説明できるようになる

「気温」と「アイス」

- 気温↑　→　売れそう
- 気温↓　→　売れなさそう

「アイスの売り上げ」は
「気温」からある程度わかりそう？

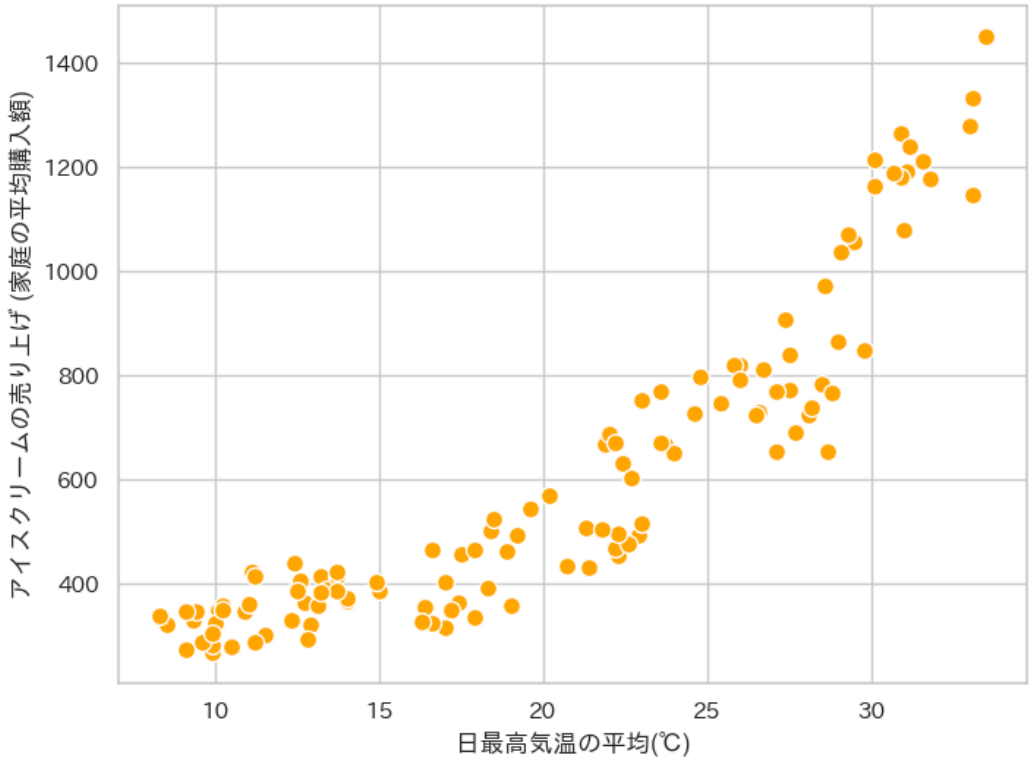

< ...来月の売り上げが予想できたらどのくらい牛乳仕入れたらいいかわかって嬉しいな.



データは https://okumuralab.org/~okumura/stat/160118.html　から引用

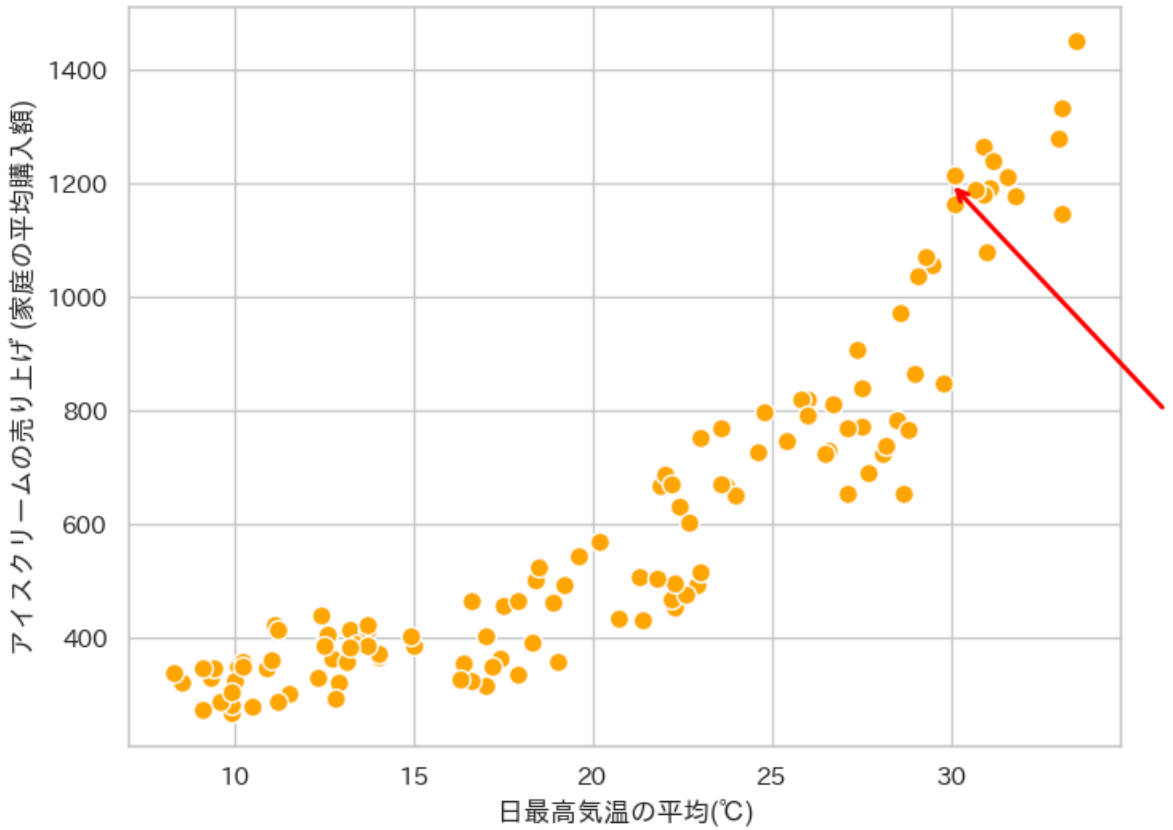
アイスの売り上げを予測するAIをつくる.

< なんか来月の予想平均気温30度って気象庁が言ってたな.



< !!!!!

アイスの売り上げを予測するAIをつくる.




< 過去に30℃のときは...

過去を参照すると...

一番簡単な方法: 過去の全く同じ状況を参照する

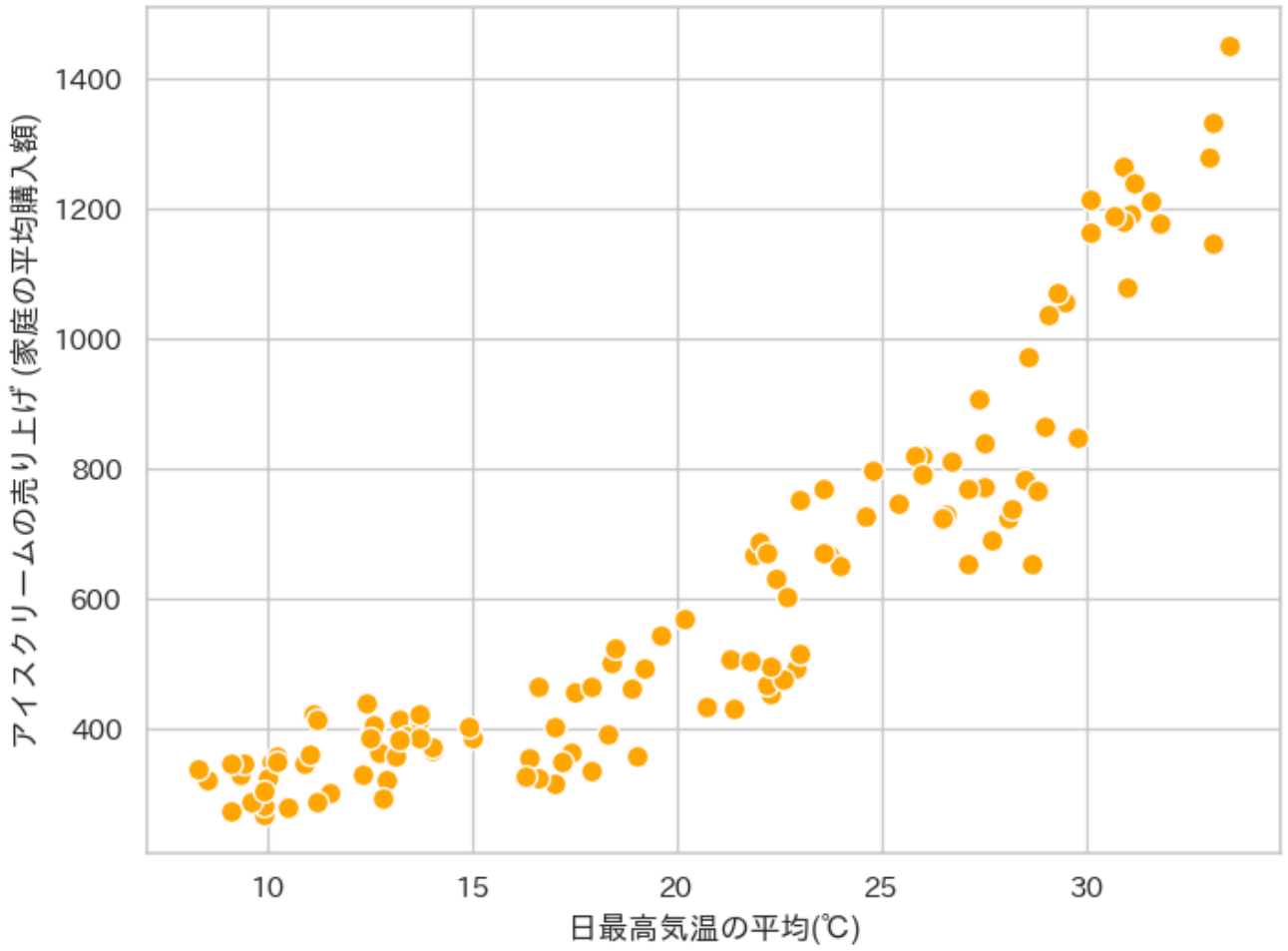
 < これでアイスの売り上げを予測するAIの完成や！

 **<そのまた来月の予想平均気温は40℃です.**

 **< !?**

気象庁ロゴは https://www.jma.go.jp/jma/kishou/info/logo.html より

詰んだ


1400
1200
1000
800
600
400
アイスクリームの売り上げ(家庭の平均購入額)
10
15
20
25
30
日最高気温の平均(℃)

 < 40℃ないやんけ

「予測」を考える

「予測」ってなんだっけ？
→ 入力を受け取ってそれっぽい出力をすること

⇩

今回は 「入力: 気温」 → 「出力: アイスの売り上げ」

そして ✅ **入力は知ってるものだけとは限らない**

予測できるようになる ↔ ？

 ← こいつが本当にやらなくてはいけなかったことは...

売り上げ = f(気温) となる関数 f の推定

このような入力データを受け取り結果を返す f を**モデル**と呼ぶ

線形回帰

売り上げ = f(気温) となる関数 f を作りたい.

⇨ 一旦話を簡単にするために

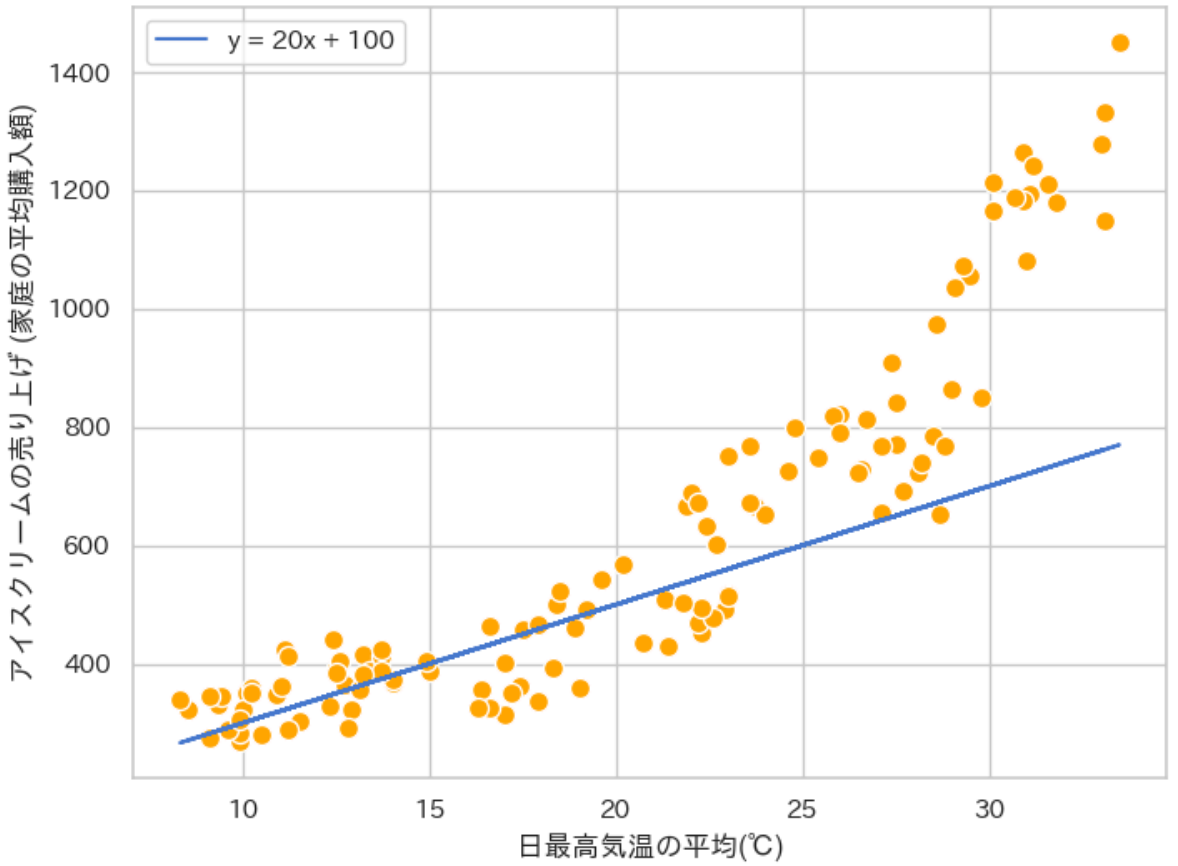
$$\text{「}f\text{(気温)} = a \times \text{気温} + b\text{」}$$

のかたちであることにしてみる.

ためしてみる

$a = 20,\ b = 100$ のとき...

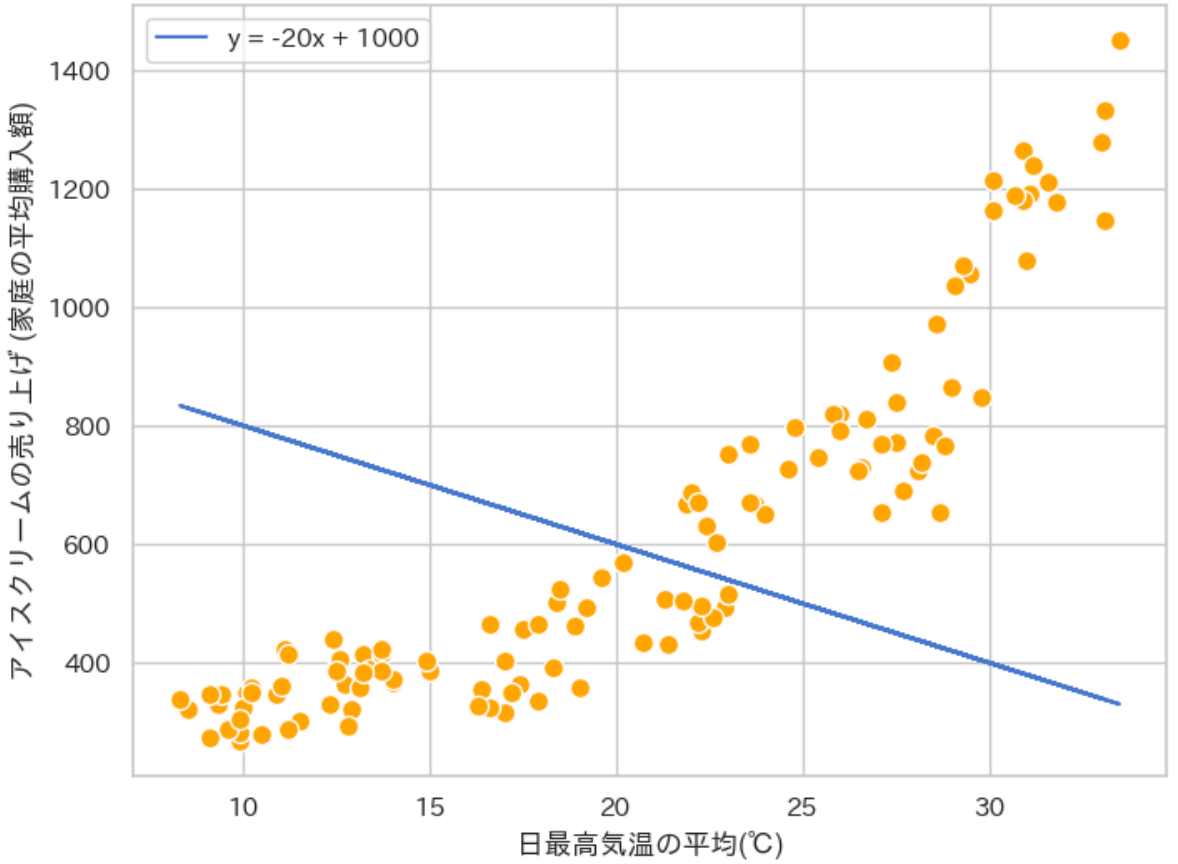
 < わるくない



ためしてみる

$a = -20,\ b = 1000$ のとき...

 < おわてます



パラメータ

a, b を変えることでモデル f の具体的な形が変わった！

このように各モデルが固有に持ってモデル自身の性質を定める

数を 「**パラメータ**」という. (f は a, b をパラメータとして持つ)

⬇

f の構造を決めておけば...

「f の推定 $\leftrightarrow$ f のパラメータの推定」

関数 f がパラメータ $\boldsymbol{\theta}$ を持つことを陽に示すために $f(x; \boldsymbol{\theta})$ と書くことがあります. 今回の場合は $f(x; a, b)$ となります.

ちょっとまとめ

- アイスの売り上げを予測するには, 気温から売り上げを予測する「関数」を構築するのが必要であった.
- いったん, 今回は関数の形として $f(x) = ax + b$ (一次関数) に限って関数を決めることにした.
- この関数はパラメータとして (a, b) をもち, (a, b) を変えることで性質が変わるのがわかった
- これからやる仕事は,「(a, b) をいい感じのものにする」ことで「いい感じの f を作る」こと

さっきの例

$a = 20,\ b = 100$ のとき...

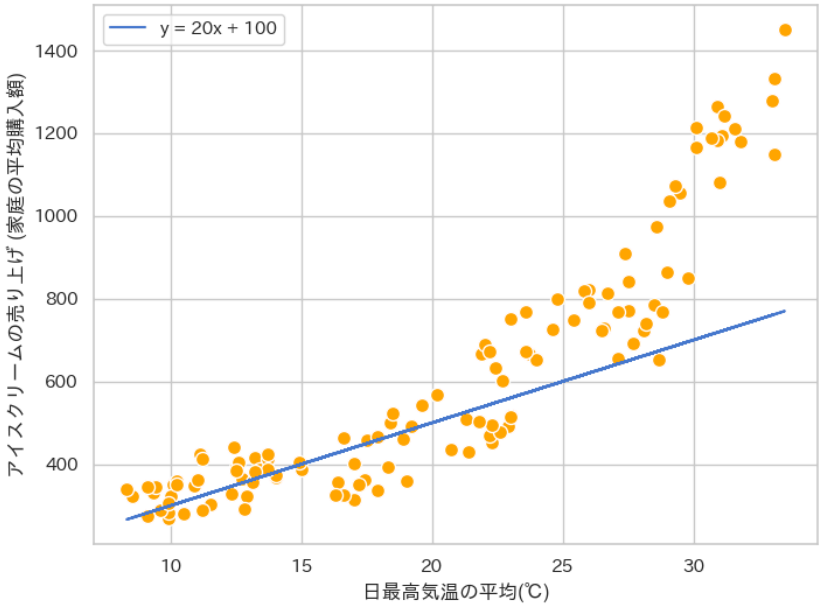
< わるくない

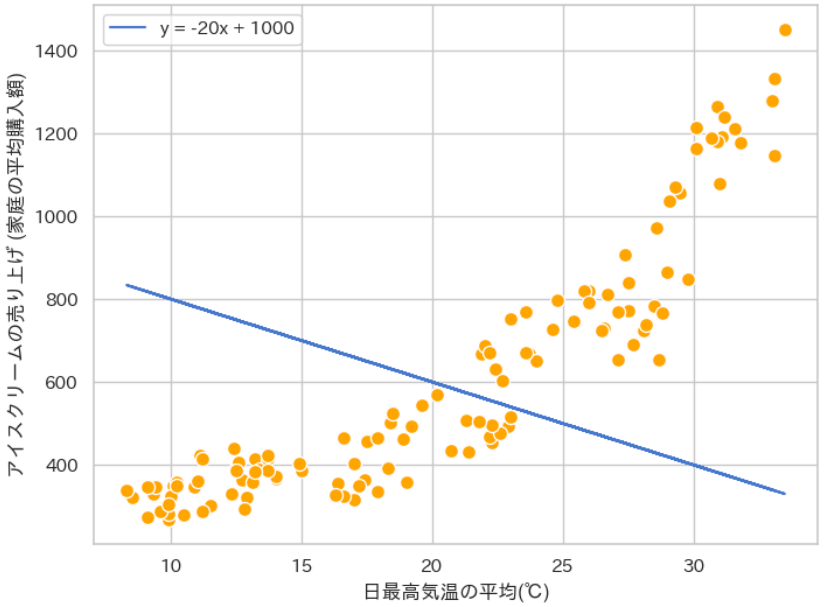
$a = -20,\ b = 1000$ のとき...

< おわてます

⇩ なぜ？

< グラフを見ればわかる.

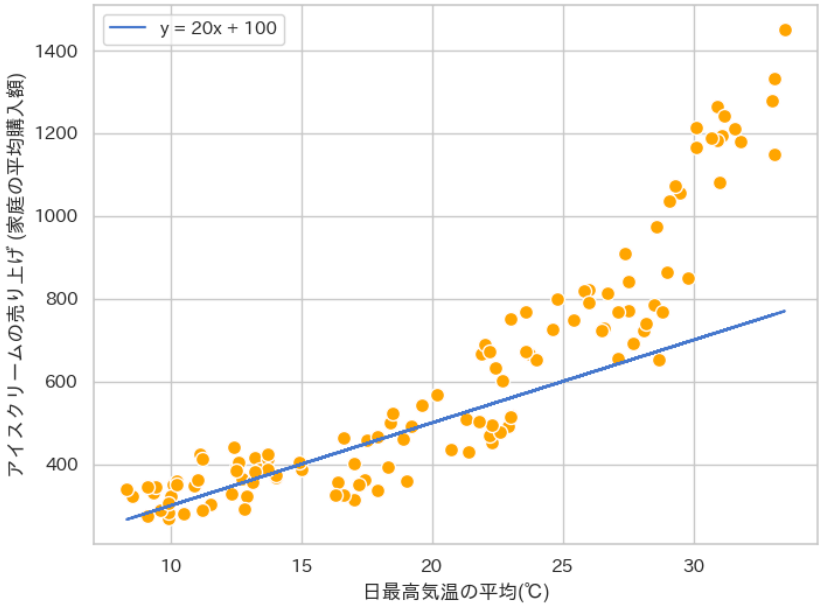


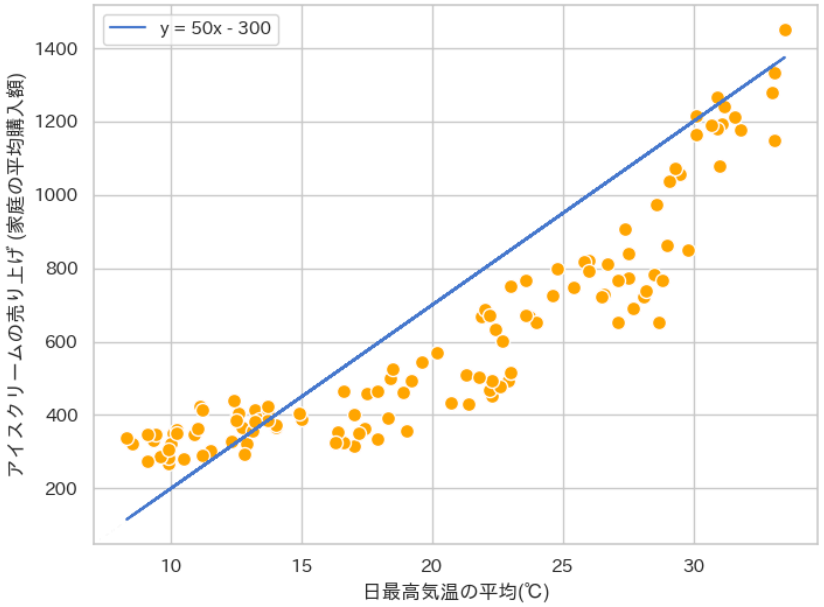


いい勝負?

上: $a = 20,\ b = 100$

下: $a = 50,\ b = -300$





破綻

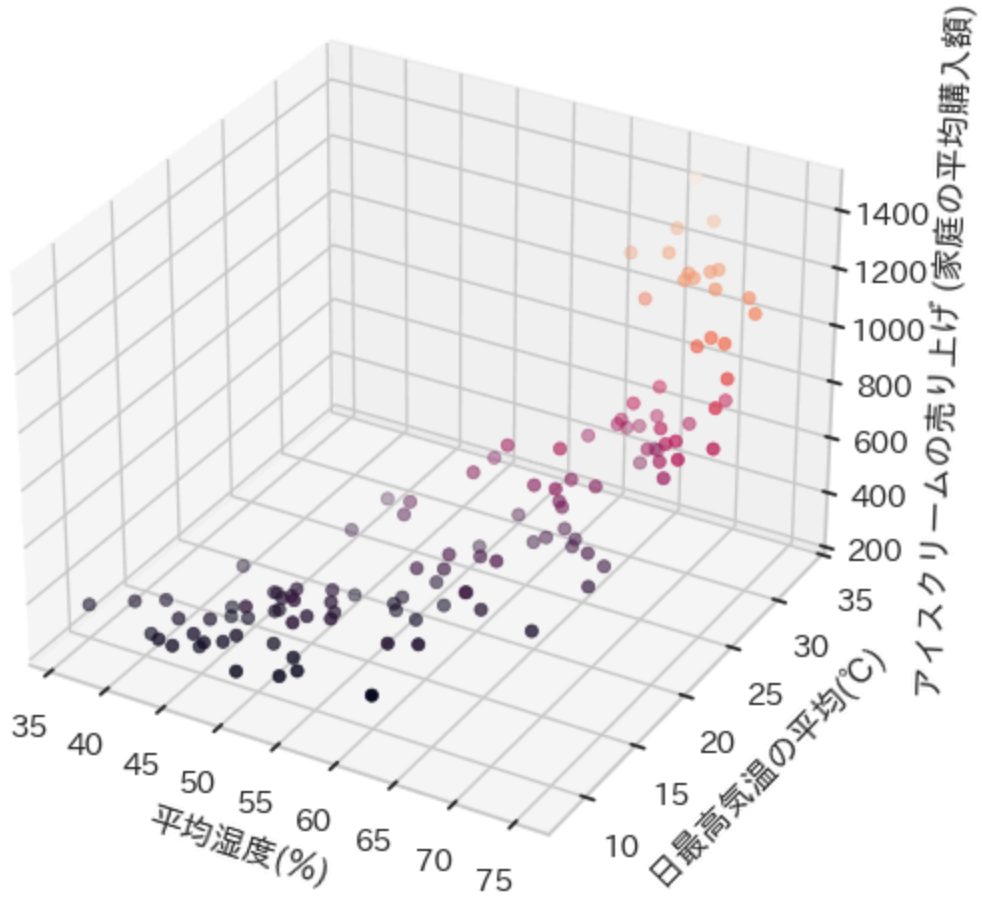
🤔💭 湿度や人口、子供の割合なんかも売り上げとは関係しそうだからこれらも入力に入れたいな。

$$y = f(\text{気温}, \text{湿度}, \text{人口}, \cdots)$$

⇩ **グラフが書けない！**

案1. 高次元の存在になる

案2. 定量的な指標を考える



定量的な指標を考える: 損失関数の導入

良さとは？

⇩

悪くなさ

⇩

悪くなさとは何か？

⇩

データと予測の遠さ

平均二乗誤差(Mean Squared Error)

平均二乗誤差(Mean Squared Error)

$$\frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - f(x_i)\right)^2$$

y_i：実際の値　(確定値) ... 過去のアイスの売り上げ
f：モデル
x_i：入力データ　(確定値) ... 過去の気温

なぜ差を二乗するのか疑問に思った人もいるかもしれません.
全てをここで話すと情報量過多なので一旦置いといてあとで軽く議論します.(末尾の付録)

計算例

$\boldsymbol{x} = (50, 80)^T$, $\boldsymbol{y} = (140, 200)^T$, $f(x) = 2x + 50$ のとき,

$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - f(x_i)\right)^2 &= \frac{1}{2}\left((140 - (2 \times 50 + 50))^2 + (200 - (2 \times 80 + 50))^2\right) \\
&= \frac{1}{2}\left((140 - 150)^2 + (200 - 210)^2\right) \\
&= \frac{1}{2}\left((-10)^2 + (-10)^2\right) \\
&= \frac{1}{2} \times 200 \\
&= 100
\end{aligned}
$$

損失関数

このモデルの悪くなさを定義する関数を「損失関数」と呼ぶ.

学習とは？

⇨ ✅ **「損失関数を最小にする f のパラメータを探す過程」**

何を動かして損失を小さくする？

Q. 損失は何の関数？（何を動かして損失を小さくする？）

✅ 各 x_i, y_i は変数みたいな見た目だけど **「もう観測された確定値」**

$$\mathcal{L}(a,b) = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - f(x_i;a,b)\right)^2$$

ものすごく進んだ話: たまに「入力データ」っぽいものに当たるものについても変数とみることもあります.
自分の知っている話だと DeepSDF という三次元形状を表現する NN では latent code と呼ばれる物体固有の表現を表すベクトルも変化させて損失関数を最小化していました.

いい勝負だったやつの計算例

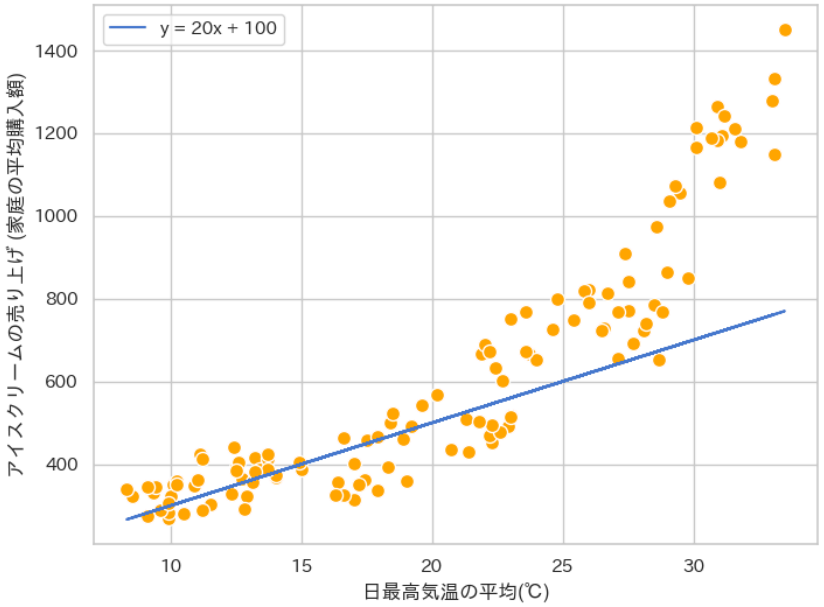
上: $a = 20,\ b = 100$
下: $a = 50,\ b = -300$

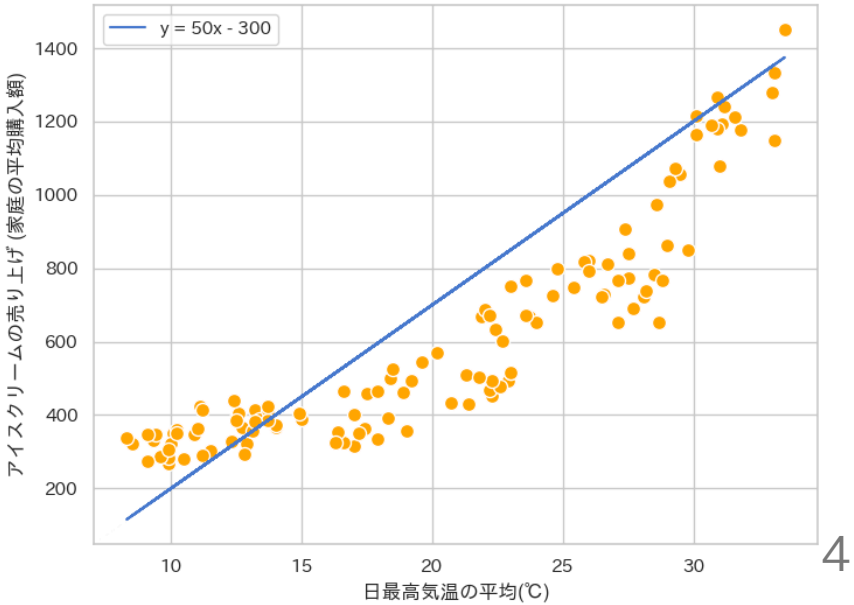
頑張って計算すると,

$$\mathcal{L}(20, 100) = 40268.55$$
$$\mathcal{L}(50, -300) = 39310.45$$

⇩

$$a = 50,\ b = -300\ \ win$$

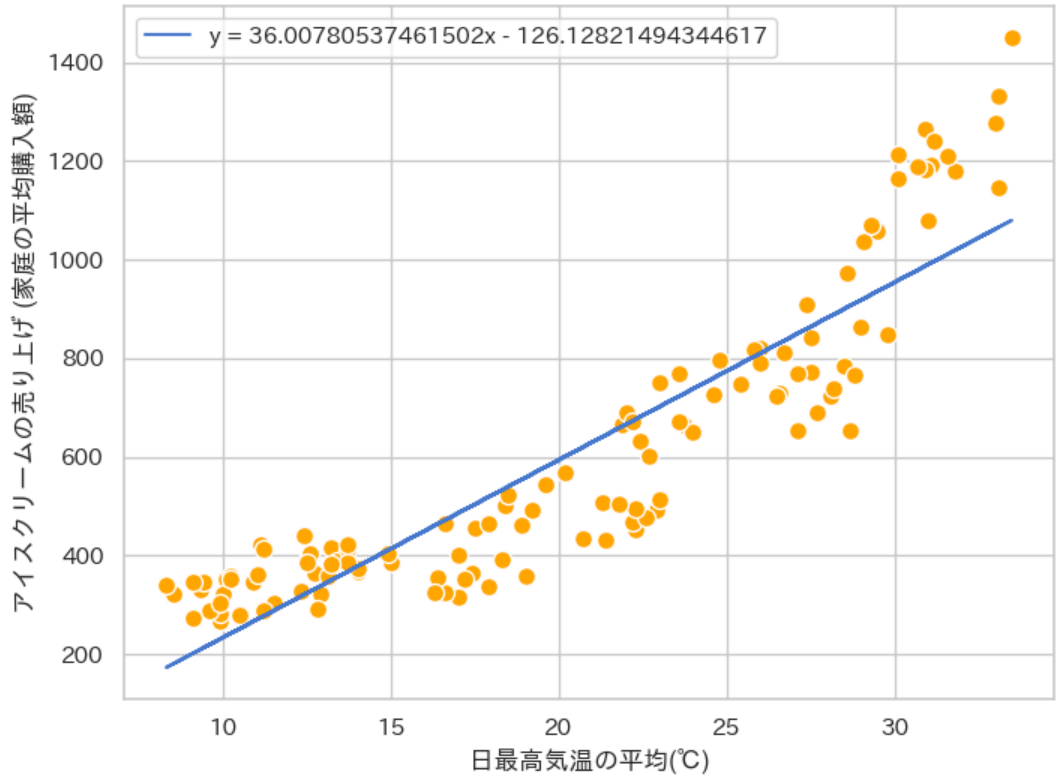




実は今回は

$a \approx 36.00780537461501$
$b \approx 126.12821494344632$

で $\mathcal{L}(a, b)$ が最小



当然の疑問

いや
それ
どう
やったの

次回予告

第二回：勾配降下法

まとめ

- アイスの売り上げを予測するには気温から売り上げを予測する「関数」を構築するのが必要であった.
- いったん, 今回は関数の形として $f(x) = ax + b$ (一次関数) に限って関数を決めることにした.
- この関数はパラメータとして (a, b) をもち, (a, b) を変えることで性質が変わるのがわかった
- モデルの「よさ」のめやすとして 「損失関数」を導入した
- パラメータを変えることで損失関数を最小化する過程のことを「学習」と呼ぶ

付録: なぜ二乗するのか？

レベル1の説明

⇨ 性質がいいから

- 微分可能で導関数も簡単　(絶対値関数は微分不可能な点がある)
- 計算もそんなに大変ではない (百乗誤差などと比べて)

理論的なことを考えると微分可能でないと大変なことが多いです.
一方で現実の最適化だと微分不可能な点が有限個(何なら可算無限個) あっても何とかなることが多いです.

付録: なぜ二乗するのか？

レベル2の(ちゃんとした)説明

⇨ 誤差が正規分布 $\mathcal{N}(0, \sigma^2)$ にしたがうと仮定したとき,
二乗誤差の最小化は尤度の最大化に対応する

付録: なぜ二乗するのか？

[証明]

$y_i = f(x_i) + \epsilon_i,\ \epsilon_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2)$ とする.

このとき $y_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(f(x_i), \sigma^2)$ より
尤度は

$$\prod_{i=0}^{n-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f(x_i))^2}{2\sigma^2}\right)$$

σ^2 が固定されていることに注意すると,
これの最大化は結局 $\sum_{i=0}^{n-1}(y_i - f(x_i))^2$ の最小化に帰着する. □

機械学習講習会

[2]「勾配降下法」

2024/06/25

traP Kaggle班

まとめ

- アイスの売り上げを予測するには, 気温から売り上げを予測する「関数」を構築するのが必要であった.
- いったん, 今回は関数の形として $f(x) = ax + b$ (一次関数) に限って,関数を決めることにした.
- この関数は, パラメータとして (a, b) をもち, (a, b) を変えることで性質が変わるのがわかった
- モデルの「よさ」のめやすとして, 「損失関数」を導入した
- パラメータを変えることで損失関数を最小化する過程のことを「学習」と呼ぶ

前回到達したところ...

$a,\ b$ を動かすことで....

$\mathcal{L}(a,b) = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - f(x_i;a,b)\right)^2$ を小さくしたい 🥺

「関数の最小化」を考える

問題

最小化してください.

$$f(x) = x^2 + 4x + 6$$

「関数の最小化」を考える

問題

最小化してください.

$$f(x) = x^2 + 4x + 6$$

解答

$$f(x) = x^2 + 4x + 6 = (x+2)^2 + 2$$

$$\therefore\ x = -2 \text{ のとき最小値}$$

どう「解けた」？？

- 簡単な数式の操作で解けた！
- 機械的に書くなら

  「 $ax^2 + bx + c$ を最小にする x は $x = -\dfrac{b}{2a}$ 」という公式を使った

プログラムに起こすと...

```
# ax^2 + bx + c を最小にする x を返す関数.
def solve(a, b, c):
    return -b / (2 * a)
```

第二問

最小化してください.

$$f(x) = x^2 + e^{-x}$$

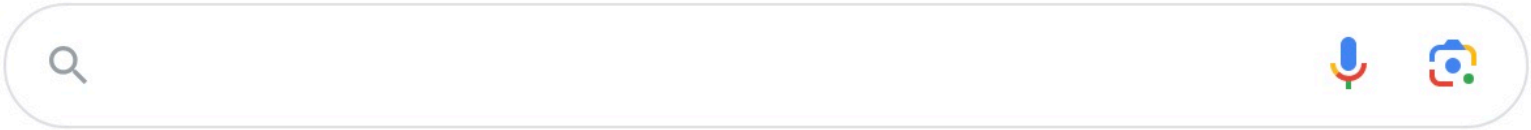
第二問

$f'(x) = 2x - e^{-x}$ なので, 最小値であることの必要条件 $f'(x) = 0$ を調べると...

$$2x - e^{-x} = 0$$

を満たす x を考えると.....

?



Google 検索 I'm Feeling Lucky


Google
wolfram alpha
wolfram alpha
wolfram alpha 積分
wolfram alpha 微分方程式
wolfram alpha 微分

WOLFRAM言語とMATHEMATICAの開発元による

WolframAlpha

solve 2x - e^{-x} = 0

自然言語　数学入力　拡張キーボード　例を見る　アップロード　ランダムな例を使う

WOLFRAM言語とMATHEMATICAの開発元による

WolframAlpha

solve 2x - e^{-x} = 0

自然言語　数学入力　拡張キーボード　例を見る　アップロード　ランダムな例を使う

入力解釈

$2x - e^{(-x)} = 0$　を解く

結果

$x = W_n\left(\frac{1}{2}\right)$, $n \in \mathbb{Z}$

W_k(z) は乗積対数関数の解析接続です

Z は整数の集合です

実数解　表示桁数を増やす

ランベルトのW関数

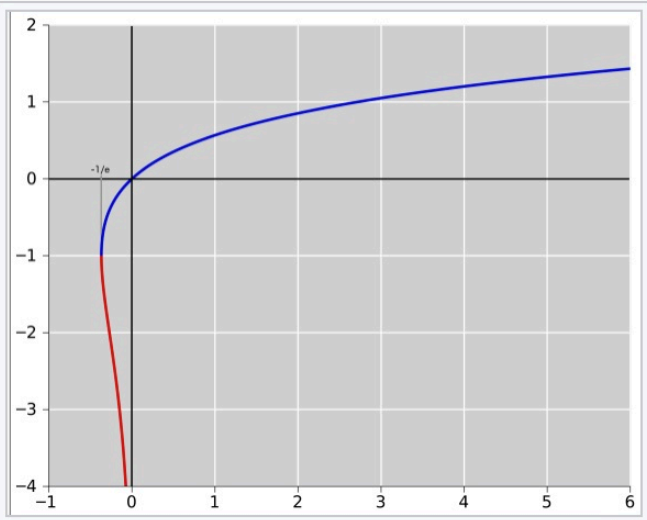
出典: フリー百科事典『ウィキペディア（Wikipedia）』

ランベルトのW函数（ランベルトのWかんすう、英: *Lambert W function*）あるいは**オメガ函数** (*ω function*)、対数積（*product logarithm*; 乗積対数）は、函数 $f(z) = ze^z$ の逆関係の分枝として得られる函数 W の総称である。ここで、e^z は指数函数、z は任意の複素数とする。すなわち、W は $z = f^{-1}(ze^z) = W(ze^z)$ を満たす。

上記の方程式で、$z' = ze^z$ と置きかえれば、任意の複素数 z' に対する W 函数（一般には W 関係）の定義方程式

$$z' = W(z')e^{W(z')}$$

を得る。



$W(x)$ のグラフの $W > -4$ および $x < 6$ の部分。$W \geq -1$ なる上の枝を主枝 W_0 と言い、$W \leq -1$ なる下側の分枝を W_{-1} という。

函数 f は単射ではないから、関係 W は（0 を除いて）多価である。仮に実数値の W に注意を制限するとすれば、複素変数 z は実変数 x に取り換えられ、関係の定義域は区間 $x \geq -1/e$ に限られ、また開区間 $(-1/e, 0)$ 上で二価の函数になる。さらに制約条件として $W \geq -1$ を追加すれば一価函数 $W_0(x)$ が定義されて、$W_0(0) = 0$ および $W_0(-1/e) = -1$ を得る。それと同時に、下側の枝は $W \leq -1$ であって、$W_{-1}(x)$ と書かれる。これは $W_{-1}(-1/e) = -1$ から $W_{-1}(-0) = -\infty$ まで単調減少する。

ランベルト W 関係は初等函数では表すことができない[1]。ランベルト W は組合せ論において有用で、例えば木の数え上げに用いられる。指数函数を含む様々な方程式（例えばプランク分布、ボーズ–アインシュタイン分布、フェルミ–ディラック分布などの最大値）を解くのに用いられ、また$y'(t) = ay(t-1)$ のような遅延微分方程式（英語版）の解としても生じる。生化学において、また特に酵素動力学において、ミカエリ

?

一般の関数の最小化

いいたかったこと

✅ **このレベルの単純な形の関数でも解をよく知っている形で書き表すことは難しい**

もう一度目的を整理する

われわれの目標...

誤差 $\mathcal{L}(a, b)$ を最小化したかった.

効いてくる条件①

Q. 厳密な最小値を得る必要があるか？

効いてくる条件①

A. No. 厳密に最小値を得る必要はない

数学の答案で最小値 1 になるところを 1.001と答えたら当然 🤦

一方 「誤差 1」 が 「誤差1.001」 になってもほとんど変わらない

効いてくる条件②

$\mathcal{L}$ は非常に複雑になりうる

第一回では **話を簡単にするために** $f(x) = ax + b$ の形を考えたが...

(特にニューラルネットワーク以降は) **非常に複雑になりうる**

$$\mathcal{L}(\mathbf{W^{(1)}}, \mathbf{W^{(2)}}, \cdots, \mathbf{W^{(n)}}, \mathbf{b^{(1)}}, \mathbf{b^{(2)}}, \cdots, \mathbf{b^{(n)}}) = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - {W^{(n)}}^T \sigma\left(\cdots\sigma\left({W^{(1)}}^T x_i + b^{(1)}\right)\cdots + b^{(n-1)}\right)\right)^2,\ \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{\sum_{p\in S} \left|f(p;\boldsymbol{\theta}) - \mathcal{F}(p)\right|^2 \cdot \omega_p}{\sum_{p\in S} \omega_p}$$

$$\vdots$$

複雑そうな式を気分で乗せただけなのであまり意図はありません

われわれに必要な道具

✅ 非常に広い範囲の関数に対して

そこそこ小さい値を探せる方法

勾配降下法

微分のおさらい

微分係数

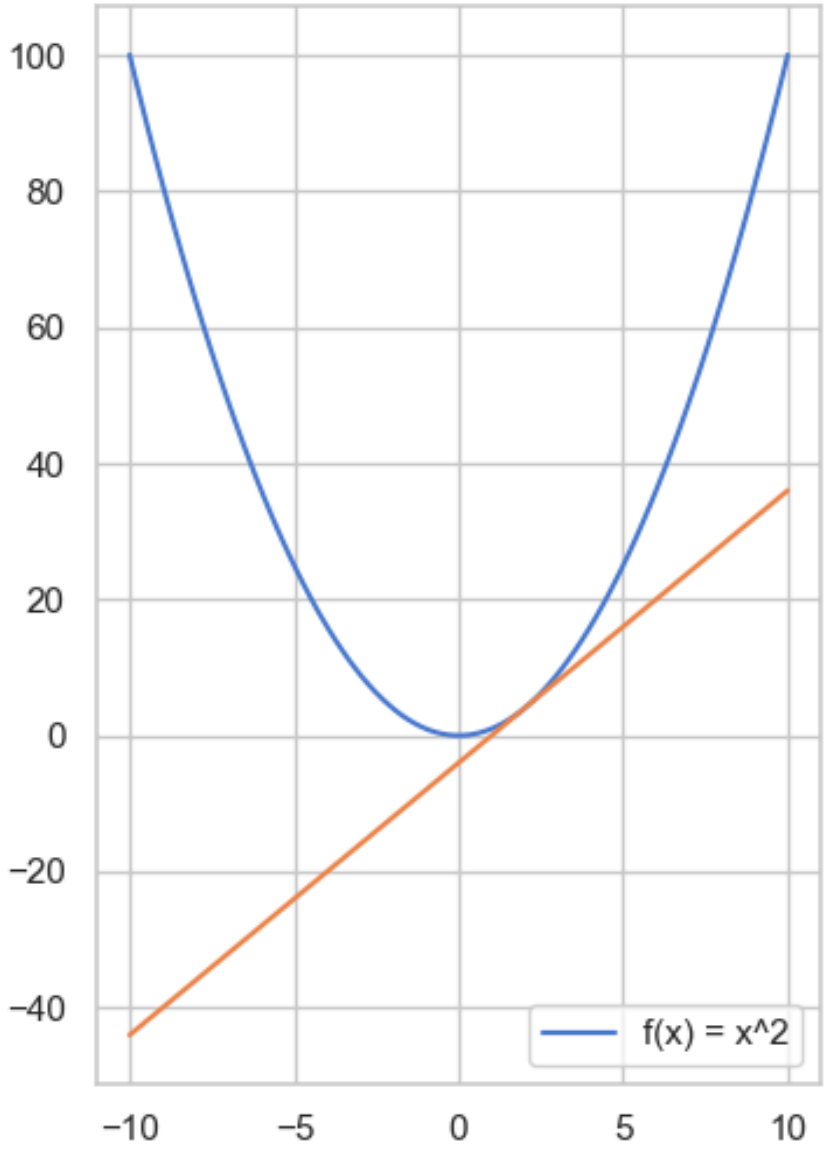
関数 f の x における微分係数

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

微分は「傾き」

微分係数

$f'(x)$ は x における接線の傾き



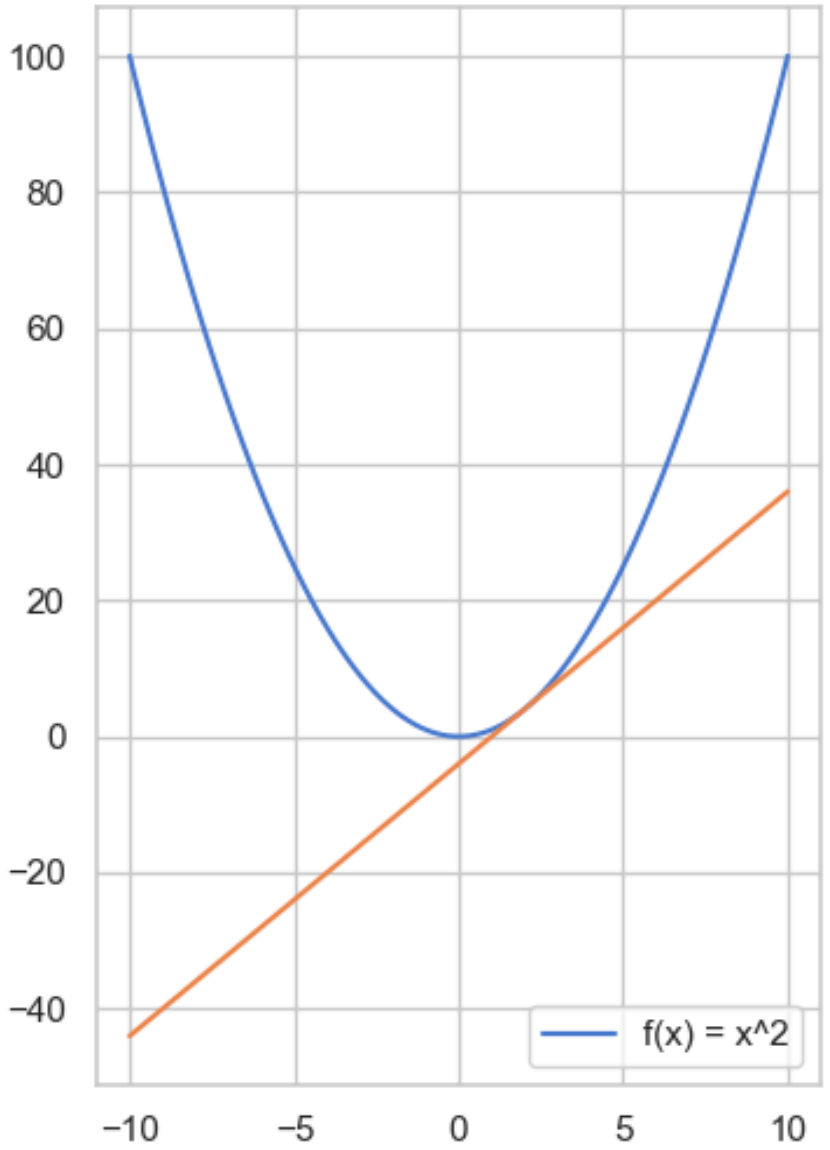
微分は「傾き」

微分係数

$f'(x)$ は, x における接線の傾き

⬇

$-f'(x)$ 方向に関数を
すこし動かすと関数の値はす
こし小さくなる



「傾き」で値を更新してみる

例) $f(x) = x^2$

$x = 3$ で $f(3) = 9,\ f'(3) = 6$

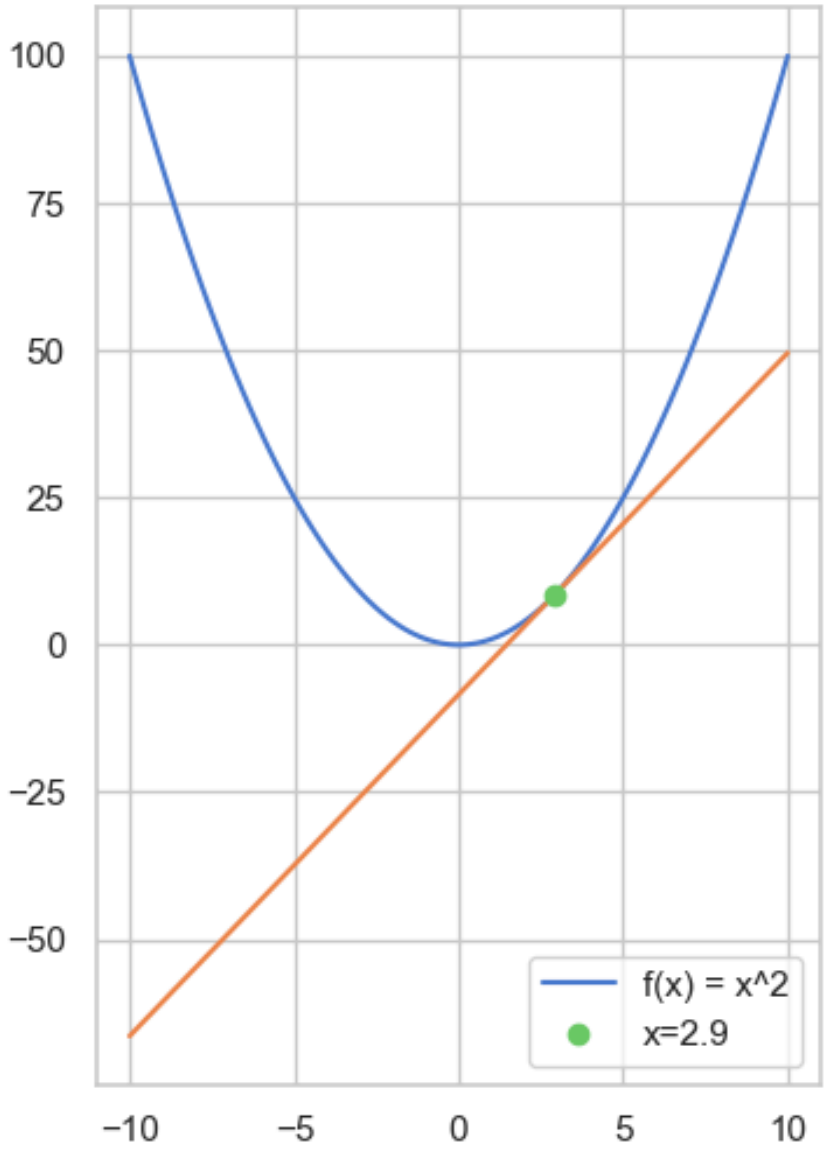
$\therefore -f'(x)$ は負の方向

⬇

すこし負の方向に x を動かしてみる

$f(2.9) = 8.41 < 9$

✅ 小さくなった



「傾き」で値を更新してみる

例) $f(x) = x^2$

$x = 2.9$ で

$f(2.9) = 8.41,\ f'(2.9) = 5.8$

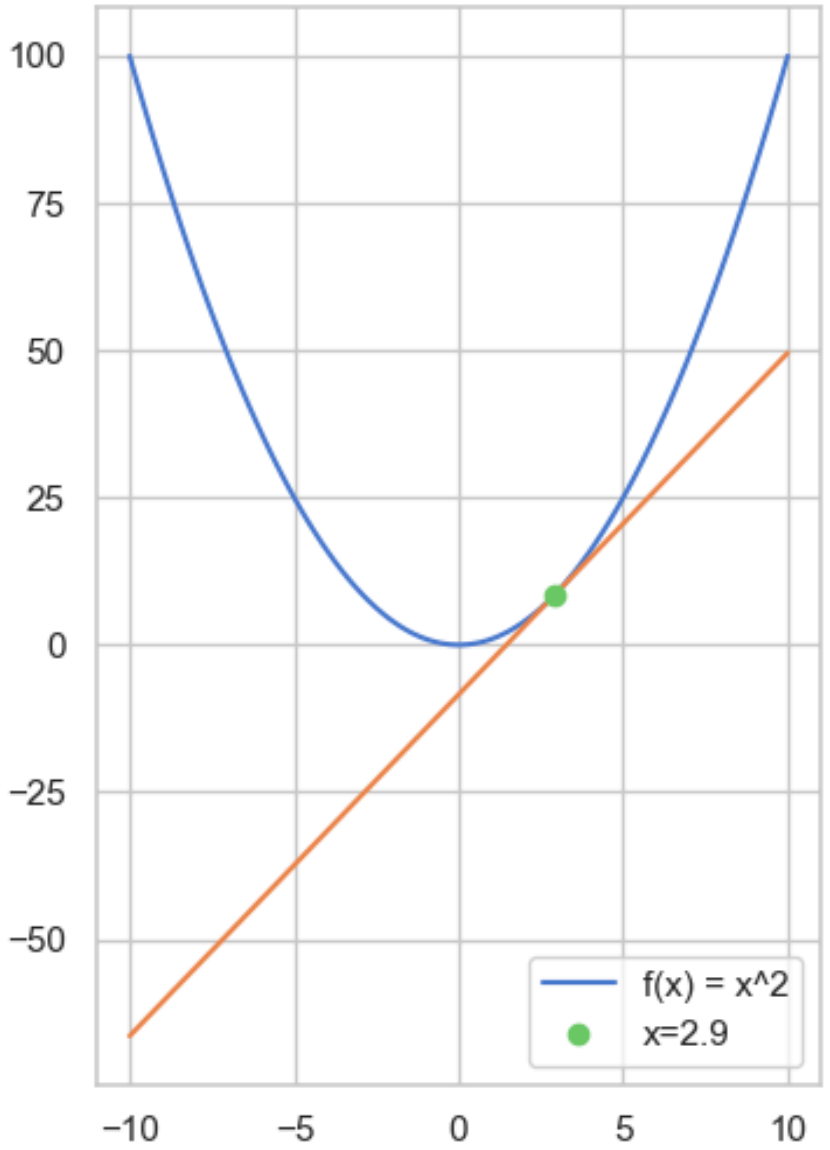
$\therefore -f'(x)$ は負の方向

⬇

すこし負の方向に x を動かしてみる

$f(2.8) = 7.84 < 8.41$

✅ 小さくなった



「傾き」で値を更新してみる

これを繰り返すことで小さい値まで到達できそう！

勾配降下法

勾配降下法

関数 $f(x)$ と初期値 x_0 が与えられたとき,
次の式で $\{x_k\}$ を更新するアルゴリズム

$$x_{k+1} = x_k - \eta f'(x_k)$$

(η は**学習率**と呼ばれる定数)

正確にはこれは最急降下法と呼ばれるアルゴリズムで,「勾配降下法」は勾配を使った最適化手法の総称として用いられることが多いと思います.
(そこまで目くじらを立てる人はいないと思いますし, 勾配降下法あるいは勾配法と言われたらたいていの人がこれを思い浮かべると思います.)

勾配降下法

マイナーチェンジが大量！
(実際に使われるやつは第五回で予定)

$$x_{n+1} = x_n - \eta f'(x_n)$$

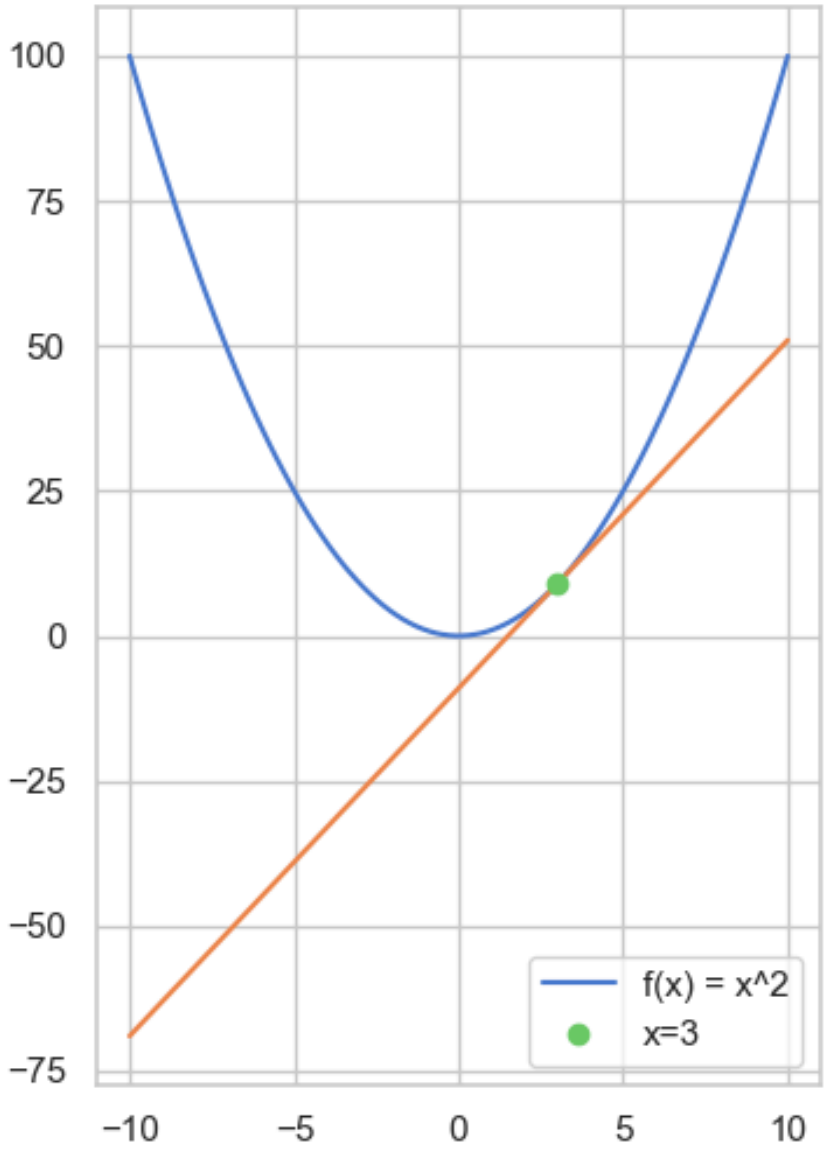
抑えてほしいこと 👀

1. 値が $-f'(x)$ の方向に更新される
2. 学習率によって更新幅を制御する

勾配降下法のお気持ち

値が $-f'(x)$ の方向に更新される

(さっきの説明の通り)



学習率による更新幅の制御

✅ 微分はあくまで「**その点**の情報」

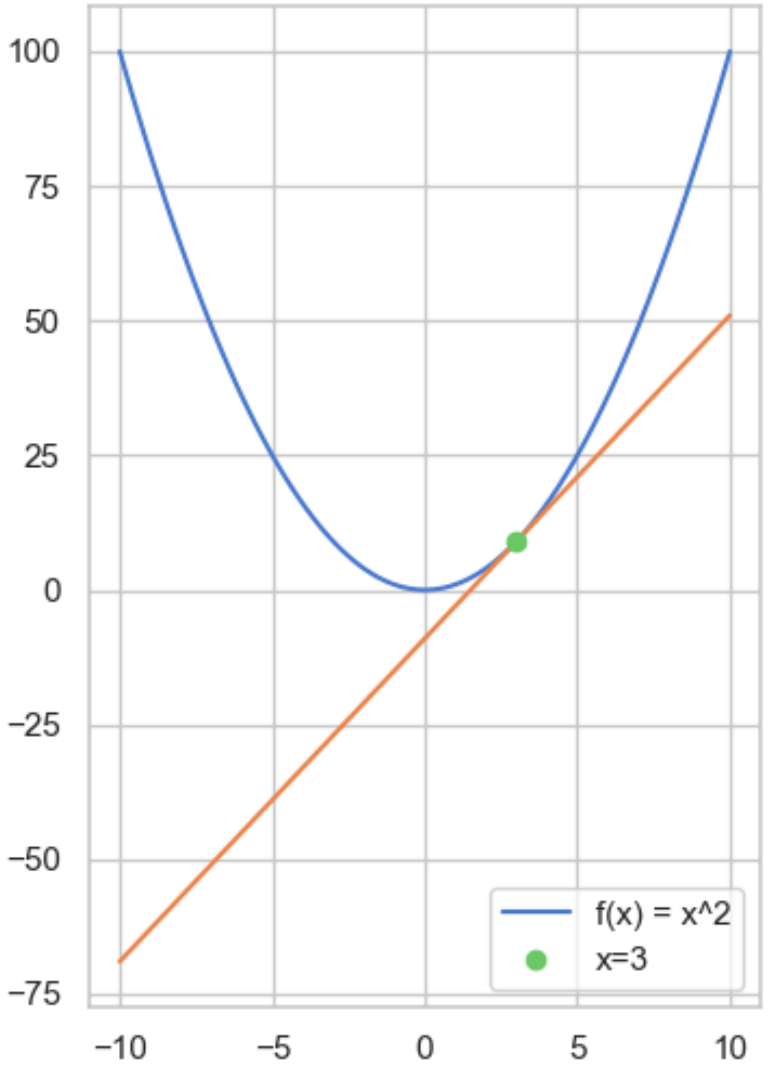
傾向が成り立つのはその周辺だけ

⬇

少しずつ更新していく必要がある

⬇

小さな値 **学習率** η をかけることで
少しずつ更新する



実際にやってみる

$f(x) = x^2$

初期値として $x_0 = 3$
学習率として $\eta = 0.1$ を設定.(この二つは自分で決める！)

$$x_1 = x_0 - \eta f'(x_0) = 3 - 0.1 \times 6 = 2.4$$
$$x_2 = x_1 - \eta f'(x_1) = 2.4 - 0.1 \times 4.8 = 1.92$$
$$x_3 = x_2 - \eta f'(x_2) = 1.92 - 0.1 \times 3.84 = 1.536$$
$$\dots$$
$$x_{100} = 0.0000000006111107929$$

✅ **最小値を与える $x = 0$ に非常に近い値が得られた！**

勾配降下法のココがすごい！

✅ その式を (解析的に) 解いた結果が何であるか知らなくても,
導関数さえ求められれば解を探しにいける

実際にやってみる2

第二問

最小化してください.

$$f(x) = x^2 + e^{-x}$$

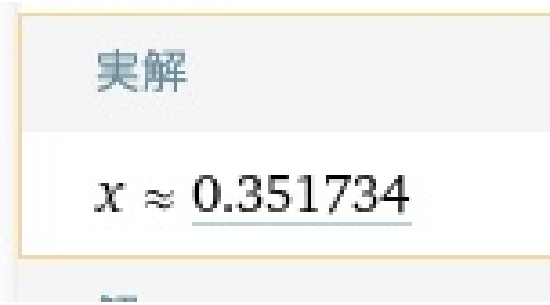
実際にやってみる2

$f'(x) = 2x - e^{-x}$.

初期値として $x = 3$, 学習率として
$\eta = 0.01$ を設定.

$x_0 = 3$
$x_1 = 2.9404978706836786$
$\vdots$
$x_{1000} = 0.35173371125366865$

ヨシ！🐱



Pythonによる実装

```
from math import exp

x = 3
# (注意: $\eta$ は 学習率 (learning rate) の略である lr としています.)
lr = 0.0005

# 最小化したい関数
def f(x):
  return x ** 2 + exp(-x)

# f の x での微分係数
def grad(x):
    return 2 * x - exp(-x)
```

Pythonによる実装

$x_{n+1} = x_n - \eta f'(x_n)$ をコードに起こす

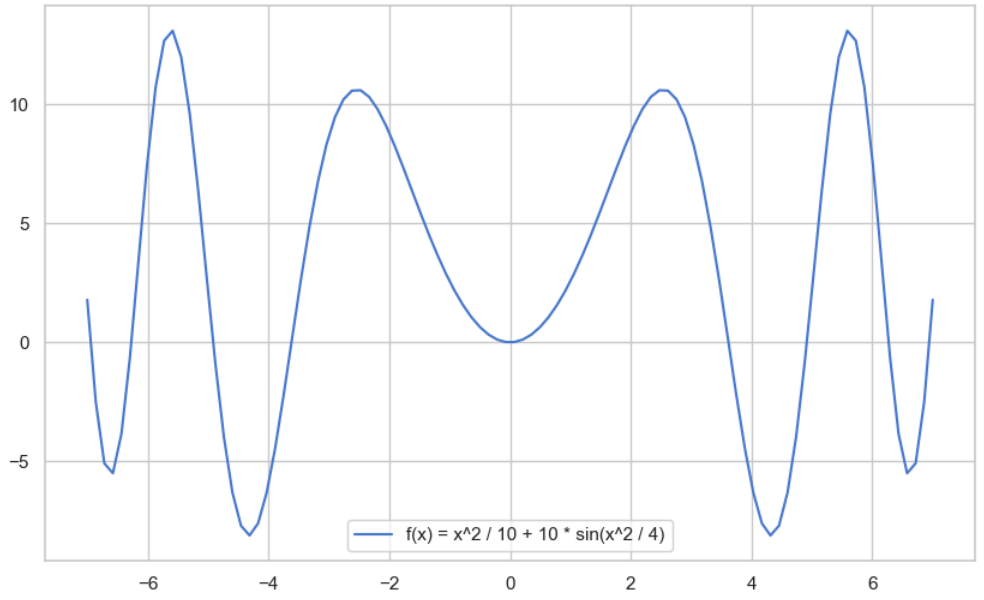
```
for i in range(10001):
    # 更新式
    x = x - lr * grad(x)
    if i % 1000 == 0:
        print('x_', i, '=', x , ', f(x) =', f(x))
```

```
x_ 0 = 2.997024893534184 , f(x) = 9.032093623218246
x_ 1000 = 1.1617489280037716 , f(x) = 1.6625989669983947
x_ 2000 = 0.5760466279295902 , f(x) = 0.8939459518186053
x_ 3000 = 0.4109554481889124 , f(x) = 0.8319008499233866
...
x_ 9000 = 0.3517515401706734 , f(x) = 0.8271840265571999
x_ 10000 = 0.3517383210080008 , f(x) = 0.8271840261562484
```

常に上手くいく？

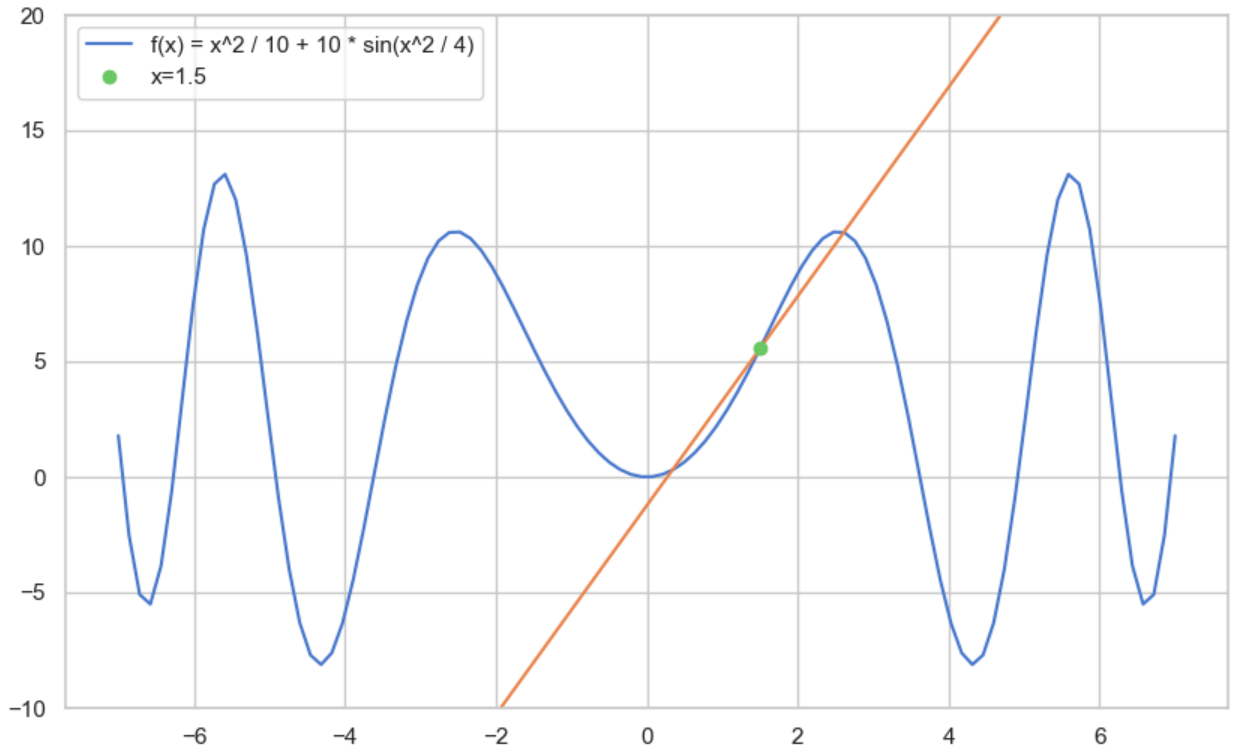
✅ 勾配降下法があまりうまくいかない関数もある

例) $f(x) = \frac{x^2}{10} + 10 \sin\left(\frac{x^2}{4}\right)$



うまくいかない例

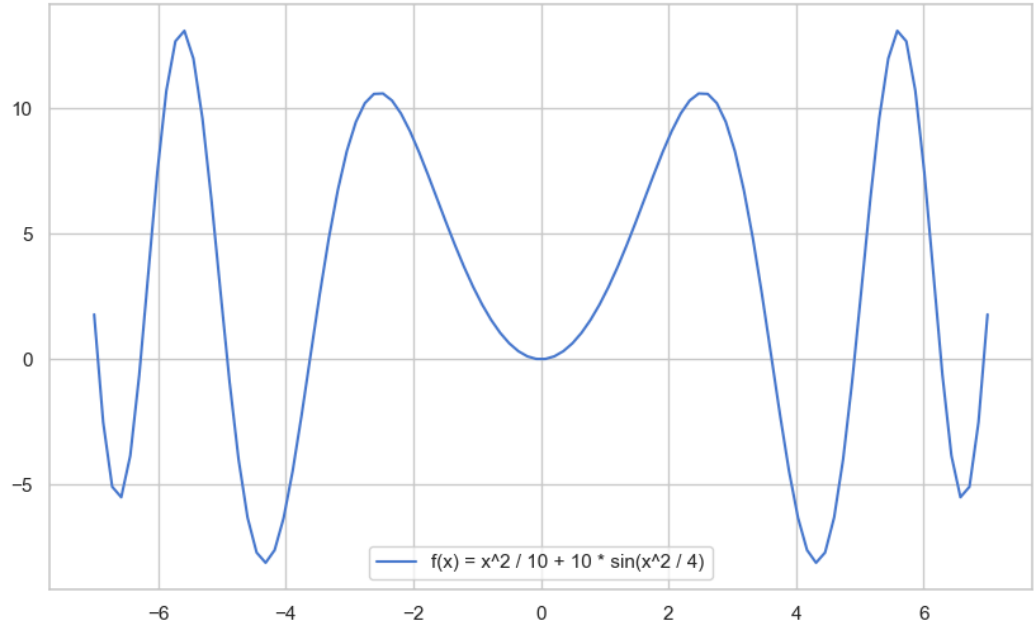
$x = 1.5$ あたりから勾配降下法をすると、$x = 0$ に収束する！



局所最適解への収束

局所最適解 ... 付近では最小値 ($x = -6, -4, 0, 4, 6$ あたりのもの全て)

大域最適解 ... 全体で最小値 ($x = -4, 4$ あたりのもの)



マイナーチェンジ

⇨ なるべく局所最適解に **ハマりまくらない** ように色々と工夫 (詳しくは第5回)

- Momentum

$$v_{n+1} = \alpha v_n - \eta f'(x_n)$$
$$x_{n+1} = x_n + v_{n+1}$$

- AdaGrad

$$h_{n+1} = h_n + f'(x_n)^2$$
$$x_{n+1} = x_n - \frac{\eta}{\sqrt{h_{n+1}}} f'(x_n)$$
$$\vdots$$

多変数関数への応用

多変数関数の場合は,微分係数→勾配ベクトル　に置き換えればOK

$$\boldsymbol{x_{n+1}} = \boldsymbol{x_n} - \eta \nabla f(\boldsymbol{x_n})$$

勾配ベクトルは各変数の偏微分係数を並べたものです. 例えば $f(x,y) = x^2 + y^2$ の (x,y) における勾配ベクトルは $(2x, 2y)$ です.
これを$\nabla f(x,y) = (2x, 2y)$ とかきます. 一年生はちょうど微分積分学第一でやるころかと思うので大きくは扱いませんでしたが, 一変数の場合できちんと理解できていれば大丈夫です.

再掲: 一般の関数の最小化

第三問

最小化してください.

$$-\frac{1}{(x^2+1)}\log\left(\frac{1}{1+e^{-x}}+1\right)$$

嫌です.

次回予告

第三回　自動微分

機械学習講習会 第三回

- 「自動微分」

traP Kaggle班

2024/06/28

前回のまとめ

- 損失関数の最小化を考える上で, 一般の関数の最小化を考えることにした
- 損失関数の厳密な最小値を求める必要はなく, また損失関数は非常に複雑になりうるので, 広い範囲の関数に対してそこそこ上手くいく方法を考えることにした
- たいていの関数に対して, 導関数を求めることさえできればそれなりに小さい値を探しに行けるようになった
- 逆に, **「導関数」は自分で求める必要がある**

実は

いまはね

思い出すシリーズ: 一般の関数の最小化

第三問

最小化してください.

$$-\frac{1}{(x^2+1)}\log\left(\frac{1}{1+e^{-x}}+1\right)$$

$\mathcal{L}$ は非常に複雑になりうる

第一回では **話を簡単にするために** $f(x) = ax + b$ の形を考えたが...

(特にニューラルネットワーク以降は) **非常に複雑になりうる**

$$\mathcal{L}(\mathbf{W^{(1)}}, \mathbf{W^{(2)}}, \cdots, \mathbf{W^{(n)}}, \mathbf{b^{(1)}}, \mathbf{b^{(2)}}, \cdots, \mathbf{b^{(n)}}) = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - {W^{(n)}}^T\sigma\left(\cdots\sigma\left({W^{(1)}}^T x_i + b^{(1)}\right)\cdots + b^{(n-1)}\right)\right)^2,\ \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{\sum_{p\in S}\left|f(p;\boldsymbol{\theta}) - \mathcal{F}(p)\right|^2 \cdot \omega_p}{\sum_{p\in S}\omega_p}$$

$$\vdots$$

自動微分

✅ 人間が微分を行うのは限界がある

➪ 計算機にやらせよう！

自動微分

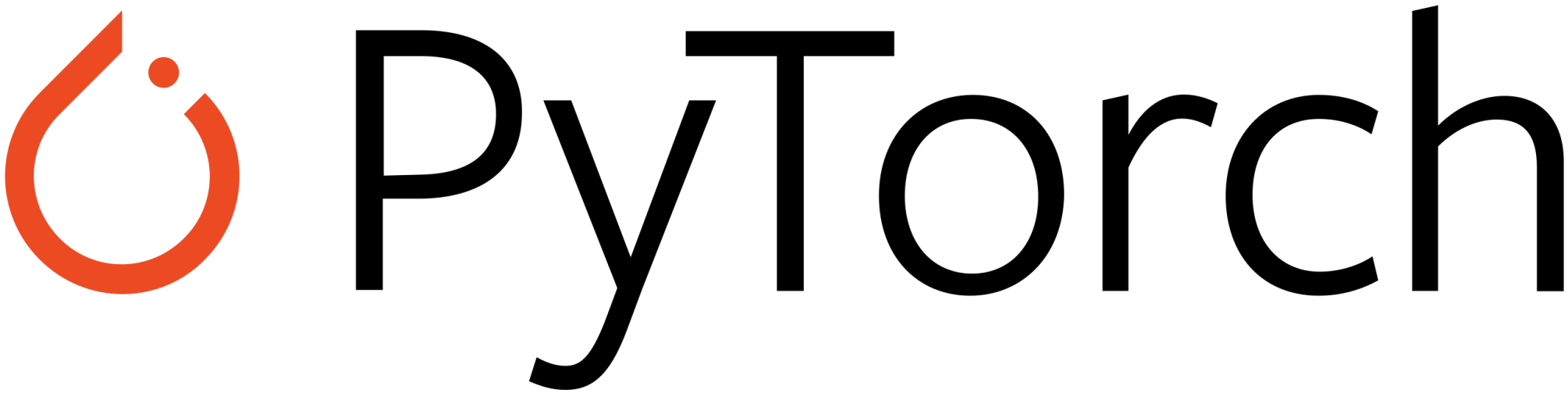
(Automatic Differentiation)

正確には「自動微分」は, コンピュータに自動で微分を行わせる手法のうち, とくに関数を単純な関数の合成と見て連鎖律を利用して, 陽に導関数を求めることなく微分を行う手法を指します. (より狭義に, back propagationを用いるもののみを指すこともあるようです).

おしながき

- PyTorchの導入
- PyTorchを使った自動微分
- 自動微分を使った勾配降下法の実装
- 自動微分の理論とアルゴリズム

PyTorch



自動微分

結論から言うと... **PyTorchを使うと微分ができる.**

```
>>> x = torch.tensor(2.0, requires_grad=True)
>>> def f(x):
...     return x ** 2 + 4 * x + 3
...
>>> y = f(x)
>>> y.backward()
>>> x.grad
tensor(8.)
```

($f(x) = x^2 + 4x + 3$ の $x = 2$ における微分係数 8 が計算されている)

そもそもPyTorchとは？　～深層学習フレームワーク～

事実:

ニューラルネットワークのさまざまな派生系の

- 基本的な部品
- 部品に対してやる作業

は大体同じ！

そもそもPyTorchとは？　～深層学習フレームワーク～

例) 新しい車を開発するときも,部品は大体同じ,組み立ても大体同じ

⇩

毎回同じことをみんながそれぞれやるのは面倒

⇩

共通基盤 を提供するソフトウェアの需要がある

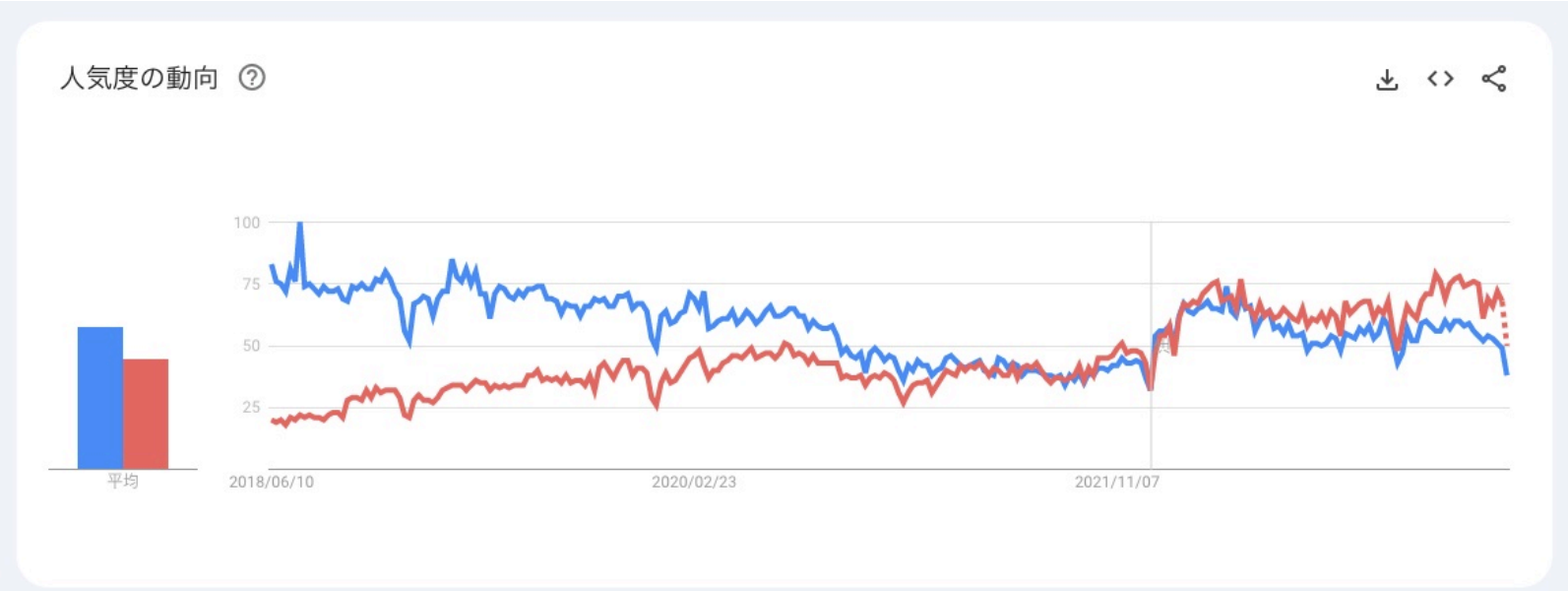
どの組み立て機を使う？ 有名なフレームワークたち

- TensorFlow
  - (主に) Googleが開発したフレームワーク
  - 産業界で人気 (が, 最近はPyTorchに押され気味)
- PyTorch
  - (主に) Facebookが開発したフレームワーク
  - 研究界で人気 (最近はみんなこれ？)
- Keras
  - いろんなフレームワークを使いやすくしたラッパー (おもに TensorFLow)
  - とにかくサッと実装できる
- JAX/Flax, Chainer, MXNet, Caffe, Theano, ...

そもそもPyTorchとは？　～深層学習フレームワーク～

どれがいいの？

➪ PyTorchを使っておけば間違いない（と，思います）



(赤: PyTorch, 青: TensorFlow)

なので

今回は PyTorch を使います！

- 高速な実行
- 非常に柔軟な記述
- 大きなコミュニティ
- 超充実した周辺ライブラリ
- サンプル実装の充実 (← **重要!!**)

大体の有名フレームワークにそこまで致命的な速度差はなく, 記述に関しては好みによるところも多いです.PyTorchの差別化ポイントは, 有名モデルの実装サンプルが大体存在するという点です.
実際に論文を読んで実装するのは骨の折れる作業なので, サンプルが充実していのはとても大きな利点です.

今日のお話

✅ 自動微分ライブラリとしての PyTorch の使い方を習得して，

手で微分するのをやめる

Tensor 型

数学の 「数」 に対応するオブジェクトとして,PyTorchでは

Tensor 型

を使う

Tensor とは？

Tensor (テンソル)

スカラー ▶ ベクトル　▶ 行列 ... を一般化したもの。

添字 D 個によって表現される量を D 階のテンソルという

Tensor とは？

- スカラー: 添字 0 個で値が決まる $\rightarrow$ 0 階のテンソル
- ベクトル: 添字 1 個で値が決まる $\rightarrow$ 1 階のテンソル (`v = [1, 2, 3], v[0] = 1`)
- 行列: 添字 2 個で値が決まる $\rightarrow$ 2 階のテンソル
  (`M = [[1, 2], [3, 4]], M[0][0] = 1`)

⇩ 例えば

`T = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]` は 3 階のテンソル (`T[0][0][0] = 1`)

テンソルの例

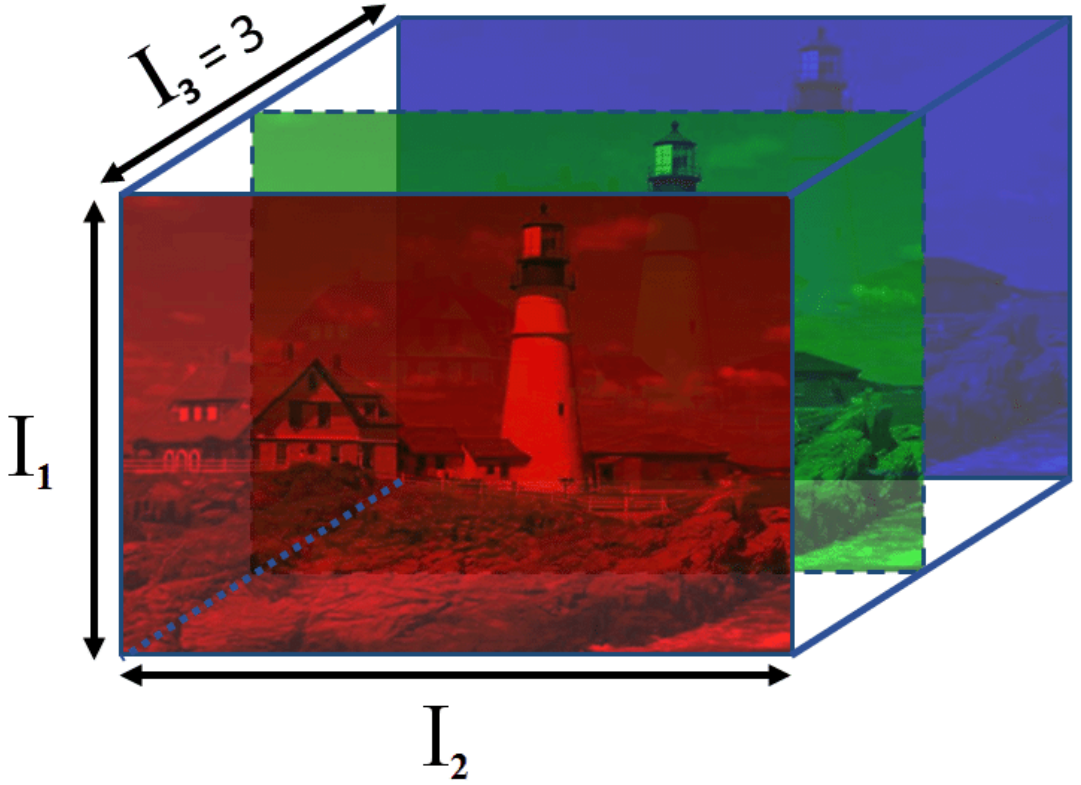
例) RGB 画像は 3 階のテンソル！

$I_{i,j,k} = (i, j)$ **画素の** k **番目の色の強さ**

⇩

n 枚の画像をまとめたものは 4 階のテンソル.

$I_{l,i,j,k} = l$ **番目の画像の** (i, j) **画素の** k **番目の色の強さ**



画像は Quantifying Blur in Color Images using Higher Order Singular Values - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/3rd-order-Tensor-representation-of-a-color-image_fig2_307091456 より

`Tensor` 型のつくりかた

`torch.tensor(data, requires_grad=False)`

- `data`: 保持するデータ(配列**っぽい**ものならなんでも)
  - リスト, タプル, NumPy配列, スカラ, ...
- `requires_grad`: 勾配 (gradient)を保持するかどうかのフラグ
  - デフォルトは `False`
  - 勾配の計算(自動微分)を行う場合は `True` にする
  - このあとこいつを微分の計算に使いますよ〜という表明

Tensor 型

```
>>> x = torch.tensor(2.0, requires_grad=True)
```

2.0 というスカラを保持する `Tensor` 型のオブジェクトを作成

```
>>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
```

$(1.0, 2.0, 3.0)$ というベクトルを保持する `Tensor` 型のオブジェクトを作成

かつては自動微分には `Variable` という名前の型が使われていて, (現在は `Tensor` 型に統合) `Tensor` と数学の変数の概念にある程度の対応があることがわかります.

`Tensor` 型

```
>>> x = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], requires_grad=True)
```

$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$ という行列を保持する `Tensor` 型のオブジェクトを作成

(`requires_grad=True` とすれば, 勾配計算が可能な `Tensor` 型を作成できる)

演習1

これらを勾配計算が可能な`Tensor`型として表現してください.

1. $x = 3.0$
2. $\vec{x} = (3.0, 4.0, 5.0)$
3. $X = \begin{pmatrix} 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \end{pmatrix}$

(このページの内容は, 実際にやらなくてもやり方がわかればOKです)

↓ 問題の続き次のページへ

演習1

(実際にやってください)

4. **整数** $x = 3$ を勾配計算が可能な`Tensor`型として表現することを試みてください.
   また,その結果を確認して説明できるようにしてください.

※ 次のページにヒントあり

演習1 ヒント

1, 2, 3: 講義資料を遡って, `torch.tensor` の第一引数と作成される `Tensor` 型の対応を見比べてみましょう.

4: Pythonのエラーは,

```
~~たくさん書いてある~
~~Error: {ここにエラーの端的な内容が書いてある}
```

という形式です."~~Error"というところのすぐ後に書いてある内容を読んでみましょう.

演習1 解答

1~3.

```
# 1
x = torch.tensor(3.0, requires_grad=True)
# 2
x = torch.tensor([3.0, 4.0, 5.0], requires_grad=True)
# 3
x = torch.tensor([[3.0, 4.0, 5.0], [6.0, 7.0, 8.0]], requires_grad=True)
```

次のページへ

演習1: 解答

4.

```
x = torch.tensor(3, requires_grad=True)
```

としてみると

> RuntimeError: Only Tensors of floating point and complex dtype can require gradients

と出力されます. これは「勾配が計算可能なのは浮動小数点数型と複素数型を格納する `Tensor` のみである」 という PyTorch の仕様によるエラーです.

`Tensor` 型に対する演算

`Tensor` 型は 「数」なので当然各種演算が可能

```
x = torch.tensor(2.0, requires_grad=True)
```

例) 四則演算

```
x + 2
# -> tensor(4., grad_fn=<AddBackward0>)
```

```
x * 2
# -> tensor(4., grad_fn=<MulBackward0>)
```

Tensor 型に対する演算

平方根を取ったり sin や exp を計算することも可能

```
torch.sqrt(x)
# → tensor(1.4142, grad_fn=<SqrtBackward0>)
```

```
torch.sin(x)
# → tensor(0.9093, grad_fn=<SinBackward0>)
```

```
torch.exp(x)
# → tensor(7.3891, grad_fn=<ExpBackward0>)
```

PyTorch と 自動微分

ここまでの内容は別にPyTorchを使わなくてもできること
PyTorchは **計算と共に勾配の計算ができる！**

抑えてほしいポイント:

> **`requires_grad=True` である `Tensor` 型に対して計算を行うと行われた演算が記録された `Tensor` ができる.**

PyTorch と 自動微分

```
x = torch.tensor(2.0, requires_grad=True)
```

足し算をする.

```
y = x + 2
```

PyTorch と 自動微分

```
print(y)
```

これの出力は,

`tensor(4., grad_fn=<AddBackward0>)`

⇩

「`Add` という演算によって作られた」という情報を `y` が持っている！

PyTorch と 自動微分

普通の Pythonの数値では,

```
x = 2
y = x + 2
print(y) # -> 4
```

`y` がどこから来たのかはわからない (値として 4 を持っている **だけで、他にはない**)

PyTorch と 自動微分

PyTorch のしている仕事

1. 演算を記録してくれる

⇩

```
: x = torch.tensor(2.0, requires_grad=True)
```

```
: y = torch.log(x)
  y.grad_fn
```

```
: <LogBackward0 at 0x174655660>
```

PyTorch と 自動微分

✅ PyTorchは `backward` 関数をつかって

記録された演算を 辿る ことで 勾配を計算できる

`backward` による勾配計算

1. `Tensor` 型のオブジェクトをつくる

```
x = torch.tensor(2.0, requires_grad=True)
```

2. 計算を行う

```
y = x + 2
```

3. `backward` メソッドを呼ぶ

```
y.backward()
```

すると...

`backward` による勾配計算

✅ `x.grad` に計算された勾配が格納される！！

```
print(x.grad) # -> tensor(1.)
```

$$\left(y = x + 2 \text{ の } \frac{dy}{dx}\Big|_{x=2} = 1 \text{ が計算されている} \right)$$

PyTorch と 自動微分

PyTorch のしている仕事

1. 演算を記録してくれる

⇩

2. 記録された演算を辿って
勾配を計算する

```
: x = torch.tensor(2.0, requires_grad=True)
: y = x + 2
: y.backward()
: x.grad
: tensor(1.)
```

自動微分の流れ

1. 変数 (`Tensor` 型)の定義
2. 計算
3. backward()

```
# 1. 変数(`Tensor` 型)の定義
x = torch.tensor(2.0, requires_grad=True)
# 2. 計算
y = x + 2
# 3. backward()
y.backward()
```

すると `x.grad` に計算された勾配が格納される.

演習2：100回唱えよう！

定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義

ありとあらゆる演算が自動微分可能

例1) $f(x) = \sin((x+2) + (1 + e^{x^2}))$　の微分

```
x = torch.tensor(2.0, requires_grad=True)
y = y = torch.sin((x + 2) + (1 + torch.exp(x ** 2)))
y.backward()
print(x.grad()) # -> tensor(-218.4625)
```

例2) $y = x^2, z = 2y + 3$ の微分($\frac{dz}{dx}$)

```
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2
z = 2 * y + 3
z.backward()
print(x.grad) # -> tensor(8.)  ... backward()した変数に対する勾配!(この場合はz)
```

ベクトル, 行列演算の勾配

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = 2 * x[0] + 3 * x[1] + 4 * x[2]
y.backward()
print(x.grad) # → tensor([2., 3., 4.])
```

$$\vec{x} = (x_1, x_2, x_3)^\mathsf{T}$$

$$y = 2x_1 + 3x_2 + 4x_3$$

$$\frac{dy}{d\vec{x}} = \left(\frac{dy}{dx_1}, \frac{dy}{dx_2}, \frac{dy}{dx_3} \right)^\mathsf{T} = (2, 3, 4)^\mathsf{T}$$

と対応

ベクトル, 行列演算の勾配

```
A = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], requires_grad=True)
y = torch.sum(A)
y.backward()
print(A.grad) # -> tensor([[1., 1., 1.],
              #          [1., 1., 1.]])
```

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \ y = \sum_{i=1}^{2} \sum_{j=1}^{3} a_{ij} = 21$$

$$\frac{dy}{dA} = \begin{pmatrix} \frac{dy}{da_{11}} & \frac{dy}{da_{12}} & \frac{dy}{da_{13}} \\ \frac{dy}{da_{21}} & \frac{dy}{da_{22}} & \frac{dy}{da_{23}} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

と対応

多変数関数の微分

```
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)
z = 2 * x + 4 * y
z.backward()
print(x.grad) # -> tensor(2.)
print(y.grad) # -> tensor(4.)
```

$$z = 2x + 4y$$

$$\frac{\partial z}{\partial x} = 2, \ \frac{\partial z}{\partial y} = 4$$

に対応

注意: 実際に適用される演算さえ微分可能ならOK

```
x = torch.tensor(2.0, requires_grad=True)

def f(x):
    return x + 3
def g(x):
    return torch.sin(x) + torch.cos(x ** 2)

if rand() < 0.5:
    y = f(x)
else:
    y = g(x)
```

✅ 実際に適用される演算は実行してみないとわからない...
が, 適用される演算はどう転んでも微分可能な演算なのでOK！
(if 文があるから, for 文があるから, 自分が定義した関数に渡したから...ということは関係なく, **実際に Tensor に適用される演算のみが問題になる**)

自動微分

抑えてほしいポイント 👀

- 任意の(勾配が定義できる)計算を `Tensor` 型に対して適用すれば常に自動微分可能
- **定義→計算→backward()** の流れ
- ベクトル, 行列など任意の `Tensor` 型について微分可能. 多変数関数の場合も同様
- 「実際に適用される演算」さえ微分可能ならOK

演習3: 自動微分

1. $y = x^2 + 2x + 1$ の $x = 3.0$ における微分係数を求めよ.
   (https://oj.abap34.com/problems/autograd-practice-1)
2. $y = f(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2$ の $x_1 = 1.0,\ x_2 = 2.0,\ x_3 = 3.0$ における勾配を求めよ.
   (https://oj.abap34.com/problems/autograd-practice-2)
3. $f(\boldsymbol{x_1}) = \boldsymbol{x}_1^T \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \boldsymbol{x_1}$ の $\boldsymbol{x}_1 = (1.0, 2.0)^T$ における勾配を求めよ.
   (https://oj.abap34.com/problems/autograd-practice-3)

演習3: 解答

1.

```
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2 + 2 * x + 1

y.backward()
gx = x.grad

print(gx.item()) # → 8.0
```

スペースの都合上 `import torch` を省略しています

演習3: 解答

2.

```
import torch

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

y = x1**2 + x2**2 + x3**2

y.backward()

print(x1.grad.item()) # -> 2.0
print(x2.grad.item()) # -> 4.0
print(x3.grad.item()) # -> 6.0
```

演習3: 解答

3.

```
W = torch.tensor([[1.0, 2.0], [2.0, 1.0]])
x1 = torch.tensor([1.0, 2.0], requires_grad=True)

y = torch.matmul(torch.matmul(x1, W), x1)
y.backward()

gx = x1.grad

print(*gx.numpy()) # -> 10.0 8.0
```

思い出すシリーズ: 勾配降下法のPyTorchによる実装

$f(x) = x^2 + e^{-x}$ の勾配降下法による最小値の探索

```
from math import exp

x = 3
lr = 0.0005

# xでの微分係数
def grad(x):
    return 2 * x - exp(-x)

for i in range(10001):
    # 更新式
    x = x - lr * grad(x)
    if i % 1000 == 0:
        print('x_', i, '=', x)
```

勾配降下法のPyTorchによる実装

これまでは,導関数 `grad` を我々が計算しなければいけなかった

➪ 自動微分で置き換えられる！

```
import torch

lr = 0.01
N = 10001
x = torch.tensor(3.0, requires_grad=True)

def f(x):
    return x ** 2 - torch.exp(-x)

for i in range(10001):
    y = f(x)
    y.backward()
    x.data = x.data - lr * x.grad
    x.grad.zero_()
```

今ならこれを倒せるはず

最小化してください.

$$-\frac{1}{(x^2+1)}\log\left(\frac{1}{1+e^{-x}}+1\right)$$

https://oj.abap34.com/problems/minimize-difficult-function

おまけ: 自動微分のアルゴリズム

どうやって PyTorch は微分を計算しているのか？🧐

おまけ: 自動微分のアルゴリズム

👦 < 微分係数を計算してください！

⇩

[いちばん素直な方法]

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

を、小さい値で近似する 👉

```
def diff(f, x):
    h = 1e-6
    return (f(x + h) - f(x)) / h
```

勾配の計算法を考える ~近似編

これでもそれなりに近い値を得られる.

例) $f(x) = x^2$ の $x = 2$ における微分係数 4 を求める.

```
>>> def diff(f, x):
...     h = 1e-6
...     return (f(x + h) - f(x)) / h
...
>>> diff(lambda x : x**2, 2)
4.0000010006480125  # だいたいあってる
```

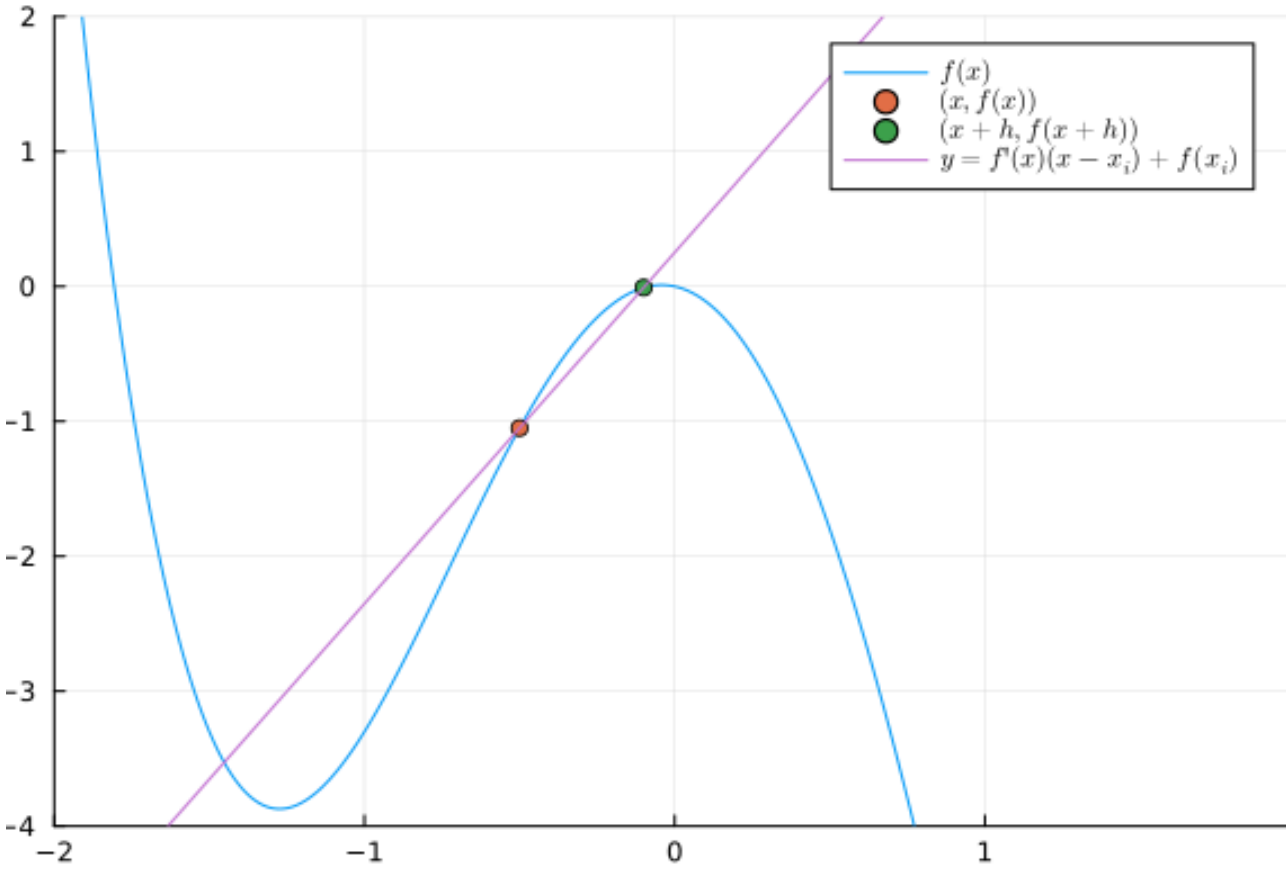
数値微分

実際に小さい h をとって近似する

「数値微分」

お手軽だけど...

- 誤差が出る
- 勾配ベクトルの計算が非効率



数値微分

問題点①. 誤差が出る

1. 本来極限をとるのに小さい h を
   とって計算しているので誤差が出る
2. 分子が極めて近い値同士の引き算に
   なっていて $\left(\frac{f(x+h)-f(x)}{h}\right)$
   桁落ちによって精度が大幅に悪化.

問題点②. 勾配ベクトルの計算が非効率

1. n 変数関数の勾配ベクトル
   $\nabla f(\boldsymbol{x}) \in \mathbb{R}^n$ を計算するには,
   各 x_i について「少し動かす→計算」
   を繰り返すので n 回 f を評価する.
2. 応用では n がとても大きくなり,
   f の評価が重くなりがちなので
   これが **致命的**

数式の構造を捉える

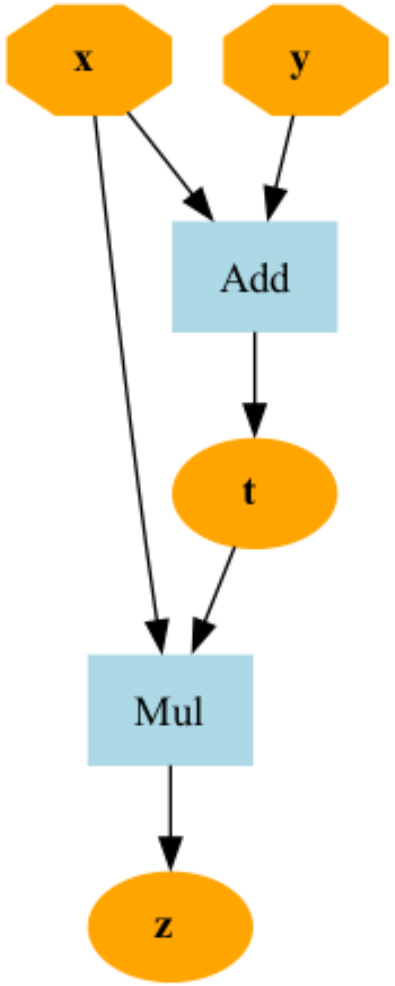
⇩

いい感じに数式の構造をとって計算したい

計算グラフ

✅ **演算は 計算グラフ とよばれる DAG で表現できる**

$t = x + y,\ z = x \times t$ の計算グラフ👉



単に計算過程を表しただけのものを Kantorovich グラフなどと呼び,
これに偏導関数などの情報を加えたものを計算グラフと呼ぶような定義もあります.
(伊里, 久保田 (1998) に詳しく形式的な定義があります)
ただ, 単に計算グラフというだけで計算過程を表現するグラフを指すという用法はかなり普及していて一般的と思われます.そのためここでもそれに従って計算過程を表現するグラフを計算グラフと呼びます.

計算グラフ

✅ PyTorch も 計算と同時 に 計算グラフを構築

(`torchviz` というライブラリを使うと可視化できる！)

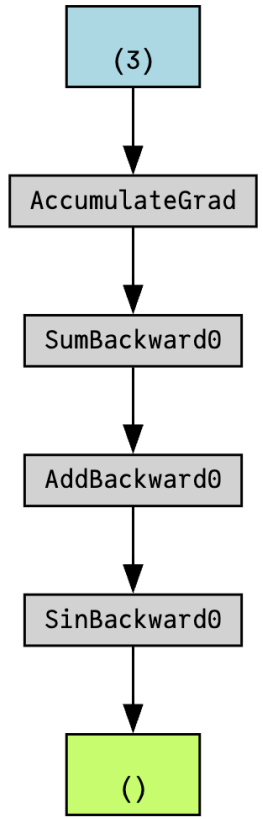
```
import torchviz
x = torch.tensor([1., 2., 3.], requires_grad=True)
y = torch.sin(torch.sum(x) + 2)
torchviz.make_dot(y)
```

PyTorch のように計算と同時に計算グラフを構築する仕組みを **define-by-run** と呼びます. これに対して計算前に計算グラフを構築する方法を **define-and-run** と呼びます. かつての TensorFlow などはこの方式でしたが, 現在では **define-by-run** が主流です.「適用される演算のみが問題になる」という節からわかるように, この方法だと制御構文などを気にせず柔軟な計算グラフの構築が可能になるからです. 一方で、静的に計算グラフを作るのはパフォーマンスの最適化の観点からは非常にやりやすいというメリットもあります.

[1]:
```
import torch
import torchviz

x = torch.tensor([1., 2., 3.], requires_grad=True)
y = torch.sin(torch.sum(x) + 2)
torchviz.make_dot(y)
```

[1]:



計算グラフによる表現

(一旦計算グラフを得たものとして)
この構造から導関数を得ることを考えてみる.

連鎖律

[連鎖律]

u, v の関数 x, y による合成関数 $z\left(x(u,v), y(u,v)\right)$ に対して,

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$

$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial v} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial v}$$

連鎖律と計算グラフの対応

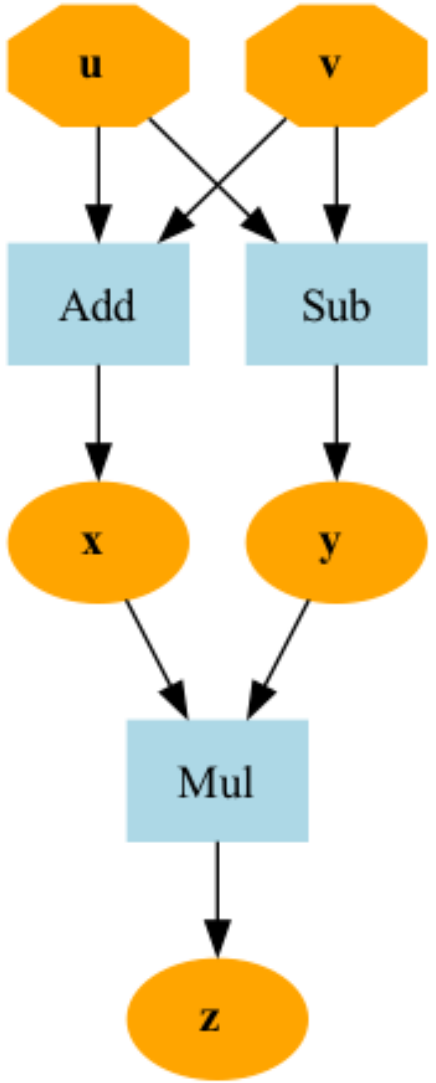
目標

$x = u + v$

$y = u - v$

$z = x \cdot y$

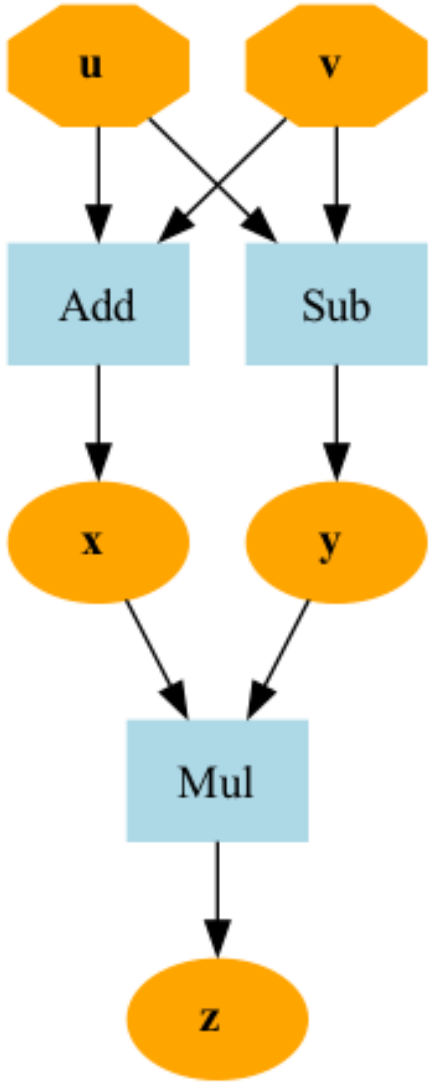
のとき, $\dfrac{\partial z}{\partial u}$ を求める



連鎖律と計算グラフの対応

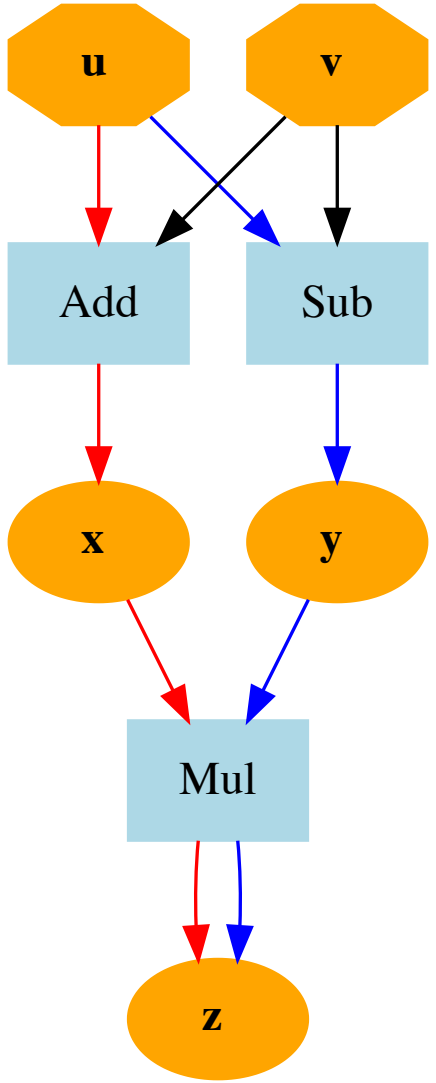
$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$

との対応は



連鎖律と計算グラフの対応

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$



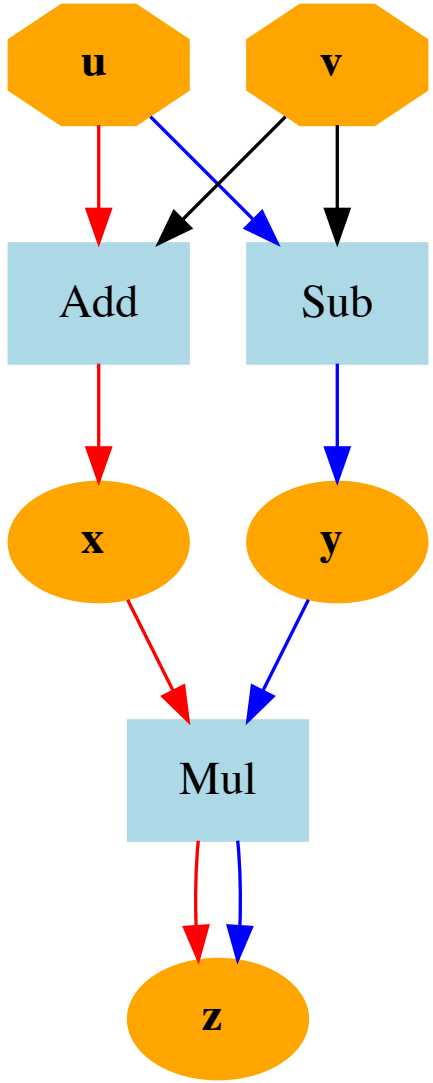
連鎖律と計算グラフの対応

✅ 変数 z に対する u による偏微分の
計算グラフ上の表現

$\leftrightarrow$ u **から** z **への全ての経路の偏微分の総積の総和**

$$\frac{\partial z}{\partial u} = \sum_{p \in \hat{P}(u,z)} \left(\prod_{(s,t) \in p} \frac{\partial t}{\partial s} \right)$$

$\hat{P}(u,z)$ は u から z への全ての経路の集合. (s,t) は変数 s から変数 t への辺を表す.



連鎖律と計算グラフの対応

演算を **基本的な演算の合成に分解** すれば、$\frac{\partial t}{\partial s}$ は事前に網羅できる！

⇨ **全体の勾配が求まる** 🙌

$$\frac{\partial z}{\partial u} = \sum_{p \in \hat{P}(u,z)} \left(\prod_{(s,t) \in p} \frac{\partial t}{\partial s} \right)$$

OO を使った典型的な自動微分の実装

1. **基本的な演算**　を用意しておく.

```
class Add:
    def __call__(self, x0: Tensor, x1: Tensor) -> Tensor:
        self.x0 = x0
        self.x1 = x1
        return Tensor(x0.value + x1.value, creator=self)

    def backward(self, gy):
        return gy, gy

class Mul:
    def __call__(self, x0: Tensor, x1: Tensor) -> Tensor:
        self.x0 = x0
        self.x1 = x1
        return Tensor(x0.value * x1.value, creator=self)

    def backward(self, gy):
        return gy * self.x1, gy * self.x0
```

OO を使った典型的な自動微分の実装

1. **変数を表すオブジェクト**を用意しておき、これの基本的な演算をオーバーライドする.

```
class Tensor:
    def __init__(self, value):
        ...

    def __add__(self, other):
        return Add()(self, other)

    def __mul__(self, other):
        return Mul()(self, other)
```

連鎖律と計算グラフの対応

✅ **実は工夫するとノード数の定数倍で勾配を計算可能！**

詳しくは Julia Tokyo #11 トーク: 「Juliaで歩く自動微分」 をみよう！

PyTorch でもこの方法で勾配を計算している.

機械学習講習会 第四回

-「ニューラルネットワークの構造」

traP Kaggle班

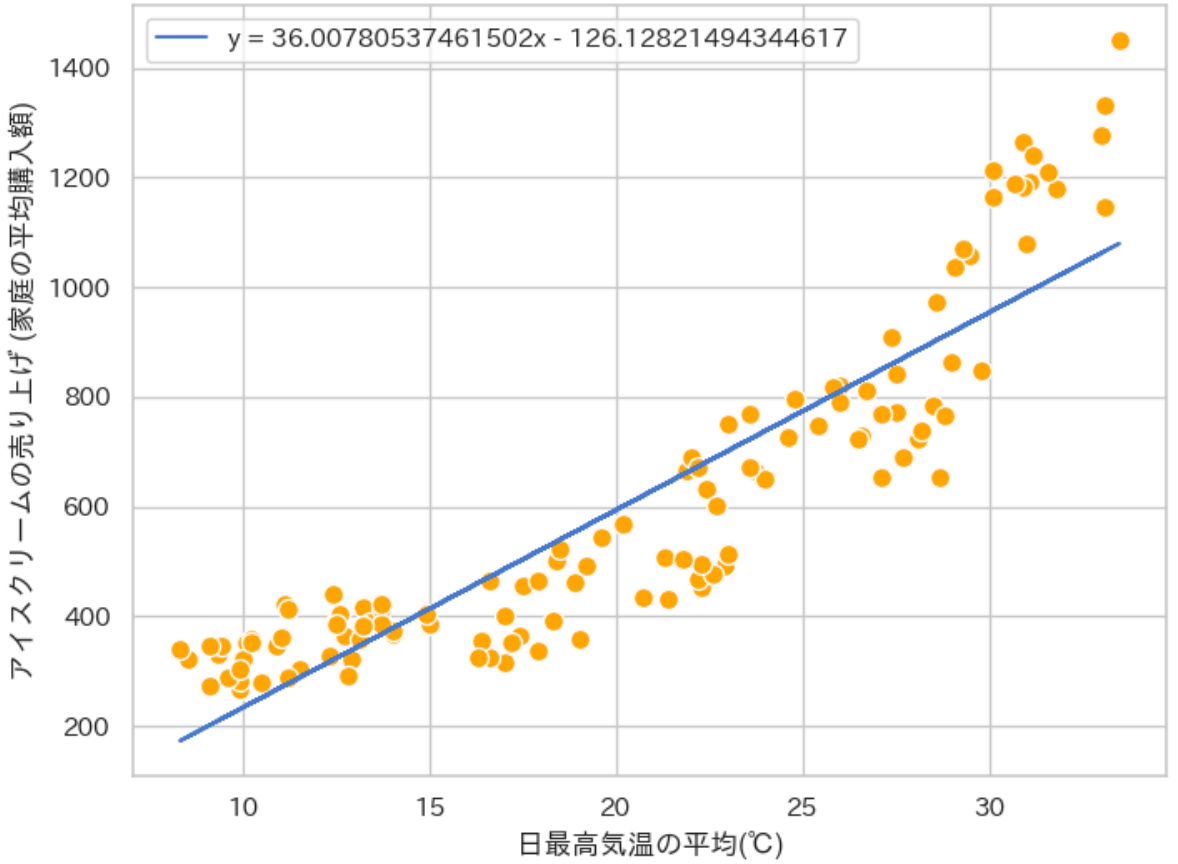
2024/07/01

振り返りタイム

第一回 「学習」
第二回 「勾配降下法」
第三回 「自動微分」

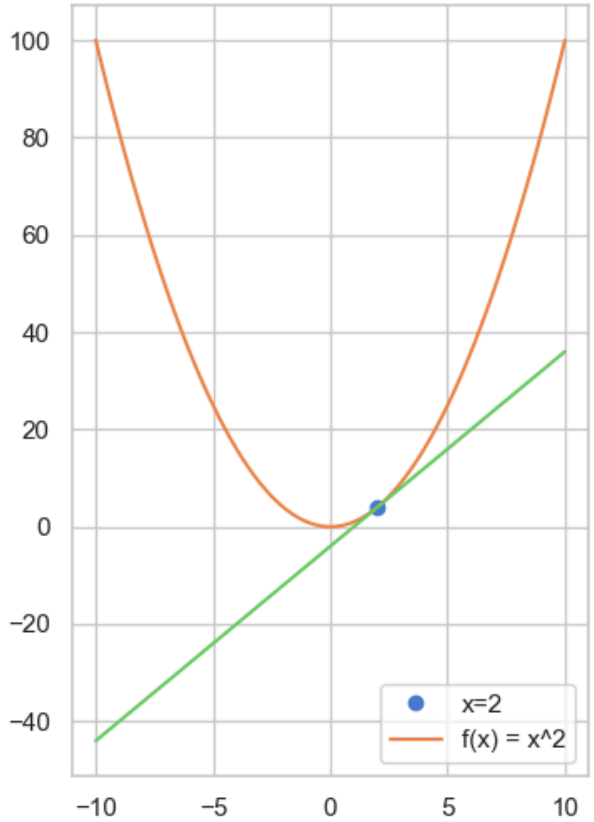
「学習」

1. 予測をするには「モデル」を作る必要があった
2. モデルのパラメータを決めるためにパラメータの関数である損失関数を導入した



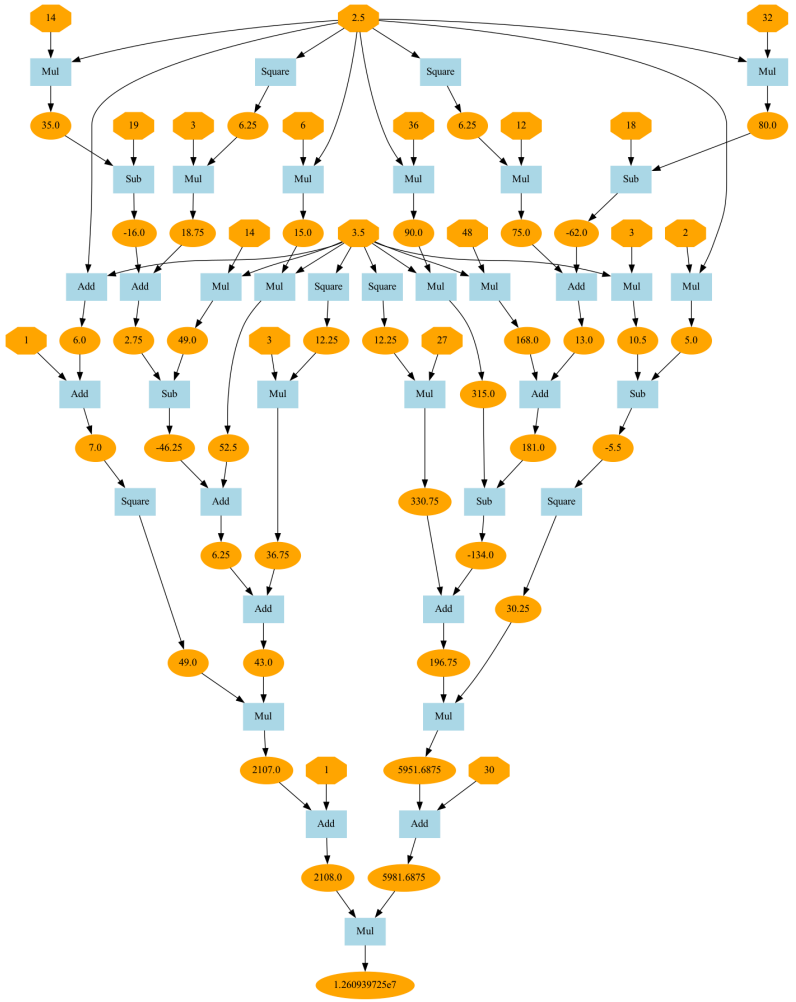
「勾配降下法」

1. 複雑になりうる損失関数を最小にするために **「勾配降下法」** を使ってパラメータを探索した



「自動微分」

1. 自動微分を使うことで, 手で微分をしなくても勾配を得て勾配降下法を適用できるようになった



振り返りタイム

1. 予測をするには「モデル」を作る必要があった
2. モデルのパラメータを決めるために, パラメータの関数である損失関数を導入した
3. 損失関数を最小にするパラメータを求めるために勾配降下法を導入した
4. 自動微分によって手で微分する必要がなくなった [← 今ココ！]

第三回までのまとめ

われわれができるようになったこと

データさえあれば...誤差を小さくするパラメータを

- 例え複雑な式でも
- 例え自分で導関数を見つけられなくても

探しにいけるようになった！
(== **学習ができるようになった！**)

線形回帰からの飛躍

ここまでは $f(x) = ax + b$ のかたちを仮定してきた (線形回帰)

⇨ われわれの手法はこの仮定に依存しているか？ 🧐

⇩

依存していない

(ように手法を選んだ！)

線形回帰からの飛躍

我々の手法 (自動微分と勾配降下法による学習) で満たすべき条件だったのは...

$L(a, b)$ が a, b について微分可能である

のみ！

⇨ この条件を満たす関数なら **どんなものでも** 学習できる！

今日のお話は...

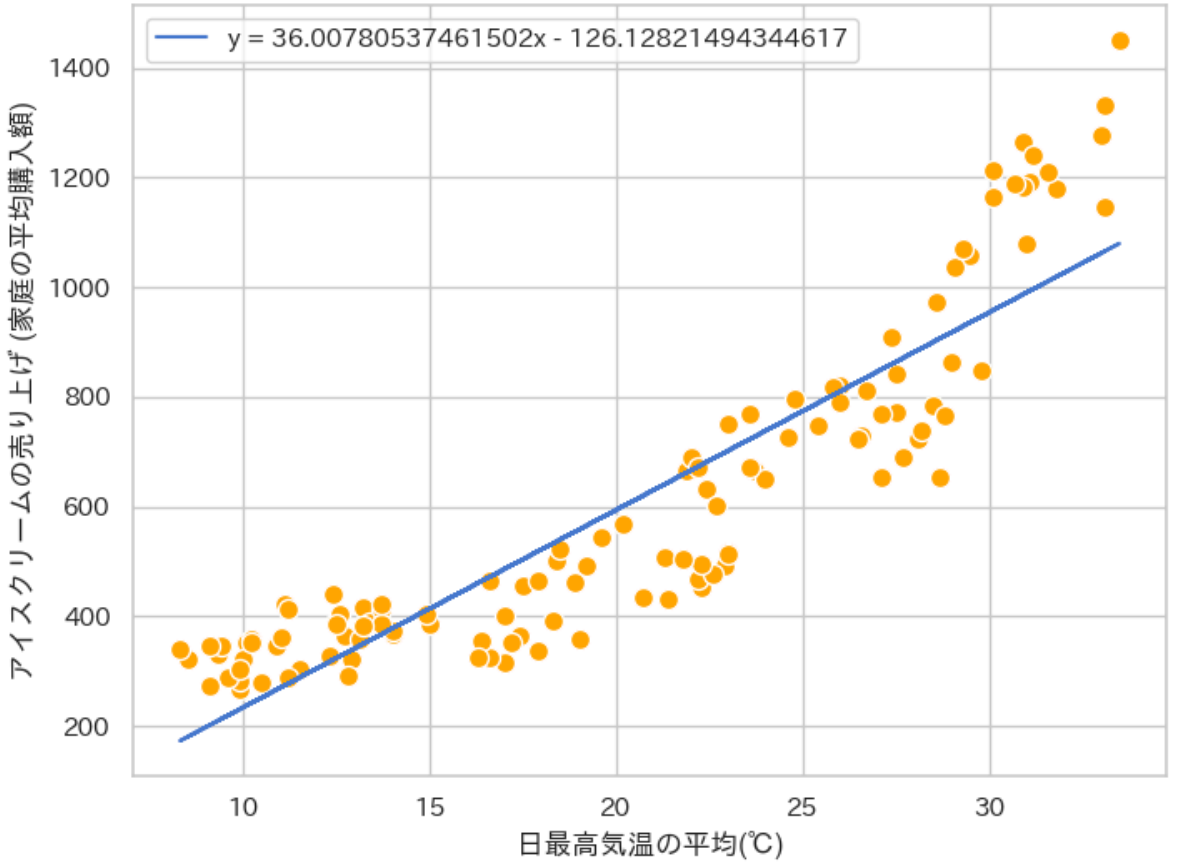
f を変えよう

$$L(a,b) = \sum_{i=0}^{n-1} (y_i - f(x_i))^2$$

線形回帰からの飛躍

$f(x) = ax + b$　は, a, b をどんなに変えても常に直線

⇨ 直線以外の関係を表現できない



どんな関数をつかうべきか?

$f(x) = ax^2 + bx + c$ でも大丈夫

$f(x) = \sin(ax + b)$ でも大丈夫

$f(x) = e^{ax+b}$ でも大丈夫

⇨ 直線以外を表現することはできるが

- 二次曲線
- sinカーブ
- 指数カーブ(?)

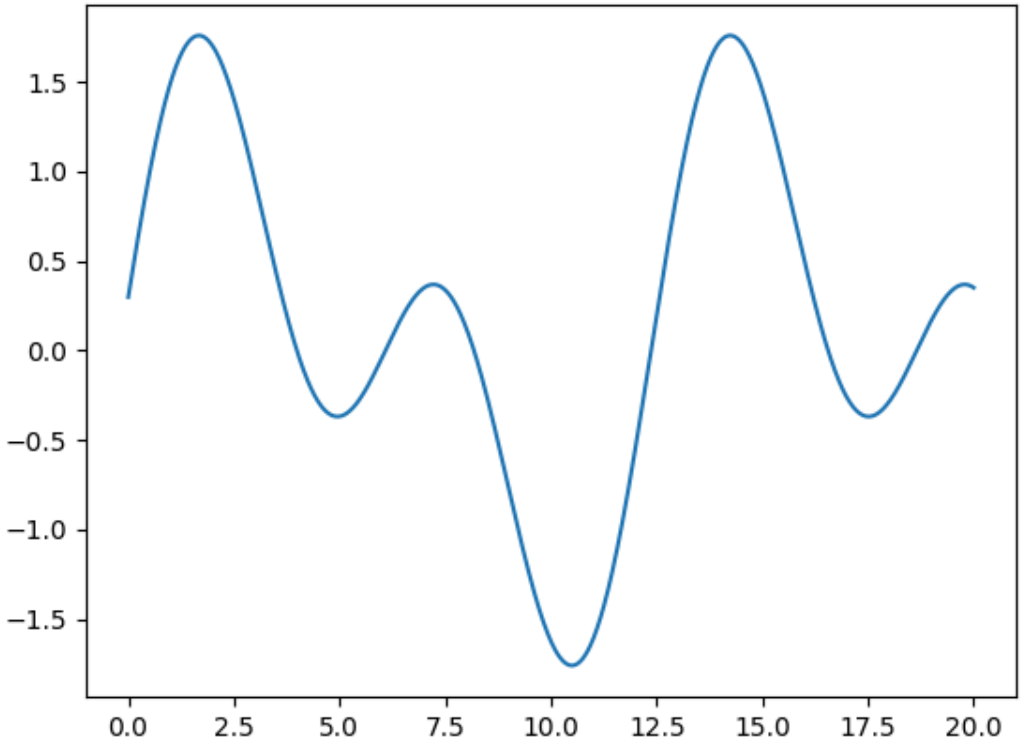
しか表現できない

複雑な関数を表現する方法を考えよう！

これらのパラメータどんなにいじっても

👉

みたいな関数は表現できない

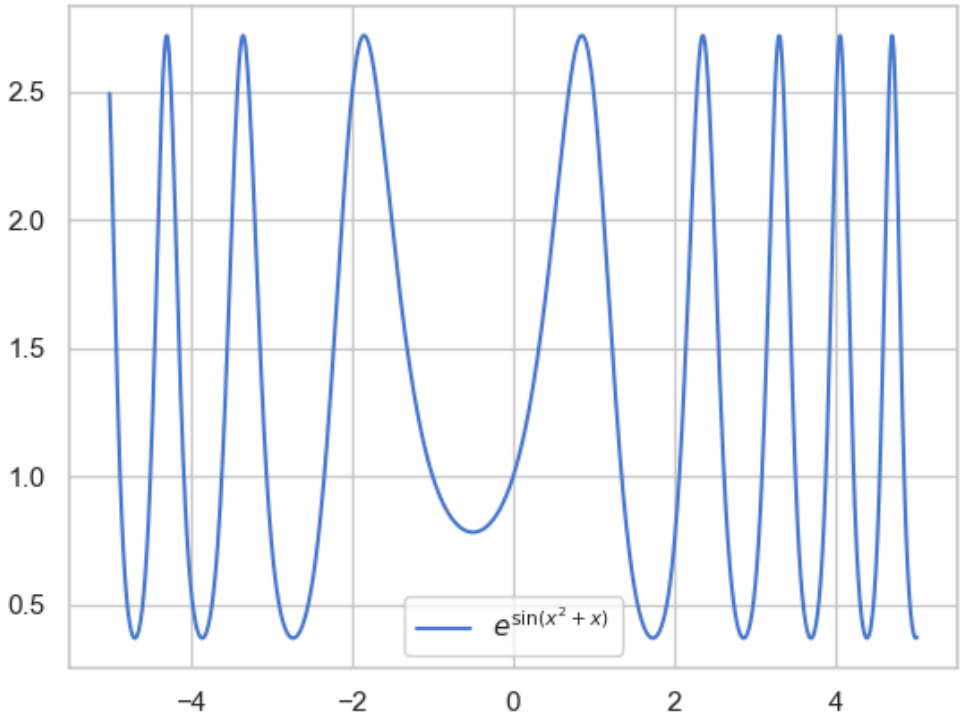


複雑さを生み出す方法

✅ アイデア1: 関数を合成する

$\exp, \sin, x^2 + x$ はそれぞれ非線形単純な関数

一方, 合成した
$h(x) = \exp(\sin(x^2 + x))$ は 👉



非線形でなくてはいけないことに注意してください！
$f_i(x) = a_i x + b_i$ は、 $f_1(f_2(f_3(\cdots f_n(x)\cdots)))$ が
$a_1(a_2(a_3(\cdots a_n x + b_n \cdots))) + b_1$ となって結局 $ax + b$ の形になってしまいます。

複雑さを生み出す方法

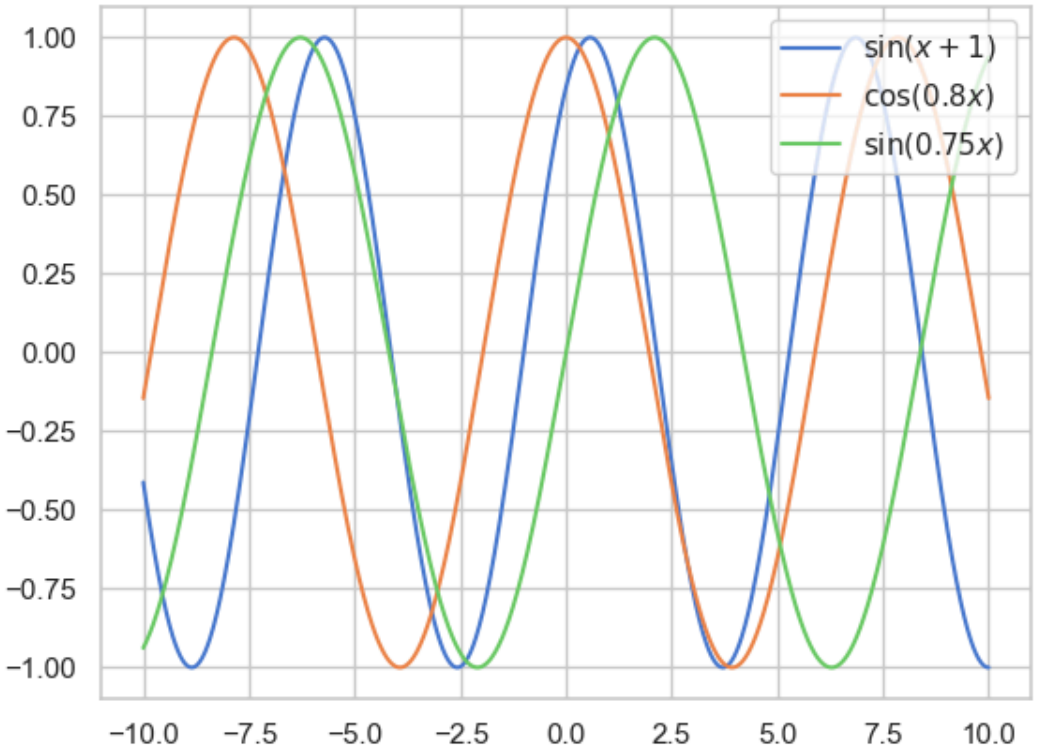
✅ **アイデア2: 和をとる**

複雑さを生み出す方法

三角関数を 3つ用意

- $f_1(x) = \sin(0.5x)$
- $f_2(x) = \cos(0.8x)$
- $f_3(x) = \sin(0.75x)$

✔ それぞれは単純.

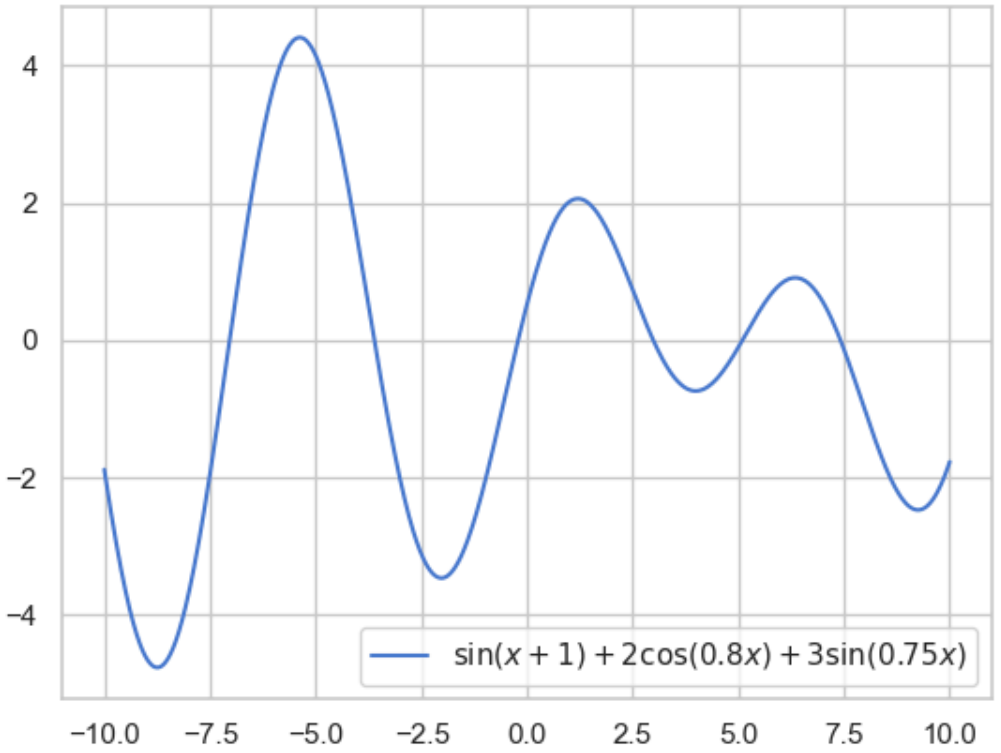


複雑さを生み出す方法

一方, 重み付き和をとると

$$f(x) = 3f_1(x) - 2f_2(x) + f_3(x)$$

そこそこ複雑になっている 👉



ぐにゃっとした関数の表現のしかた

✅ 簡単めの非線形関数の

1. 合成

2. 和

を考えたら結構複雑なやつも表現できる

パラメータを変えることによって幅広い表現が得られる確認

パラメータとして

$\boldsymbol{a} = (a_1, a_2, a_3, a_4, a_5)$,

$\boldsymbol{b} = (b_1, b_2, b_3, b_4, b_5)$,

$\boldsymbol{c} = (c_1, c_2, c_3, c_4, c_5)$

をもつ

$$f(x; \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = \sum_{i=1}^{5} a_i \sin(b_i x + c_i)$$

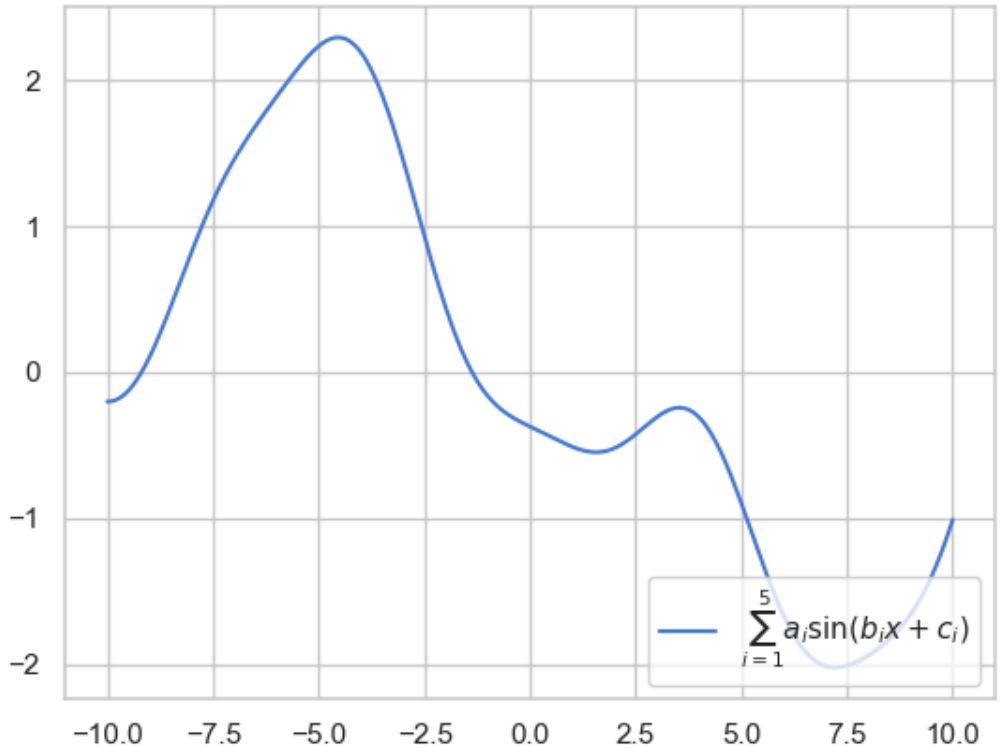
を考える

⇩

パラメータを変えることによって幅広い表現が得られる確認

$$\boldsymbol{a} = (0.83, 0.27, 0.84, 0.28, 0.14)^T$$
$$\boldsymbol{b} = (0.71, 0.47, 0.56, 0.39, 0.94)^T$$
$$\boldsymbol{c} = (0.08, 0.92, 0.16, 0.44, 0.21)^T$$

のとき



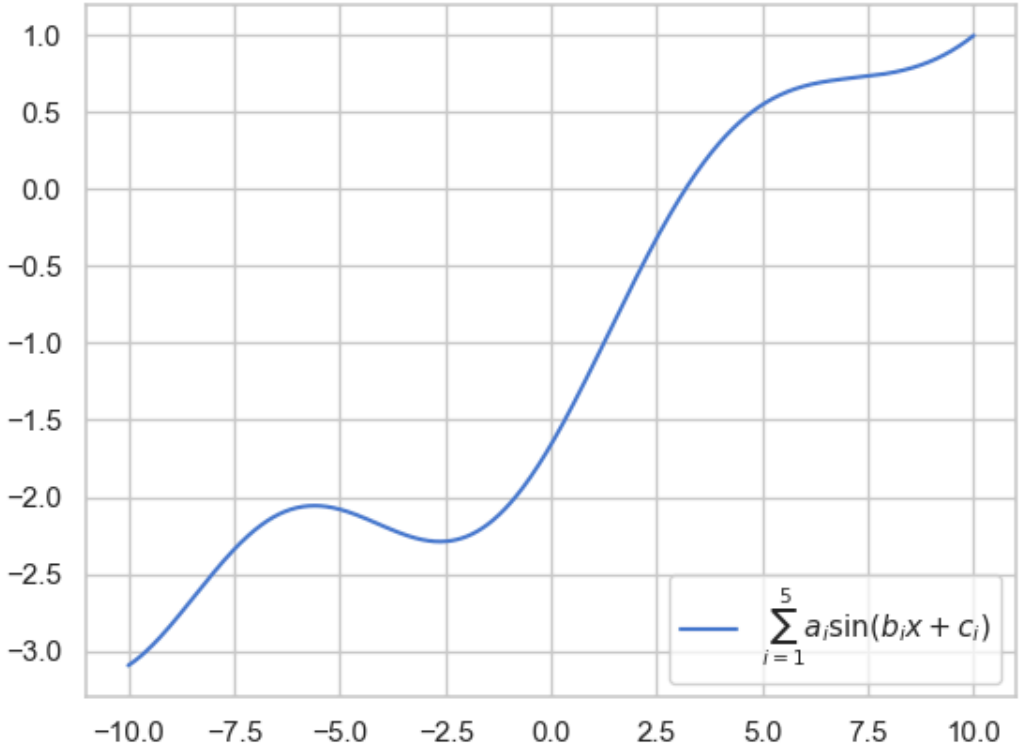
パラメータを変えることによって幅広い表現が得られる確認

$$\boldsymbol{a} = (0.39, -0.29, -0.67, -0.96, 0.92)^T$$

$$\boldsymbol{b} = (-0.35, 0.84, 0.22, -0.25, -0.04)^T$$

$$\boldsymbol{c} = (-0.61, -2.06, 3.97, 0.40, -3.85)^T$$

のとき



「基になる関数」はどう選ぶべきか？

和をとる「**基になる関数**」にどのような関数を選ぶべきか？

- 三角関数?
- 多項式関数?
- 指数関数?
- もっと別の関数?

これまでの我々のアプローチを思い出すと...

変化させるのが可能なところはパラメータにして，学習で求める」

「基になる関数」も学習で求めよう

ニューラルネットワーク

ニューラルネットワーク

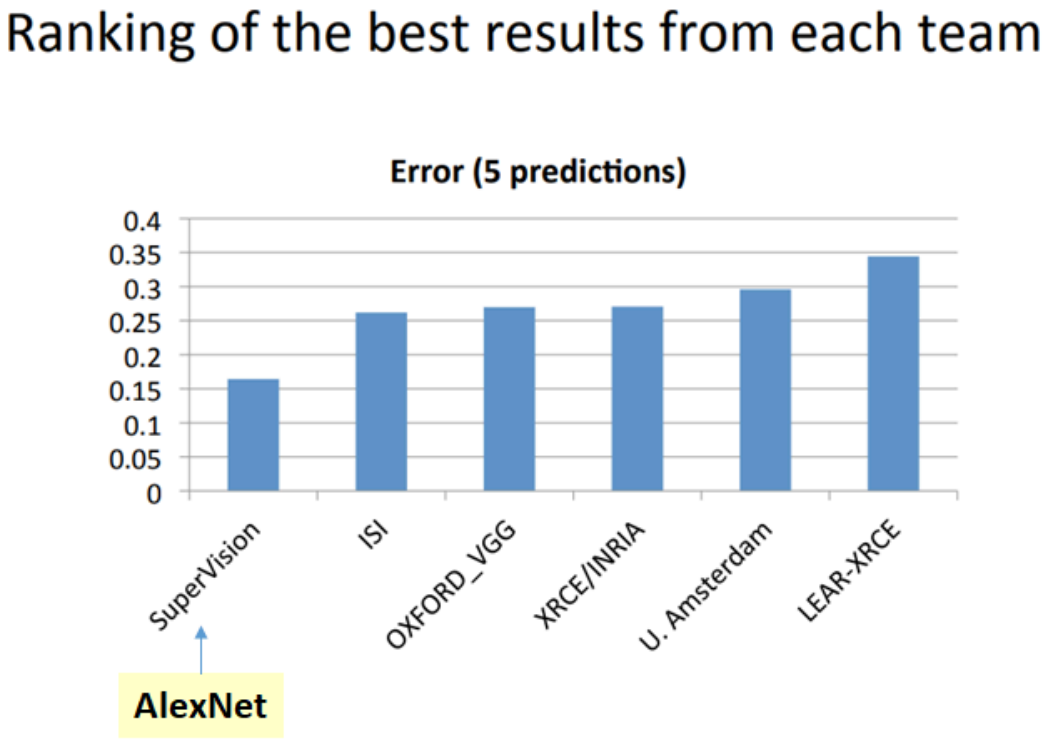
[事実1]

最近流行りの機械学習モデルはたいていニューラルネットワークをつかっている

上の画像は ChatGPT のロゴ. 中央の画像は https://diamond.jp/articles/-/241828 より. Ponanza と佐藤天彦名人の対局. 下の画像は StableDiffusion という画像生成モデルが生成した画像.

ニューラルネットワーク

[事実2]
ある程度以上複雑なタスクではニューラルネットワークが最も優れた性能を示すことが多い



グラフはILSVRC という画像認識の大会でニューラルネットワークを使ったモデル (AlexNet) が登場し, 圧倒的な精度で優勝した際のスコア.
https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160 から.

今日の内容

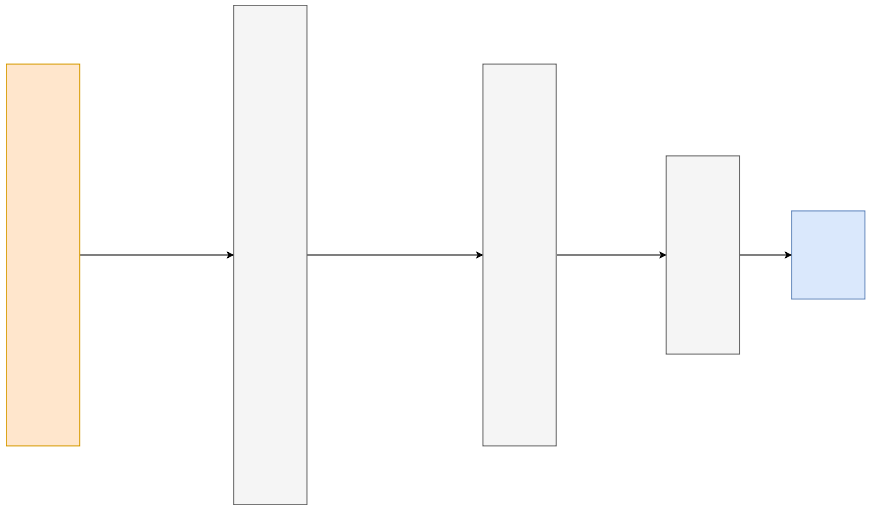
1. ニューラルネットワークの基本的な概念の整理

2. 全結合層の理解

ニューラルネットワークの構造

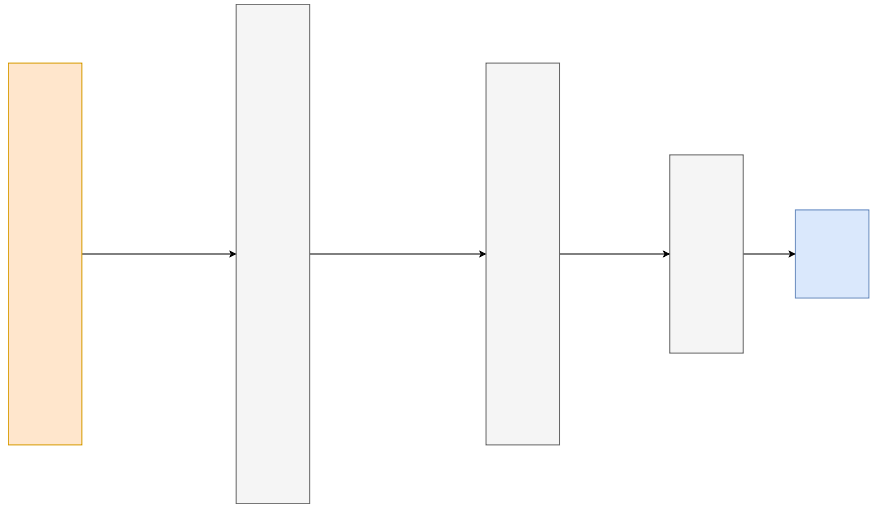
基本単位: レイヤー

ニューラルネットワークは 「レイヤー」と呼ばれる基本的な関数の合成によって構成されるモデル



ニューラルネットワークの構造

- 入力層
  入力を受け取る部分
- 出力層
  出力を出力する部分
- 中間層(隠れ層, hidden layer)
  それ以外



⇩

データの流れは,

x →**入力層→中間層...→出力層** = y

いろいろなレイヤー

PyTorch本体ででデフォルトで定義されているものだけで 160個以上? [1]

[1] `torch.nn.Module` のサブクラスの数を数えました.正確な数でないかもしれません.

全結合層 (Linear, Dense層)

もっとも普遍的・基本のレイヤー

先に全ての情報を書くと....

全結合層 (Linear, Dense層)

パラメータ $W \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$ と

各レイヤーが固有にもつ活性化関数 σ を用いて

入力として $\boldsymbol{x} \in \mathbb{R}^n$ を受け取り, $\sigma\left(W\boldsymbol{x} + \boldsymbol{b}\right)$ を出力する.

全結合層 (Linear, Dense層)

(これでわかったら苦労しないので、一つずつ見ていきます)

全結合層がしていること

1. n 個の入力を受け取り,m 個出力する
2. 複雑な関数を表現するアイデア...

    1. 非線形関数の合成
    2. 和をとる

    をする

全結合層がしていること

1. n 個の入力を受け取り, m 個出力する

パラメータ $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^{m}$ と

各レイヤーが固有にもつ活性化関数 σ を用いて

入力として $\boldsymbol{x} \in \mathbb{R}^{n}$ を受け取り, $\sigma\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$ を出力する.

👆 丁寧に計算の次元を追ってみよう！

合成

演算を d 回繰り返す

(n 次元ベクトル → m_1, → m_2, → $\cdots$, → m_d 次元ベクトルへと変換されながら
計算が進んでいく)

$$\boldsymbol{u}^{(1)} = \sigma\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{u}^{(2)} = \sigma\left(W^{(2)}\boldsymbol{u}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\cdots$$

$$\boldsymbol{u}^{(d)} = \sigma\left(W^{(d)}\boldsymbol{u}^{(d-1)} + \boldsymbol{b}^{(n)}\right)$$

全結合層がしていること

1. 複雑な関数を表現するアイデア...

    1. 非線形関数の合成
    2. 和をとる

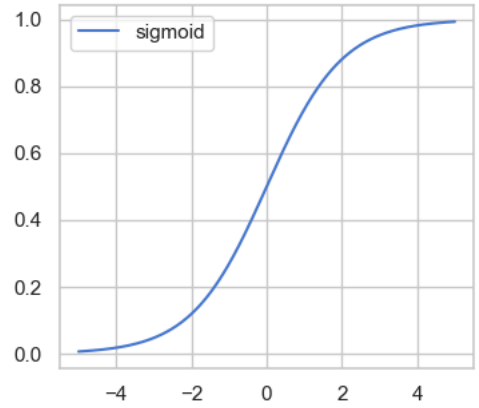
    をする

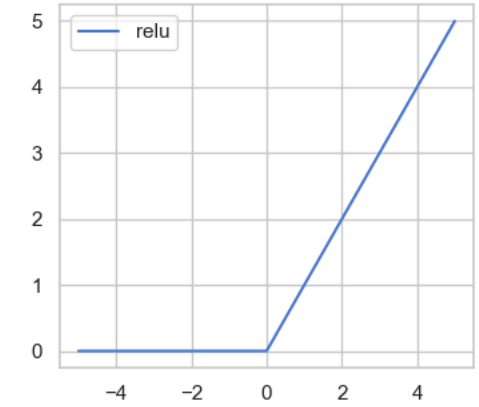
活性化関数とは？

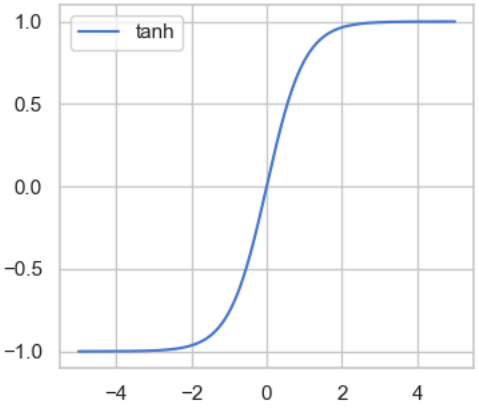
出力前に通す **非線形関数** σ
($\sigma\left(W\boldsymbol{x}+\boldsymbol{b}\right)$)

- シグモイド関数
  $$\sigma(x)=\frac{1}{1+\exp(-x)}$$
- ReLU関数
  $$\mathrm{ReLU}(x)=\max(0,x)$$
- tanh関数
  $$\tanh(x)=\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

など (大量に存在)







なぜ活性化関数が必要か？

✅ **最後に非線形関数を通すことで全結合層が非線形関数になる.**

今できたこと・・・全結合層を非線形にする.

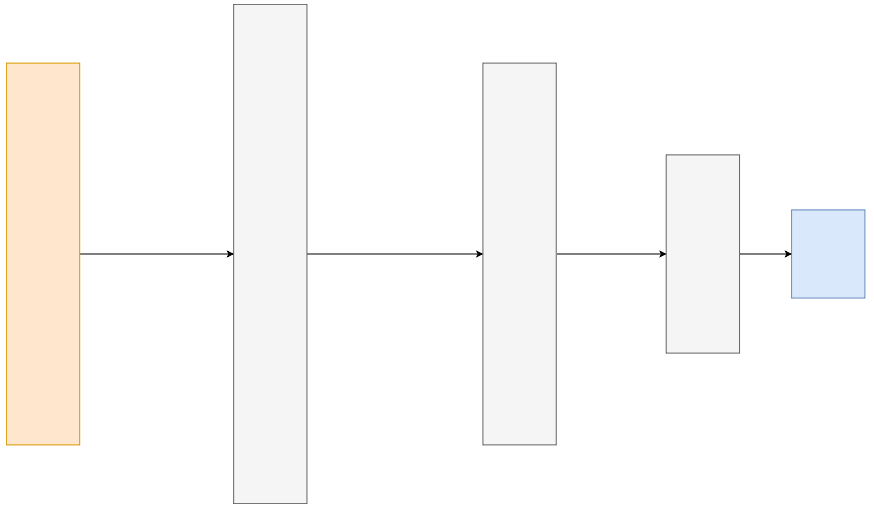
⇩

これを合成している！

＝ **非線形関数の合成**

アイデア1. 合成

非線形関数の合成を繰り返す ⇨ 複雑な関数を表現



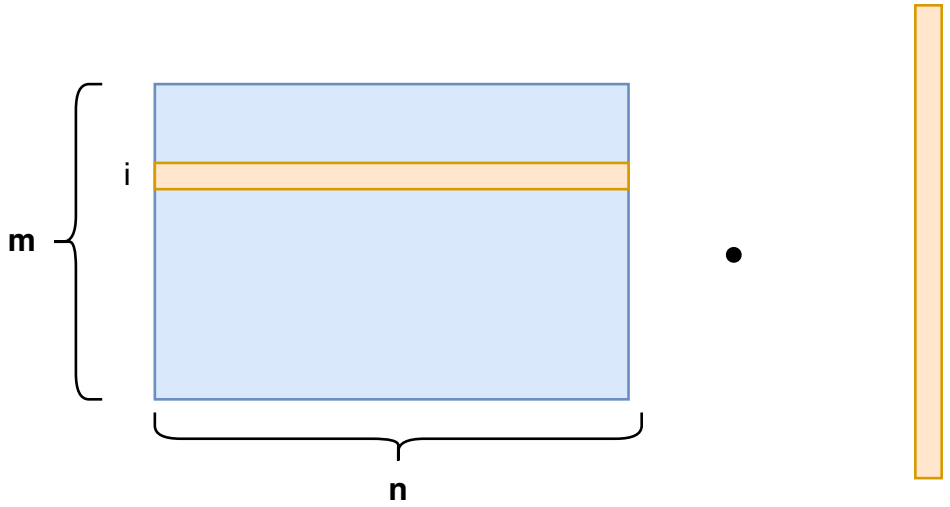
アイデア2. 和をとる

m 個の出力のひとつに注目してみる.

$$\boldsymbol{y} = \sigma\left(W\boldsymbol{x} + \boldsymbol{b}\right)$$

⇩

$$y_i = \sigma\left(\sum_j W_{ij}x_j + b_i\right)$$

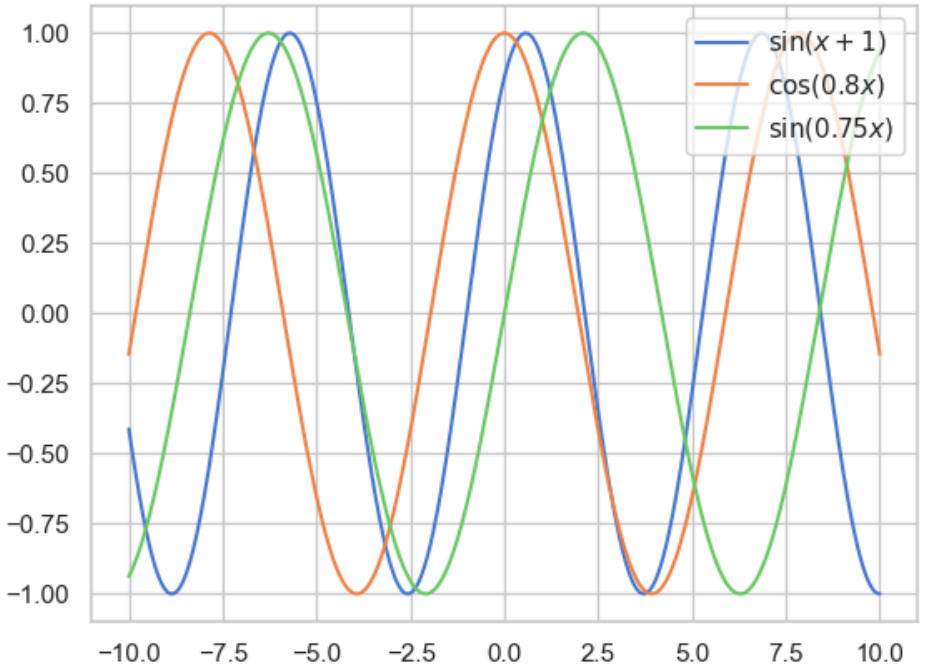


分解して考えると

$$y_i = \sigma\left(\sum_j W_{ij}x_j + b_i\right)$$ は,

非線形関数の和をとる と

同じことをしている！！

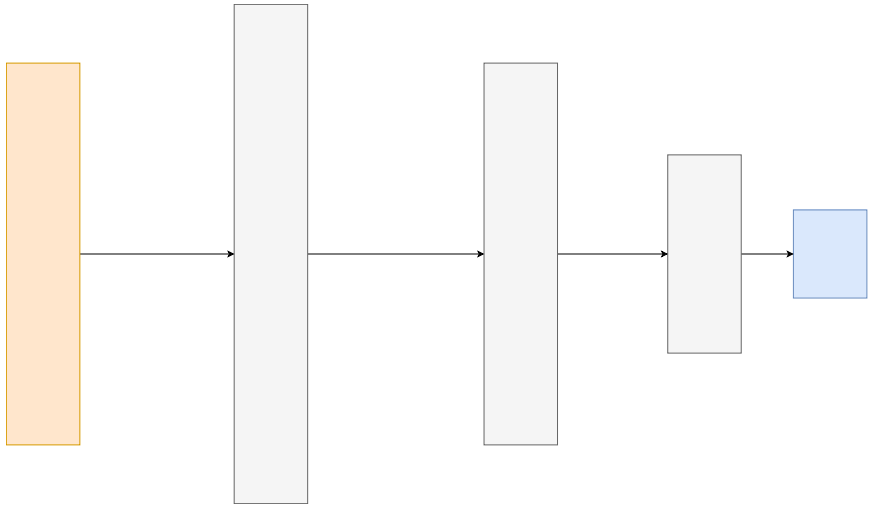


分解して考えると

$$y_i = \sigma\left(\sum_j W_{ij}x_j + b_i\right)$$

各層の入力 x_j はそれまでの層で σ を通ってきたもの！

$\leftrightarrow$ x_j は **非線形**



複雑な関数が生まれていた

$$\sigma\left(\sum_j W_{ij}x_j + b_i\right)$$

⬇

非線形関数の重みつき和

⬇

複雑な非線形関数を表現できる！ + **さらにそれを非線形関数に通す**

出力層

+ 各層で和をとる「基になる関数」は、

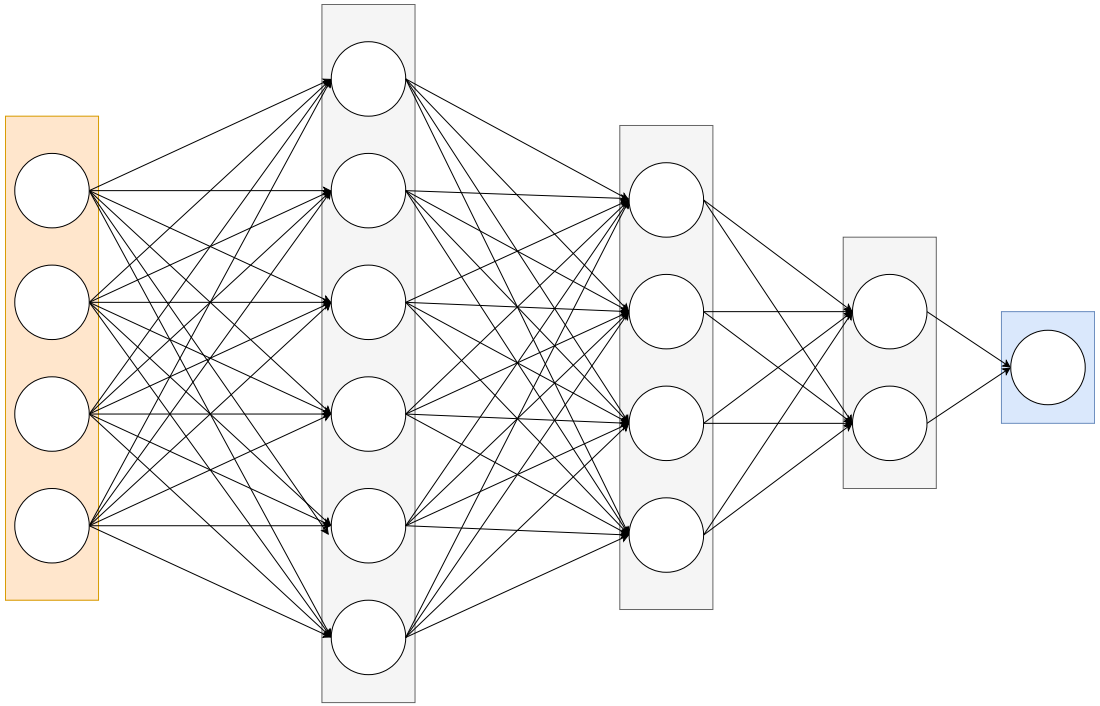
それまでの層のパラメータによって変化する

というわけで

「基になる関数」も学習で求めよう

MLP

とくに 全結合層のみからなるニューラルネットワークを
多層パーセプトロン (Multi Layer Perceptron, MLP) という



そのほかの用語たち

用語	意味
MLP (Multi Layer Perceptron)	全結合層のみからなるニューラルネットワーク
DNN (Deep Neural Network)	複数の隠れ層を持つニューラルネットワーク
ANN (Artificial Neural Network)	人工ニューラルネットワーク.本来の意味のニューラルネットワーク(動物の神経回路) と区別するためこういう名前が使われることがある

ニューラルネットワークの性質

そもそも直線をやめたくなった動機:

👱 < 直線だけしか表現できないのは困る.

👩 < いろいろな関数が表現できるようになりたい.

⇩

どれくらいの関数が表現できるようになったのか？

ニューラルネットワークの万能近似性

結論

直線 ⇨ なんでも ※

※ ざっくりとした表現です.

ニューラルネットワークの万能近似

ニューラルネットワークの万能近似定理 (普遍性定理)

隠れ層を一つ持つニューラルネットワークは,
任意の連続関数を表現できる ※

※ ざっくりとした表現です.

今日のまとめ

- 我々の学習手法は, $f(x) = ax + b$ というモデルの構造自体に直接依存しているわけではなかった
- $f(x) = ax + b$ というモデルの構造では直線しか表現することができないので, 違う形を考えることにした
- 「基になる」簡単な関数の **合成** と **和** を考えることでかなり複雑な関数も表現できることがわかった
- 「基になる」関数の選び方を考える上で, この関数自体もパラメータによって変化させるモデルとしてニューラルネットワークを導入した
- ニューラルネットワークは非常に幅広い関数を表現できることがわかった

発展的話題:万能近似の(直感的な) 説明

- ニューラルネットワークの表現能力は 1980年代後半 ~ 1990年代後半くらいまで盛んに研究
- いろいろな条件でいろいろな結果を得ている
- ここではおそらく最も有名である Cybenko による定理 [1] を紹介する

[1] Cybenko, George. "Approximation by superpositions of a sigmoidal function." Mathematics of control, signals and systems 2.4 (1989): 303-314.

準備

準備

定義1. シグモイド型関数

$$\sigma(x) \to \begin{cases} 0 & (x \to -\infty) \\ 1 & (x \to \infty) \end{cases}$$

を満たす関数を「シグモイド型関数」と呼ぶ.

$I = [0,1]^d$ として,C を I 上の連続関数全体の集合とする.

主張

定理 (Cybenko, 1989)

任意の $f \in C,\ \varepsilon > 0$ に対して,ある $g(x) = \sum_{i=1}^{n} a_i \sigma(b_i x + c_i)$ が存在して

$$\forall x \in I,\ |f(x) - g(x)| < \varepsilon$$

主張

平易に書くと,

どんな連続関数も隠れ層が一つのニューラルネットワークで十分に近似できる

ステップ1. シグモイド型関数をつかった階段関数のつくりかた

$$g(x) = \sum_{i=1}^{n} a_i \sigma(b_i x + c_i)$$

$$\left(\sigma(x) \to \begin{cases} 0 & (x \to -\infty) \\ 1 & (x \to \infty) \end{cases} \right)$$

σ はシグモイド型関数 ⇨ b_i をものすごく大きくするとどうなるか？

証明ステップ1

$b_i = 999999999999999999999999999999$

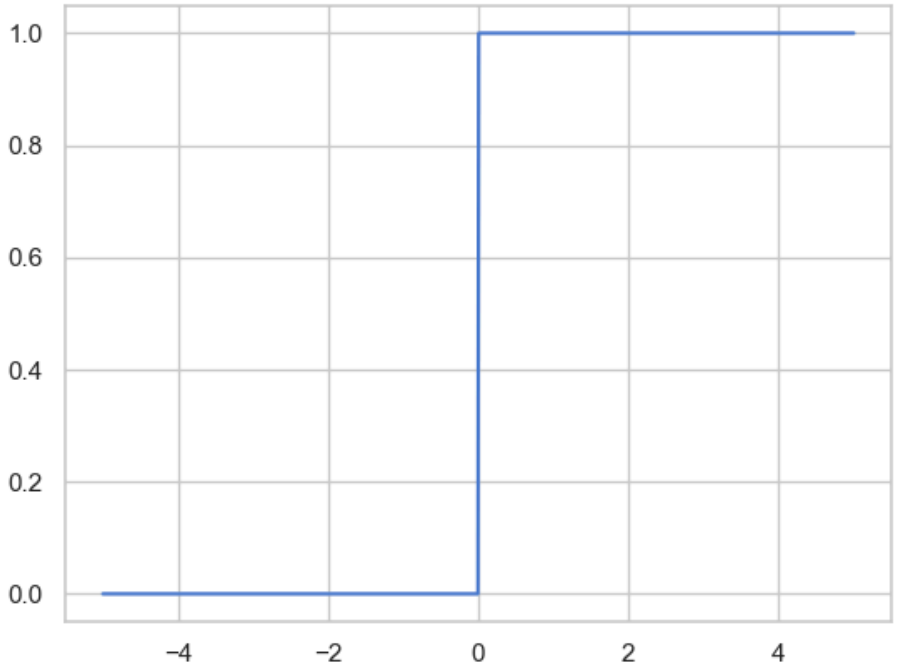
とする.

すると, $x_i - \dfrac{c_i}{b_i}$ が少しでも正なら

$\sigma(b_i x + c_i) = 1$

負なら

$\sigma(b_i x + c_i) = 0.$



証明ステップ1

$\sigma(b_i x + c_i)$ は
$b_i = 99999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999$

とすると $x_i - \dfrac{c_i}{b_i}$ が少しでも正ならば 1, そうでなければ 0 になる.

⇨ c_i を適当に調整すれば, 狙った点 t で

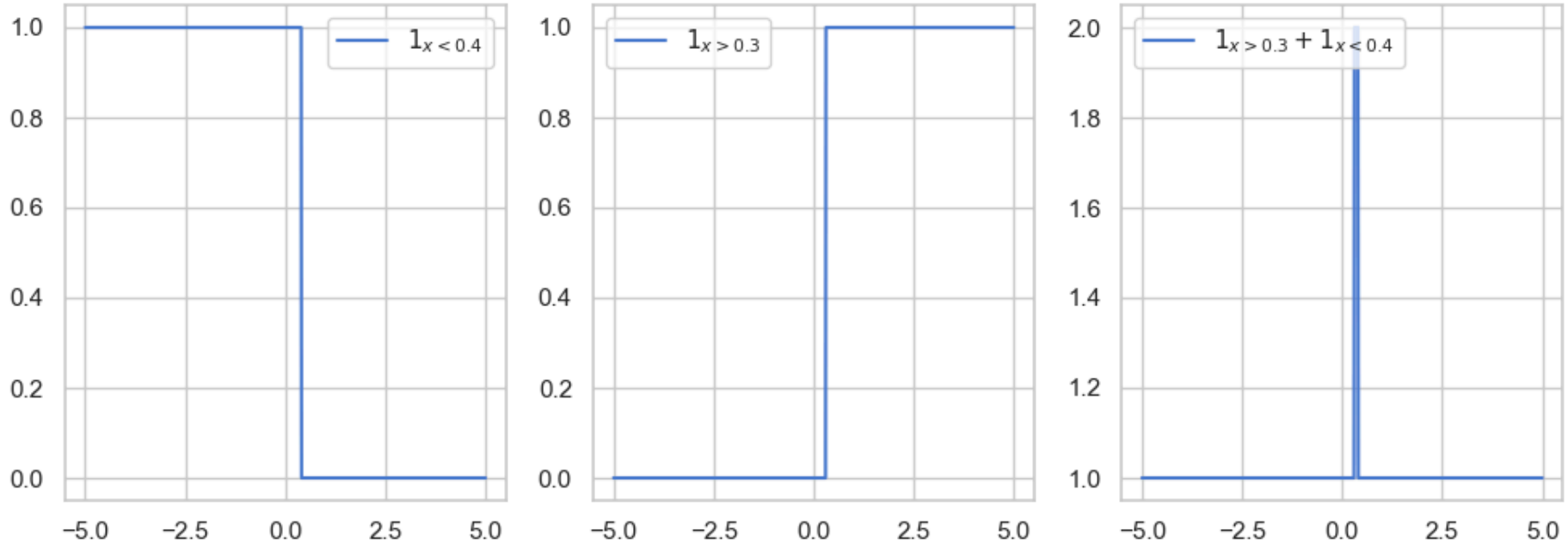
$$\sigma(b_i x + c_i) = \begin{cases} 1 & (x > t) \\ 0 & (x \leq t) \end{cases}$$

とすることができる. (例: $b_i = 10^{100}, c_i = 2 \times 10^{100}$ なら $t = 2$)

さらに b_i を負の非常に大きい数にすると, **逆のバージョンも作れる.**

証明ステップ2. 矩形関数の作り方

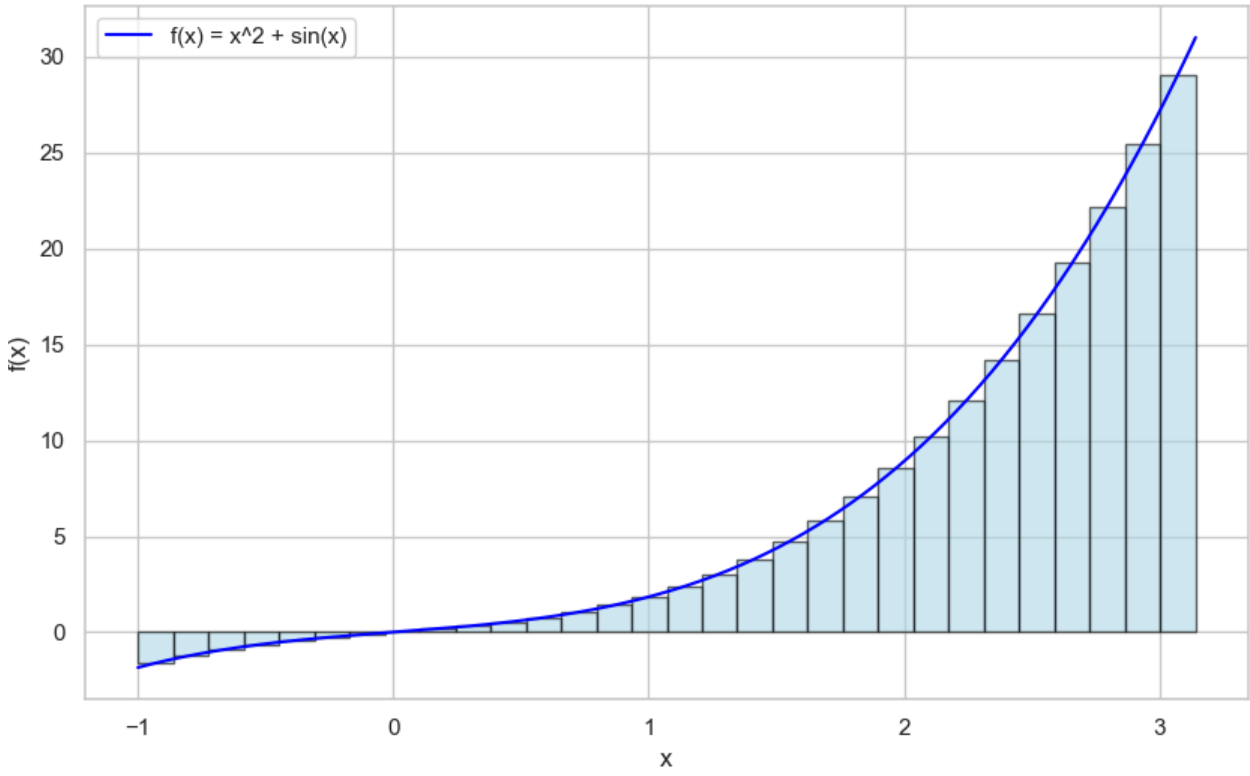
✅ すると 正の大きな数によってステップ関数にしたものと
負の大きな数によってステップ関数にしたものを足し合わせることで
矩形関数を作ることができる！



証明ステップ3.

✅ **これさえできればもうOK**

連続関数を全て**矩形関数の和**としてみればよい.



万能近似できるからいい？

任意の連続関数を近似できるモデルはニューラルネットワークだけ？

⇨ **全然ふつうにNO.**

❌「万能近似ができるからニューラルネットワークがよくつかわれる」

+ あくまでそのような a, b, c が存在するという主張であって、
それを求める方法については何ら保証していない

⇩

ニューラルネットワークの優位性を考えるなら,もうすこし議論を進めていく必要がある

「深さ」は必要？

この結果の主張:

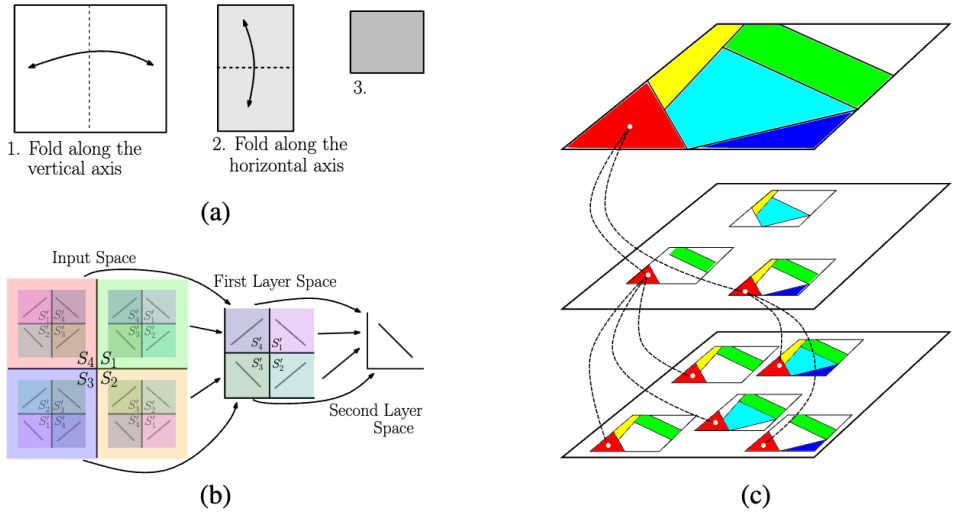
十分幅が広い「隠れ層」が一つあれば十分

世の中の主張:

たくさんの層があるNNがよく機能する

⇩　なぜ？

A. 層を深くすると指数関数的に表現力が上がり, 幅を広くすると多項式的に表現力が上がる. [1]



[1] Montufar, Guido F., et al. "On the number of linear regions of deep neural networks." Advances in neural information processing systems 27 (2014).
画像も同論文より

機械学習講習会 第五回

-「ニューラルネットワークの学習と評価」

traP Kaggle班

2024/07/03

振り返りタイム

- 我々の学習手法は, $f(x) = ax + b$ というモデルの構造自体に直接依存しているわけではなかった
- $f(x) = ax + b$ というモデルの構造では直線しか表現することができないので, 違う形を考えることにした
- 「基になる」簡単な関数の **合成** と **和** を考えることでかなり複雑な関数も表現できることがわかった
- 「基になる」関数の選び方を考える上で, この関数自体もパラメータによって変化させるモデルとしてニューラルネットワークを導入した
- ニューラルネットワークは非常に幅広い関数を表現できることがわかった

DNNの学習はむずかしい？

ニューラルネットワークは非常に多くのパラメータをもつ
(例: 全結合層はそれぞれ $W \in \mathbb{R}^{n \times m}$ と $b \in \mathbb{R}^m$ のパラメータを持つ)

⇩

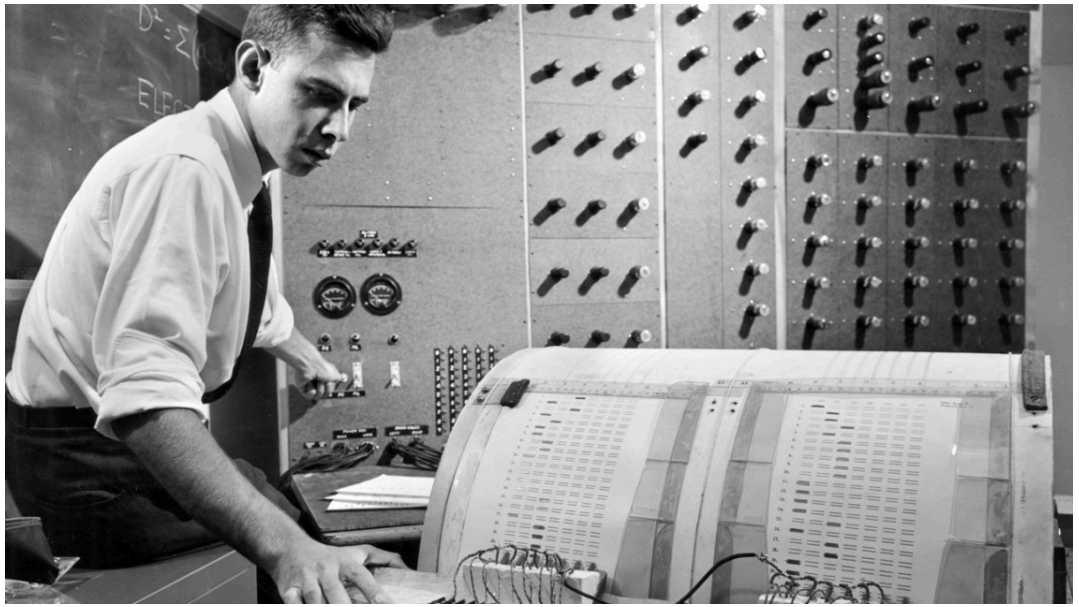
学習はそれなりに難しいタスク

DNNの学習はむずかしい？

ニューラルネットワーク研究の歴史を遡ってみると...?

⇩

😯 実は真空管で計算をしている時代からニューラルネット(の原型)が作られて計算されていた



右は真空管を使ったパーセプトロンの計算機を作っている Frank Rosenblatt.
10ニューロン程度のパーセプトロンを作っていたらしい.
(画像は https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon より)

DNNの学習はむずかしい？

- 1986年ごろ: 多層パーセプトロン
  → ニューラルネットで全部表現できる！すごい！！
  → 数学的な研究も進み始める (Hecht-Nielsen, 1987 や Cybenko, 1989 など)

DNNの学習はむずかしい？

1990年 ~ 2000年代

- ニューラルネットワークを大きくしていくと学習がとたんに難しくなる 😔
  (= まともなパラメータを獲得してくれない)

⇩

研究も下火に

学習手法の進化

Geoffrey Hinton

DBN (Deep Belief Network) やオートエンコーダに関する研究 [1][2] を通じて DNN の学習の安定化に大きく貢献

[1] Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets." Neural computation 18.7 (2006): 1527-1554.
[2] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." Science, vol. 313, no. 5786, 2006, pp. 504-507. doi:10.1126/science.1127647.

学習手法の進化

活性化関数の進化 (ReLU)
Dropout
Batch Normalization
オプティマイザの進化 (Adam, RMSprop ...)

⇩

✅ **DNN の学習を比較的安定して行えるように**

今日のおしながき①

✅ DNN の学習を安定的に，効率的に行う技法を知る

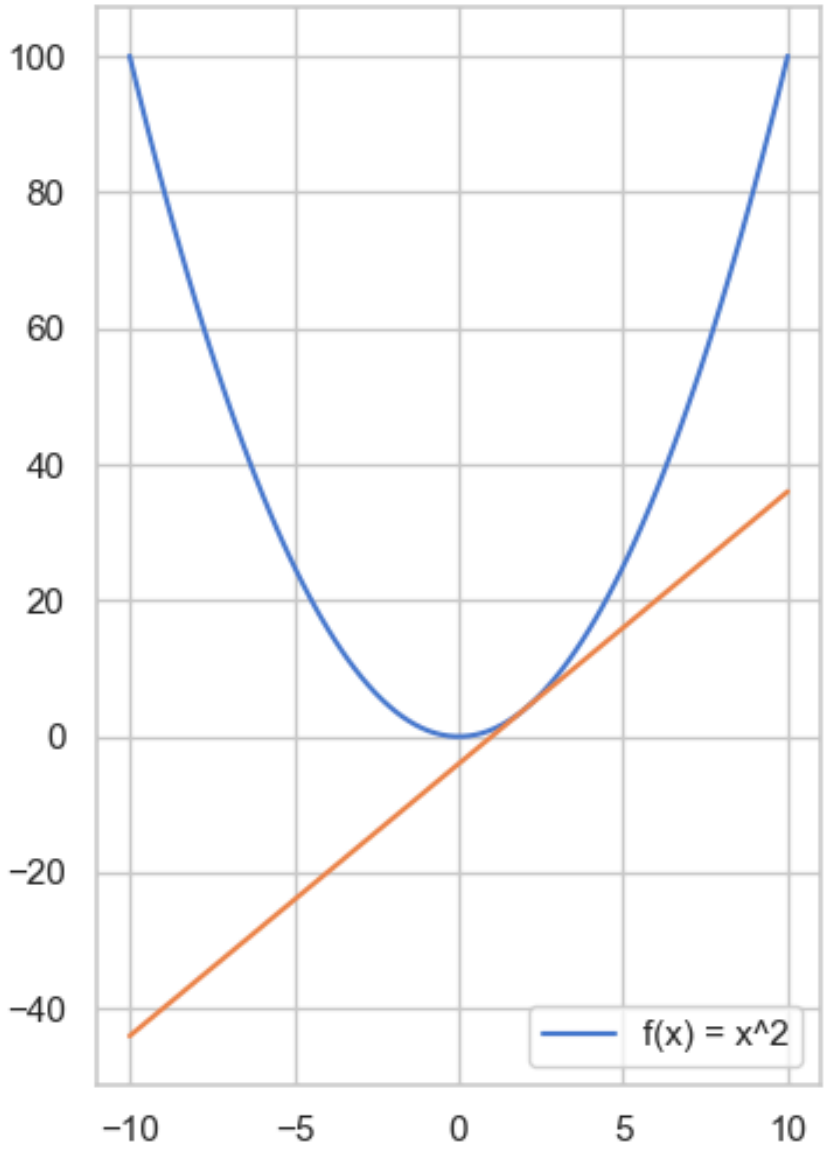
勾配降下法の復習

微分係数

$f'(x)$ は x における接線の傾き

⬇

$-f'(x)$ **方向に関数を**
すこし動かすと関数の値はす
こし小さくなる



勾配降下法の復讐

勾配降下法

関数 $f(x)$ と初期値 x_0 が与えられたとき,
次の式で $\{x_k\}$ を更新するアルゴリズム

$$x_{k+1} = x_k - \eta f'(x_k)$$

(η は**学習率**と呼ばれる定数)

今日やること1.「ニューラルネットワーク向け」の学習

勾配降下法... $x_{n+1} = x_n - \eta f'(x_n)$

をニューラルネットワークに適用するための色々な技法

- 初期化 (x_0 を決める)

⇩

- 計算 ($x_{n+1} = x_n - \eta f'(x_n)$ を計算する)

のそれぞれをカスタマイズします

今日やること1. 初期値

勾配降下法... $x_{n+1} = x_n - \eta f'(x_n)$

✅ x_0 は 自分でが決めなければいけなかった！

今日やること1. 初期値

• • •

一般の f を最小化するとき

⇨ 初期値として普遍的にいい値はない

⇨ **NNは構造が固定されているのでいい初期値を考えられる**

初期値の決め方

1. Xavierの初期値

2. Heの初期値

初期値の決め方

Xavier (Glorot) の初期値

$$\begin{cases} W_{i,j} \sim \mathcal{U}\left(-\sqrt{\dfrac{6}{n+m}}, \sqrt{\dfrac{6}{n+m}}\right) \\ b_j = 0 \end{cases}$$

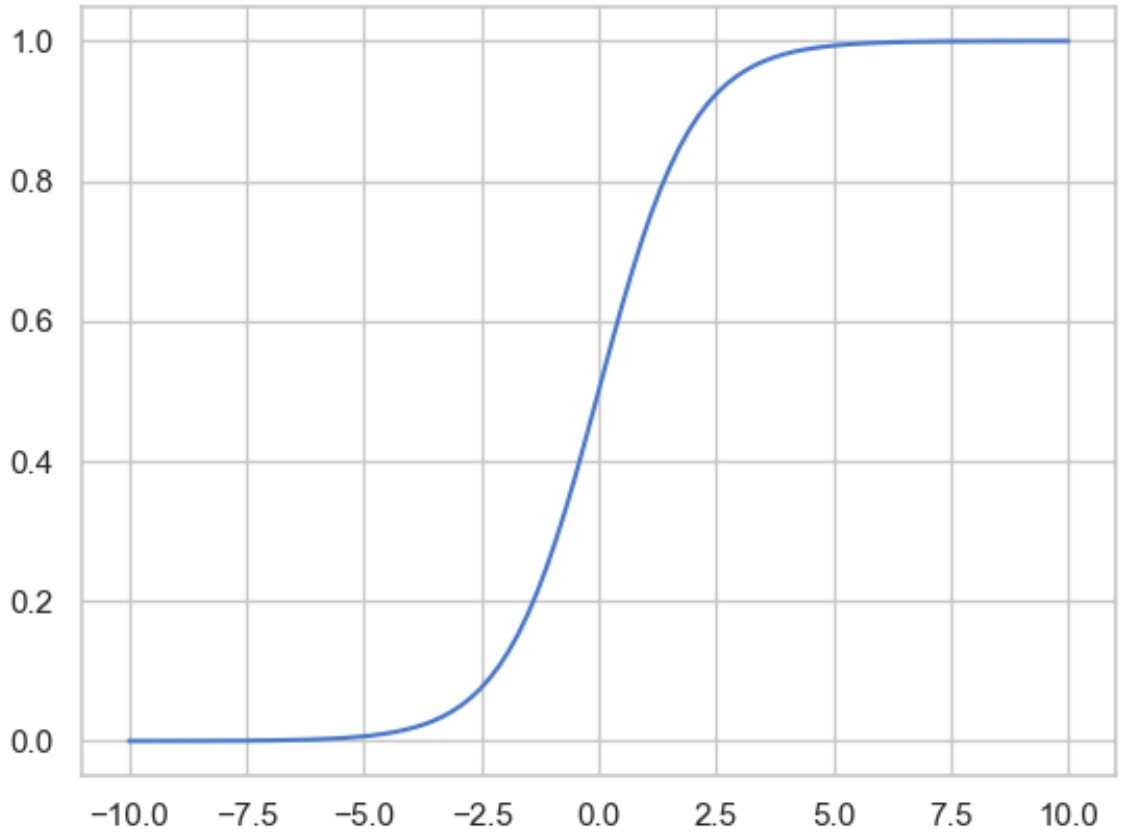
Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2010.

Xavierの初期値: 気持ち

活性化関数にとって得意なところで計算が進んでほしい.

シグモイド関数の性質

- 出力が 0 または 1 に **貼り付く**
- $|x|$ が大きいと勾配がほぼ 0

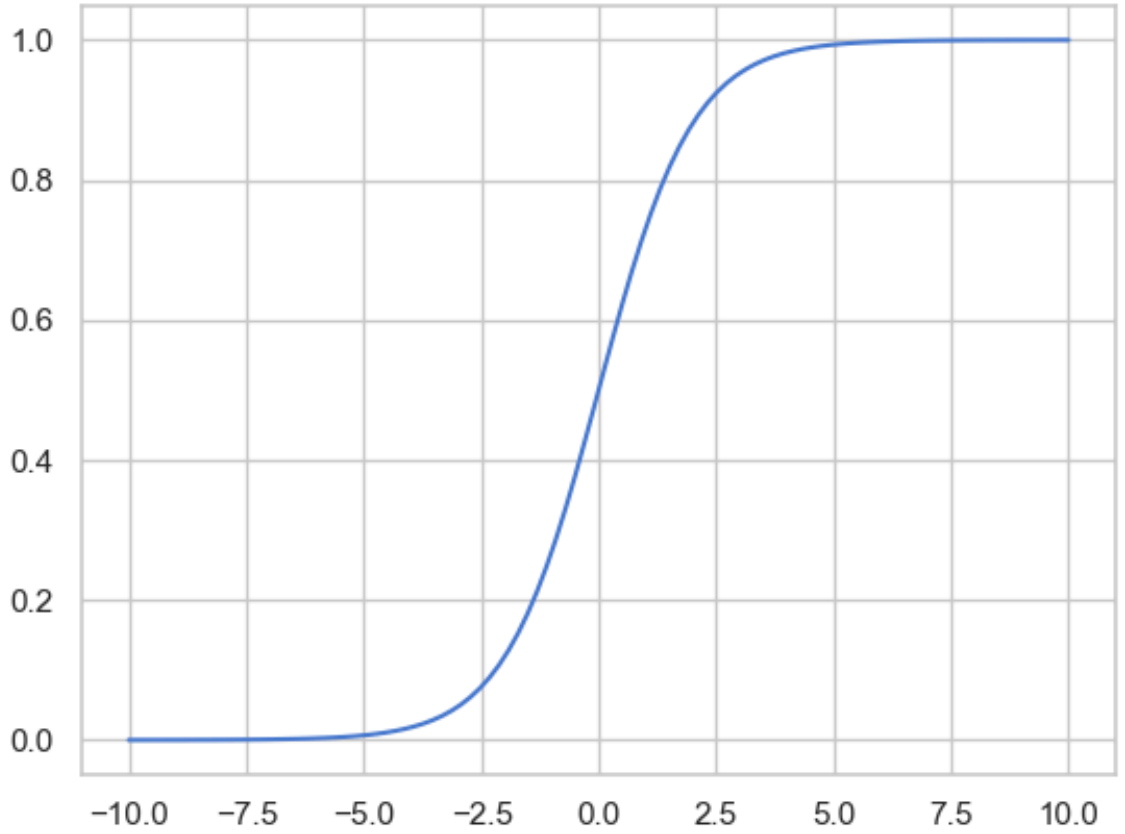


勾配消失

$$x_{k+1} = x_k - \eta f'(x_k)$$

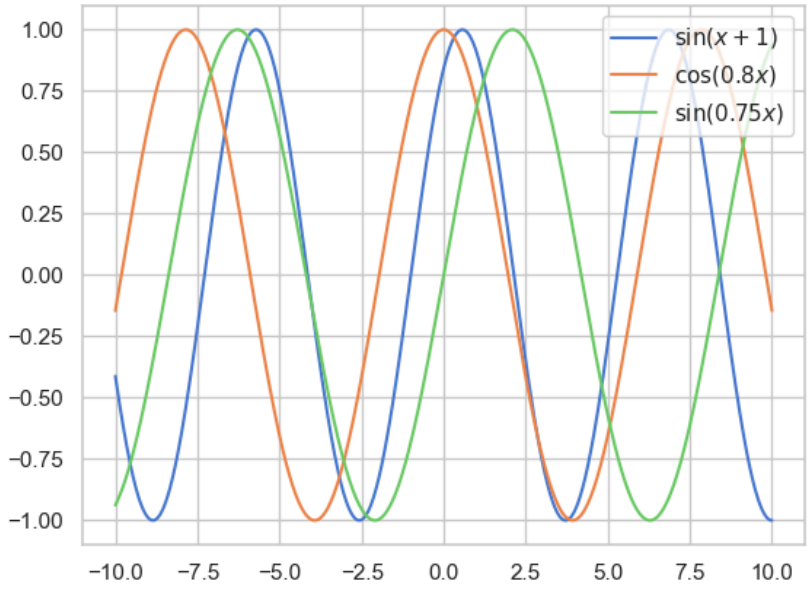
⇩

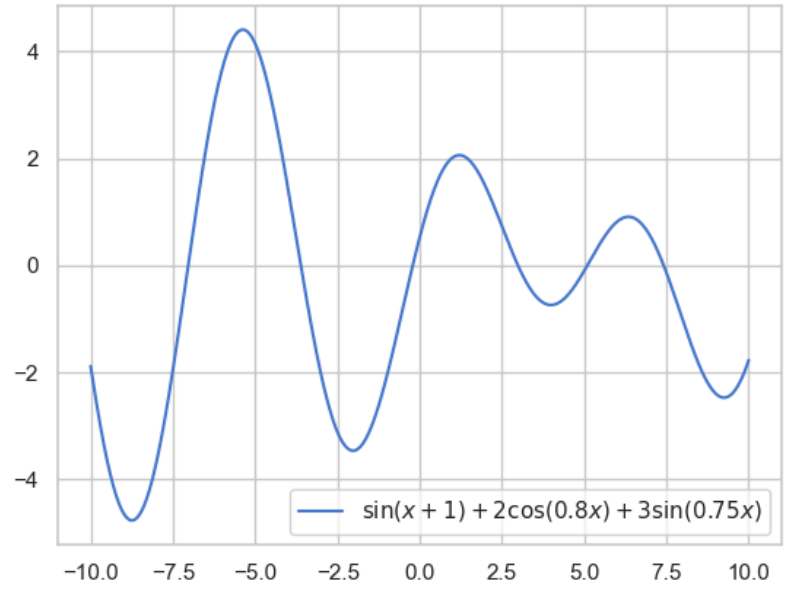
勾配がほとんど 0 だと
学習がなかなか進まなくなる❄



思い出すシリーズ: 複雑さを生む

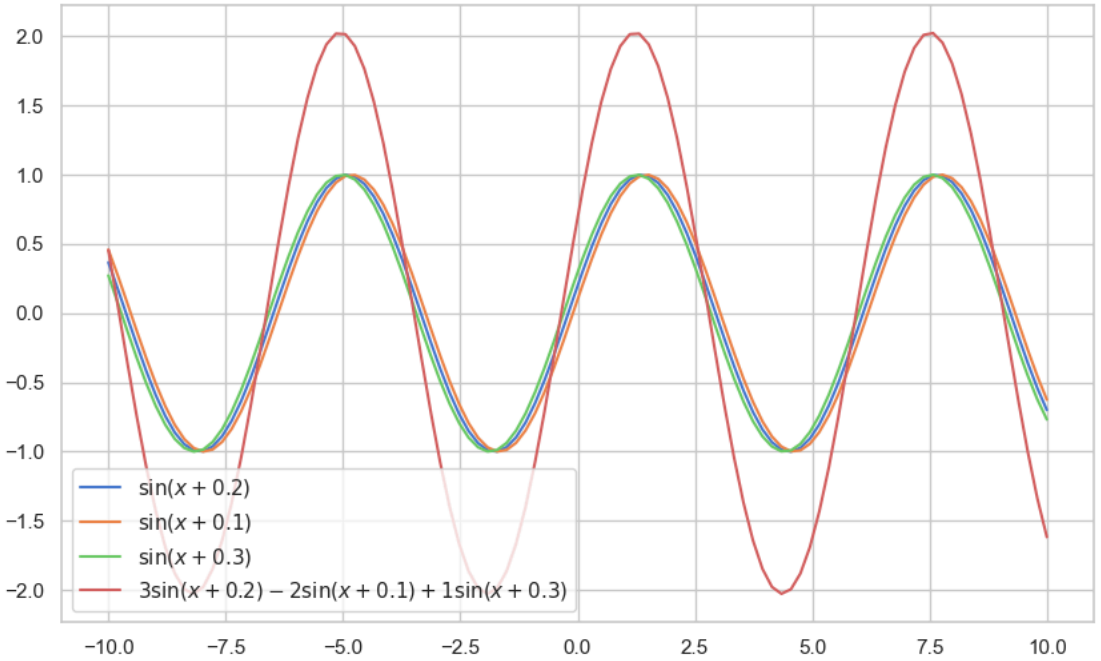
✅ 全結合層は非線形関数の和をとって複雑な関数を作っていた





思い出すシリーズ: 複雑さを生む

ほとんど同じような「基になる関数」をとっても効率がわるい



各層で分散を維持する

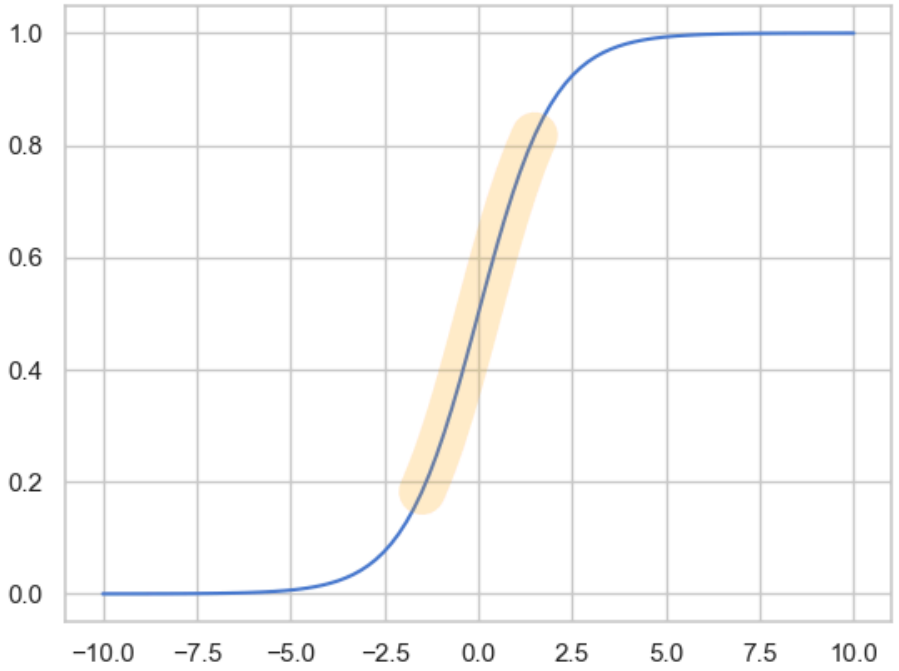
出力と勾配両方 について

- 上下に貼り付く (分散大)
- ほとんど同じ値 (分散小)

にならないように
⇔ 分散を維持するようにすると

$$\mathcal{U}(-\sqrt{6/(n+m)}, \sqrt{6/(n+m)})$$

がいい初期値になる



初期値の決め方

シグモイド関数はよくない性質 (= 勾配消失) がある！
⇨ 次第に $\mathrm{ReLU}(x) = \max(0, x)$ が使われるようになる

⇩ **ReLU 向けの初期値** (導出は Xavier と一緒)

He (Kaiming) の初期値

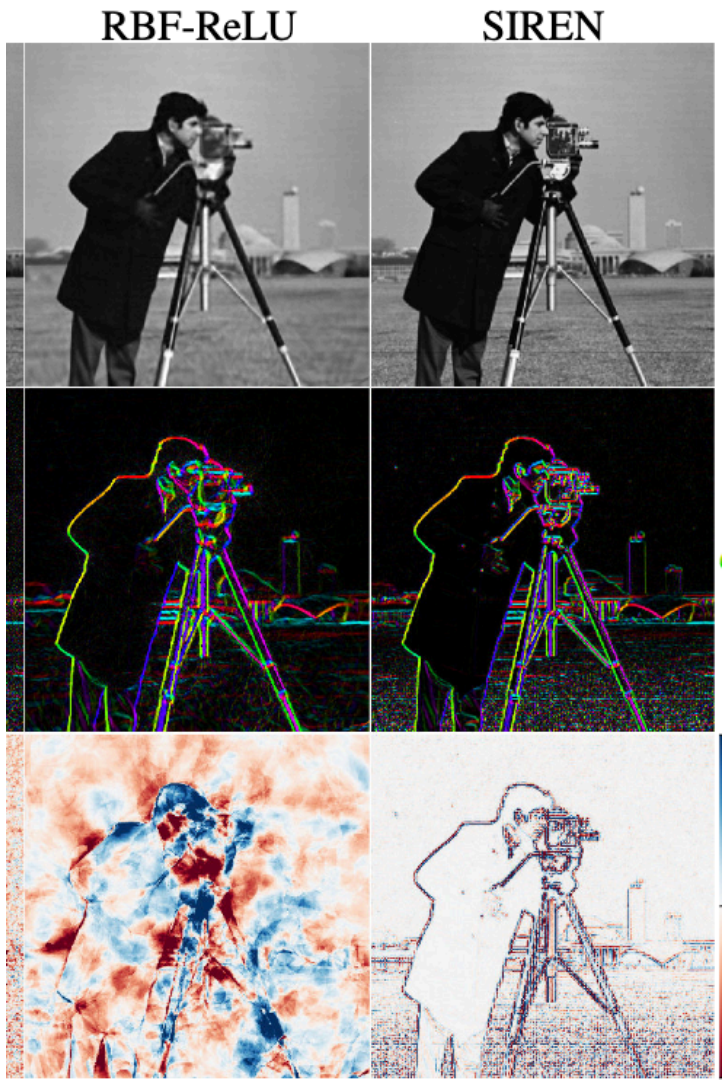
$$W_{i,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right)$$

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

導出から自然にわかること

モデルの構造 (とくに活性化関数) によって適切な初期値のとり方が変わってくる！

例) SIREN [1] という活性化関数に sin を使うモデルは $\mathcal{U}\left(-\sqrt{6/n}, \sqrt{6/n}\right)$ がいいとされている



[1] Sitzmann, Vincent, et al. "Implicit neural representations with periodic activation functions." Advances in neural information processing systems 33 (2020): 7462-7473.
画像も同論文より引用

ちゃぶ台ひっくり返し

1. 初期値で頑張る
2. モデルの中で直してしまう

Batch Normalization

Batch Normalization

- 入力をミニバッチごとに正規化するレイヤー

⇨ 学習の効率化にかなり役立ち **初期化の影響を受けにくくする**

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. pmlr, 2015.

おまけ: 「乱数」は初期値に必要か？

実は決定論的にやってもよい？

➪ **ZerO Initialization [1]**

✅ 乱数生成をやめると再現性が向上してうれしい.

[1] Zhao, Jiawei, Florian Schäfer, and Anima Anandkumar. "Zero initialization: Initializing neural networks with only zeros and ones." arXiv preprint arXiv:2110.12661 (2021).

初期値のまとめ

- 適切な初期値を選ぶことで学習の安定性を向上させることができる
- Xavierの初期値, Heの初期値などがよく使われる
- 一方, 近年は初期値にそこまで神経質にならなくてもよくなりつつある
  - さらに一方で (!?) 特殊なネットワークではそれに適した初期値を使うとよい

ミニバッチ学習

☑ 初期化 (x_0 を決める)　← Done!

⇩

☐ 計算 ($x_{n+1} = x_n - \eta f'(x_n)$ を計算する)

オプティマイザ

🔲 $x_{n+1} = x_n - \eta f'(x_n)$

$f(x_n)$ の計算はできるようになった

⇩

われわれは自動微分が使えるので
これで $f'(x_n)$ も計算できる 🤗

⇩

計算の過程もカスタマイズする！

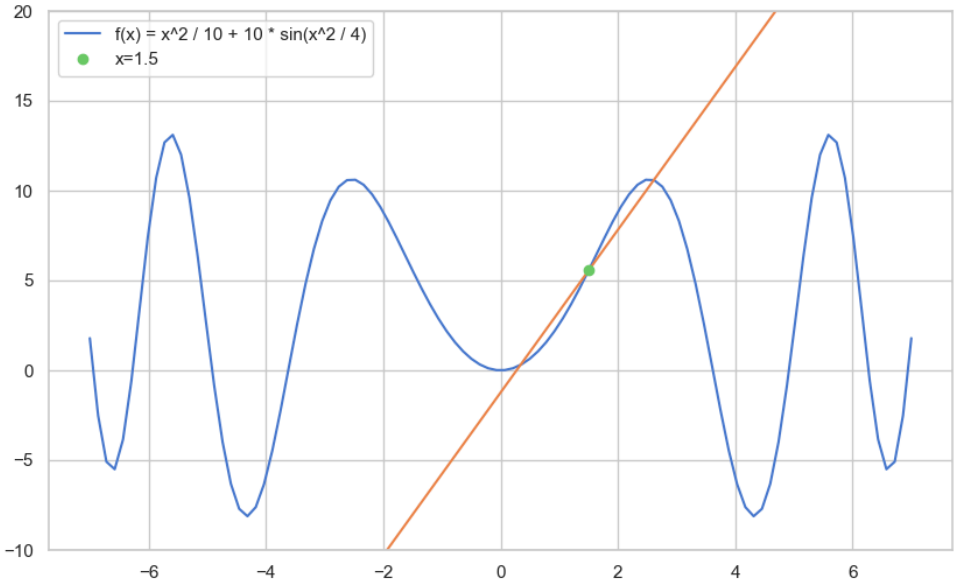
確率的勾配降下法

確率的勾配降下法 (SGD)

データの **一部** をランダムに選んで,
そのデータに対する勾配を使ってパラメータを更新する

思い出すシリーズ: 局所最適解

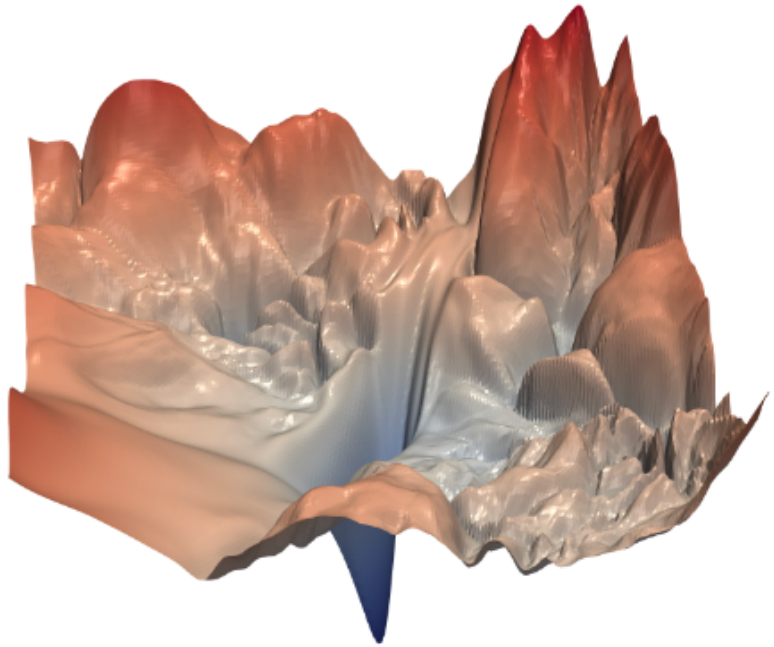
局所最適解 ... 付近で最小
大域最適解 ... 全体で最小



NNの「損失平面」

https://www.telesens.co/loss-landscape-viz/viewer.html で見てみよう！

(⚠️🚨 実際に右の3次元空間上で探索しているわけではないです！！！)



Li, Hao, et al. "Visualizing the loss landscape of neural nets." Advances in neural information processing systems 31 (2018).

画像も同論文より

局所最適解にハマらないようにするには？

谷からの脱出方法

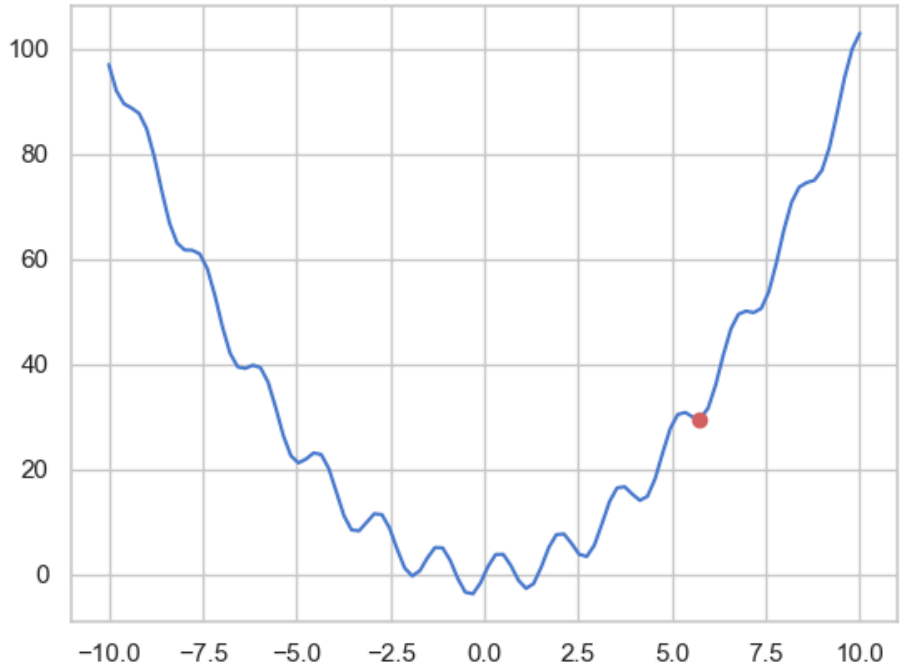
⇨ **ランダム性 を入れる**

局所最適解にハマらないようにするには？

データを選ぶときに
ランダム性が入る！

⇩

局所最適解にトラップされない



更新式の改善

プレーン な勾配降下法の更新式

$$x_{n+1} = x_n - \eta f'(x_n)$$

オプティマイザ

- 学習率に鋭敏でなく
- 安定して
- 高速に
- 高い性能を得る

ためにいろいろなオプティマイザが提案されている

(PyTorch 本体には13個)

Adadelta	Implements Adadelta algorithm.
Adagrad	Implements Adagrad algorithm.
Adam	Implements Adam algorithm.
AdamW	Implements AdamW algorithm.
SparseAdam	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm).
ASGD	Implements Averaged Stochastic Gradient Descent.
LBFGS	Implements L-BFGS algorithm.
NAdam	Implements NAdam algorithm.
RAdam	Implements RAdam algorithm.
RMSprop	Implements RMSprop algorithm.
Rprop	Implements the resilient backpropagation algorithm.

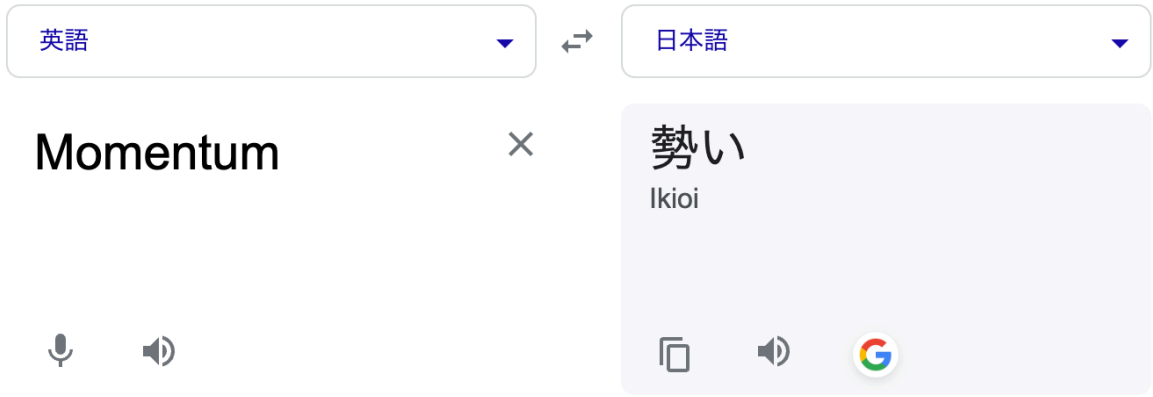
画像は https://pytorch.org/docs/stable/optim.html より (2024年7月3日)

オプティマイザの工夫の例: Momentum

Momentum

$$\begin{cases} v_{n+1} = \alpha v_n - \eta f'(x_n) \\ x_{n+1} = x_n + v_{n+1} \end{cases}$$

Momentum



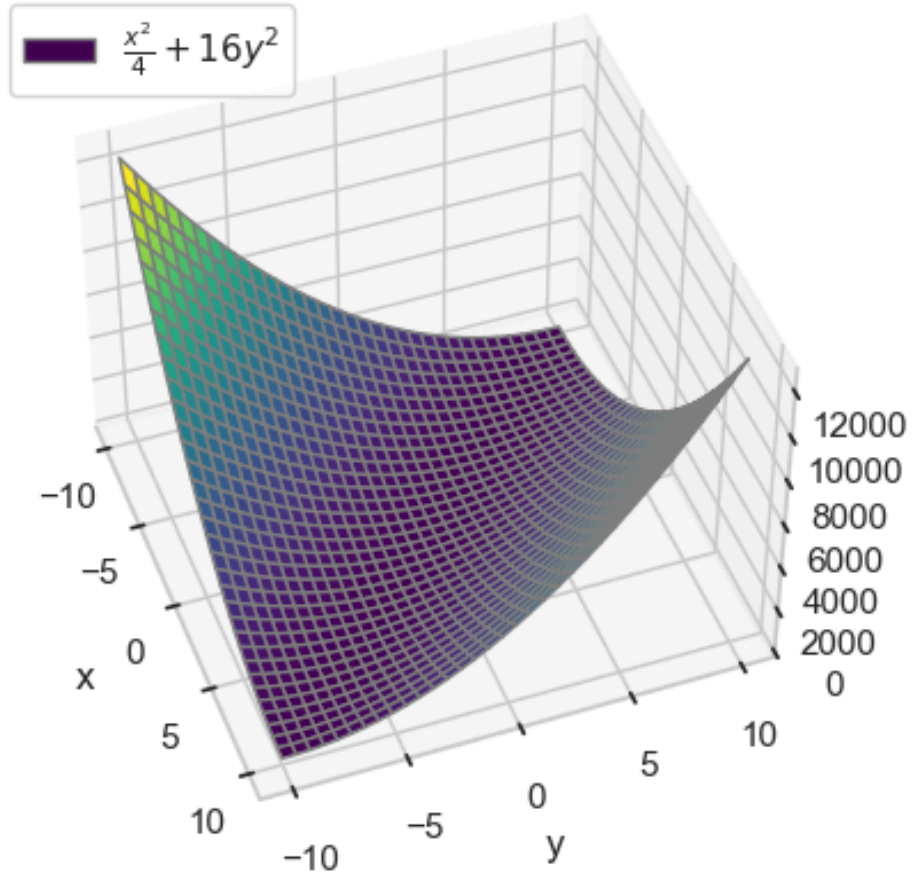
局所最適解の谷の例

✅ $f(x, y) = \dfrac{x^2}{4} + 16y^2$

の最小値

$x = 0,\ y = 0$

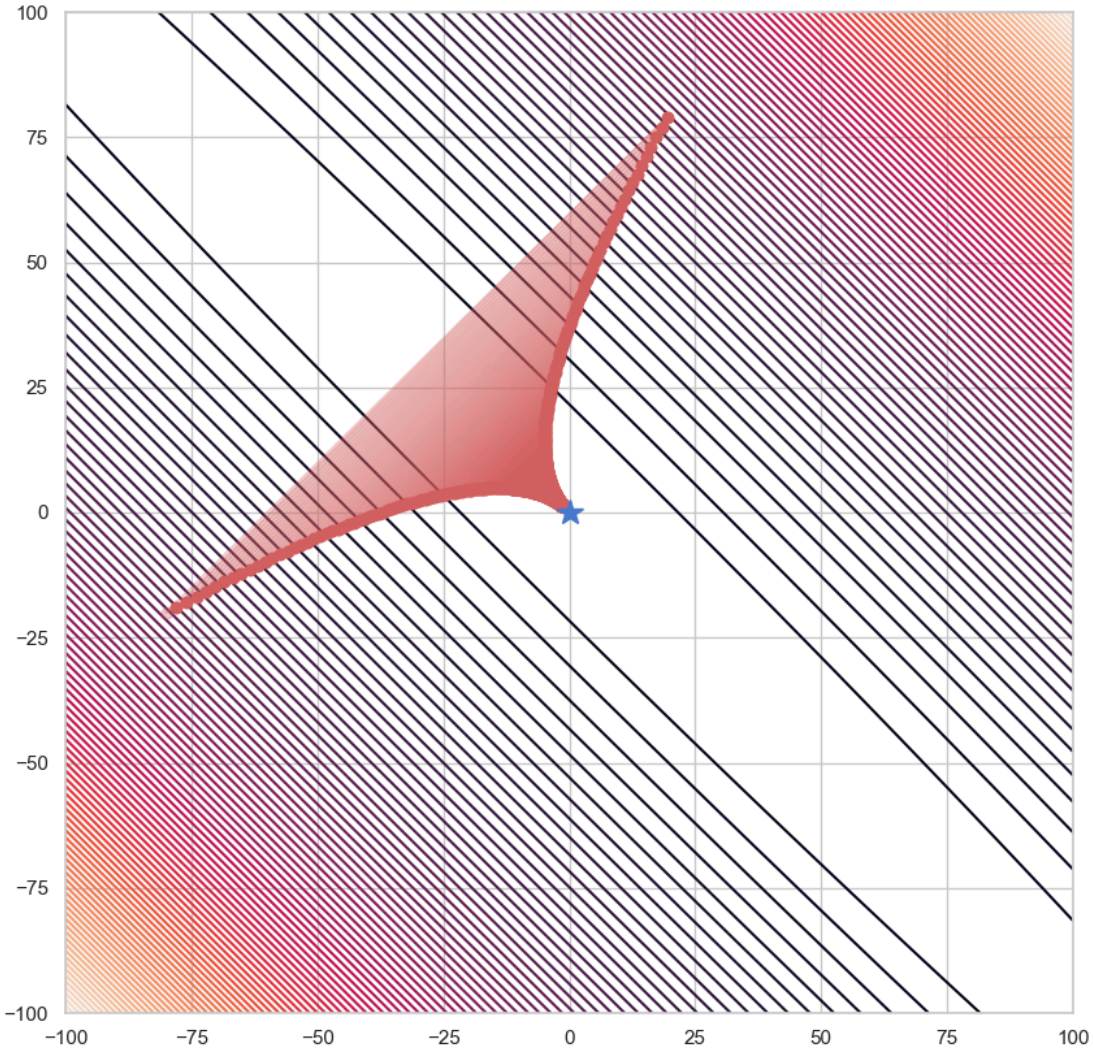
を勾配降下法で求めてみる



局所最適解の谷の例

谷を往復し続けて収束の効率がめちゃくちゃ悪い 😔

アニメーション:
https://abap34.github.io/ml-lecture/ch05/img/ch05_gradient_descent.gif



「勢い」の導入

Momentum

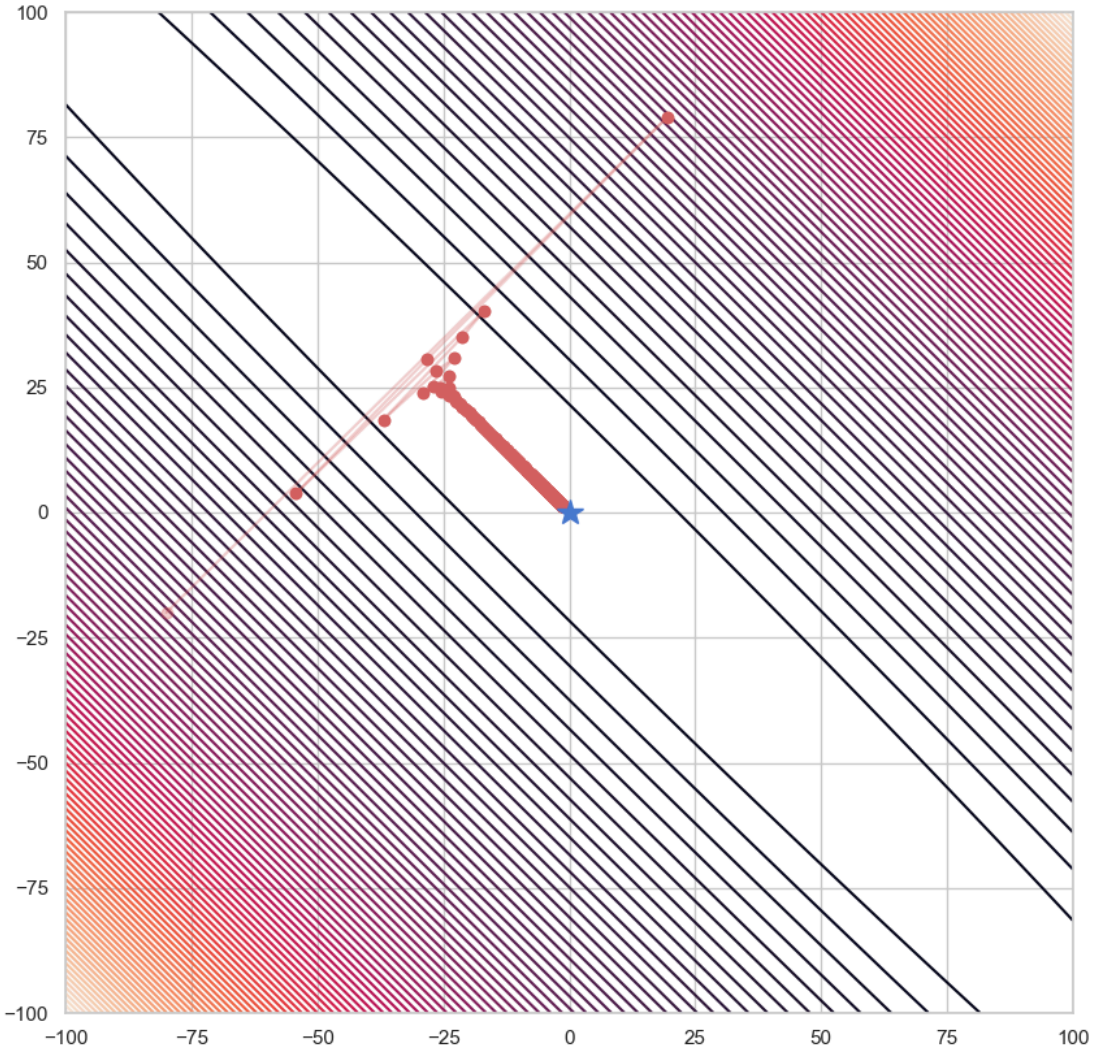
勢い を定義して,前の結果も使って更新する

$$\begin{cases} v_{n+1} = \alpha\, v_n - \eta f'(x_n) \\ x_{n+1} = x_n + v_{n+1} \end{cases}$$

Momentum による更新

✅ なにもしない SGD より早く収束！

アニメーション:
https://abap34.github.io/ml-lecture/ch05/img/ch05_momentum.gif



momentum で遊べるサイトです. おすすめです
https://distill.pub/2017/momentum/

学習

☑ 初期化 (x_0 を決める)

⇩

☑ 計算 ($x_{n+1} = x_n - \eta f'(x_n)$ を計算する)

モデルを「評価」する

✅ 「学習」部分は完了

「良さ」を再考する

いよいよ本格的なモデルが作れそうになってきた！

⇨ その前に **モデルの「良さ」** についてもう一度考えてみる

「良さ」を再考する

例) アイスの予測ができるモデルが完成した！！！

⇨ こいつの「良さ」をどう定義するべきか？

今までの「良さ」　～損失関数～

[定義] これまでの「良さ」

モデルの「良さ」とは「損失関数の小ささ」である！

これはすでに観測された値をもとに計算されるパラメータの関数で,
学習によってこの良さをあげるのがわれわれの目的だ！

疑問

本当にこれでよかったのか？

学習した後のことを考えよう

例) アイスの予測ができるモデルが完成した！！！

学習の際に使ったデータは
{(20℃, 300円), (25℃, 350円), (30℃, 400円), (35℃, 450円), (40℃, 500円)}

➪ さぁこれを使ってアイスの値段を予測するぞ！
➪ 来るデータは....

{22℃, 24℃, 25℃, ・・・}

※ 重要: これらのデータは学習段階では存在しない

真の目的は？

> なんか来月の予想平均気温30度って気象庁が言ってたな.
> 来月の売り上げが予想できたらどのくらい牛乳仕入れたらいいかわかって嬉しいな.

⇩

本当の目的は 未知のデータに対して精度良く推論すること

実はわれわれが勝手にしていた非常に重要かつ大胆な仮定

「将来も同じような入力がくる」

われわれが本当にしていたこと

未知のデータ X に対しての誤差 $\mathcal{L}(X;\boldsymbol{\theta})$ は最小化できない (未知だから)

かわりに既知のデータ x' に対しての誤差 $\hat{\mathcal{L}}(x';\boldsymbol{\theta})$ を最小化する

⇩ なぜなら,

将来のデータと過去のデータは大体変わらないだろうから.

「良さ」の再定義

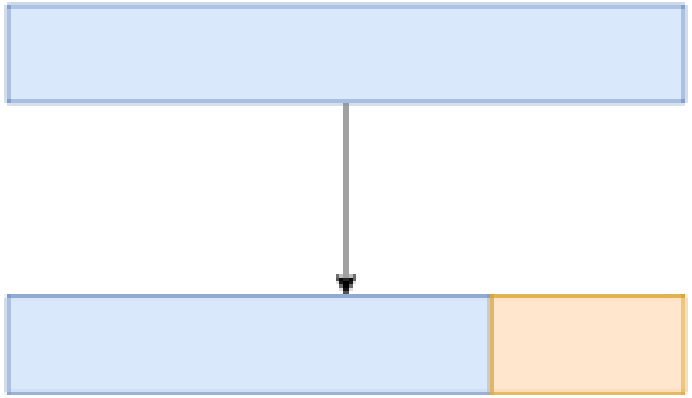
ほんとうに高めたいもの: **未知のデータへの予測性能**

これを新たに良さとしたい！！

未知のデータに対する性能を検証する

バリデーション

学習データを分割して一部を学習に使い, 残りを検証に使う



未知のデータに対する性能を検証する

学習データ

{ (20℃, 300円), (25℃, 350円), (30℃, 400円), (35℃, 450円), (40℃, 500円) }

⇩ 分割

- 学習データ
  { (20℃, 300円), (25℃, 350円), (30℃, 400円) }
- 検証用データ
  { (35℃, 450円), (40℃, 500円) }

未知のデータに対する性能を検証する

学習データ
{ (20℃, 300円), (25℃, 350円), (30℃, 400円) }

のみで学習をおこなう

⇩

(35℃, 450円), (40℃, 500円)に対して推論を行い,誤差を評価

400円,500円と推論したとすると,
「検証用データに対する」平均二乗誤差は

$$\frac{1}{2}\left((400-450)^2+(500-500)^2\right)=1250$$

未知のデータに対する性能を検証する

学習データ: { (20℃, 300円), (25℃, 350円), (30℃, 400円) } のみで学習!

検証用データはパラメータの更新に使わず誤差の計算だけ

⇩　つまり

擬似的に 未知のデータ を作成して,「未知のデータに対する性能」を評価

何が起きたか？

われわれの真の目標は **未知のデータをよく予測すること**

⇨ **モデルの「良さ」は 「検証用データに対する性能」**

損失関数と評価指標

これの計算結果に基づいてモデルを変更することはない. 単に評価するだけ

⇩

計算さえできればいいので,われわれの学習手法で損失関数が満たす必要があった

- 微分可能

などの条件は必要ない！

⇩

もっといろいろなものが使える.

例) 正解率, 絶対誤差　etc....

損失関数と評価指標

この検証用データに対して定義される「良さ」を **「評価指標」** という.

つまり **損失関数の値を最小化することで「評価指標を改善する」のが目標.**

損失関数と評価指標

注意⚠️: これらは学習とは全く独立した作業.

⇨ **これの計算結果に基づいてモデルを変更することはない. 単に評価するだけ**

⇩

逆にいえば **評価指標は直接最適化されない！**

⇩

損失関数を最小化することで評価指標が改善するように損失関数を考える.

多分同じ中身.

既知のデータ x′

未知のデータ X

損失関数: L(x′)

究極目標: S(X)

計算・最適化できる！(x′ は観測済み)

計算できない！(X は未知だから)

間接的に最適化

同じ状況にして擬似的に計算

なぜ同じ状況と言えるのか？
⇨ 多分同じ中身と仮定してるから.

検証用データに対する評価指標 S(x″)

計算できる.

ちょっとまとめ

- 損失関数の値はあくまで「訓練データに対してこれくらいの誤差になるよ」という値
- ほんとうに興味があるのは, 知らないデータに対してどれくらいうまく予測できるか
- これの検証のために擬似的に学習に使わない未知のデータを作り, 未知のデータに対する予測の評価をする

バリデーションの手法や切り方についてはいろいろあり, 話すとかなり長くなりますのでここでは割愛します.
例えば Cross Validation や時系列を意識した Validation, テストデータとバリデーションデータの性質を近づけるための手法などもあります。
詳しくは 8月に実施予定の講習会で扱われるはずです！

バリデーションと過学習

バリデーションデータは学習データからランダムにとってきたもの.

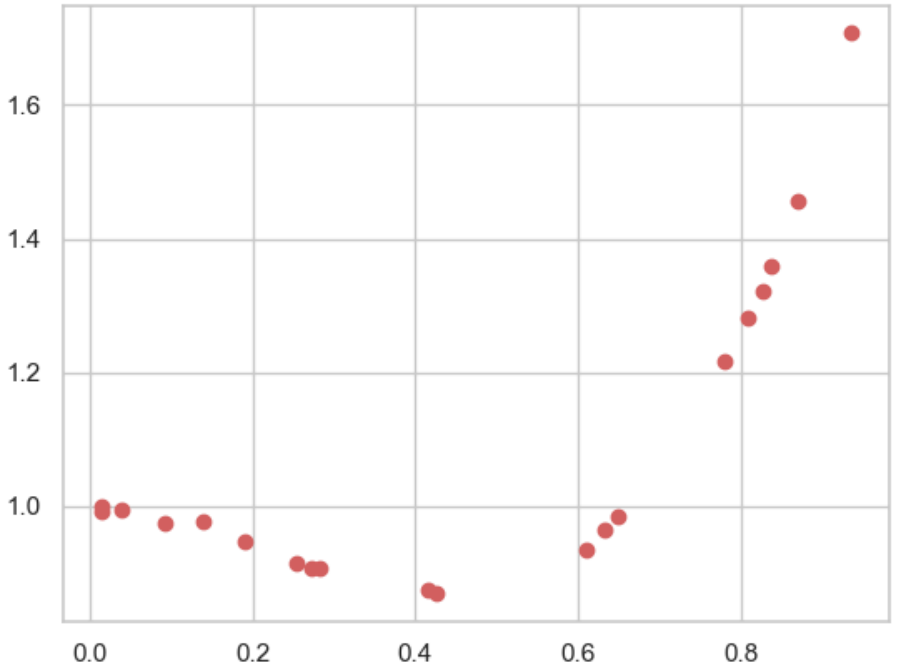
⇨ 学習データと評価の結果が異なることってあるの？ 🤔💭

⇩

はい.

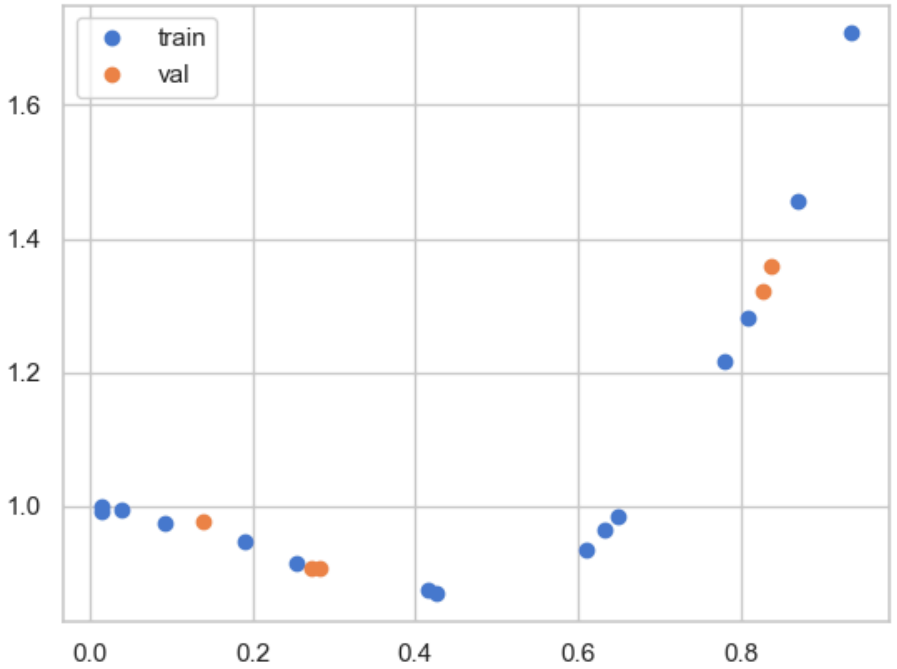
過学習

$f(x) = 3x^3 - 2x^2 + 1$ にちょっと
だけ誤差を載せたもの 👉



過学習

学習データと検証データに分ける 👉



振り返り

NN の万能近似性から, 常に損失を 0 にできる.

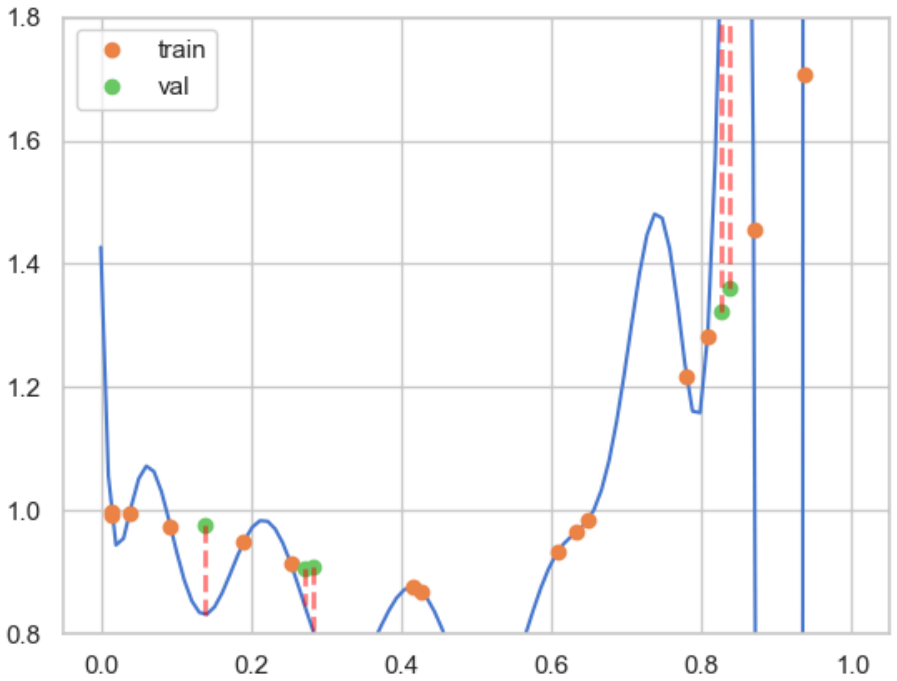
前期の線形代数の知識だけで証明できるので暇な人はやってみてください！
もう少し正確に書くと「"矛盾のないデータ" （$x_i = x_j \Rightarrow y_i = y_j$ が成立している) なら任意の i に対して $y_i = f(x_i)$ となる NN が存在する」を示してください

過学習

学習データに対して損失関数を
最小化ヨシ！ ✍️

⇩

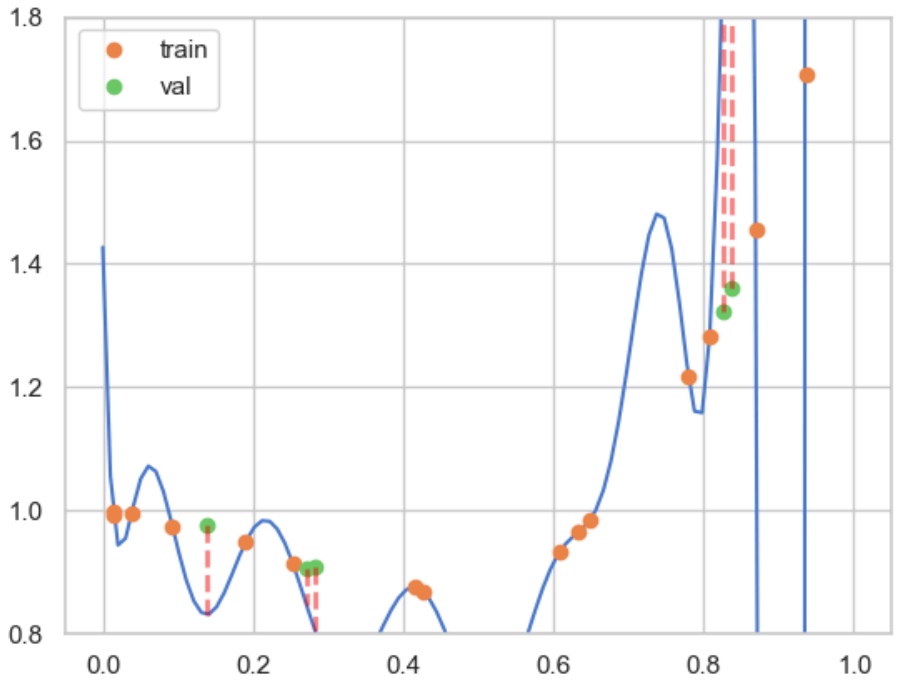
損失関数は小さくできたが
バリデーションデータには全く
当てはまっていない！！



過学習

過学習 (過剰適合, overfitting, overlearning)

学習データに過剰に適合してしまい, 未知のデータに対する予測性能が低下してしまっている状態.



学習曲線 (learning curve)

学習曲線

(learning curve)

- 横軸に学習のステップ
- 縦軸に損失関数の値

をプロットしたもの

➪ 学習曲線を見て過学習を見つける

バリデーションの重要性について

「AI作りました！ちなみにどのくらいの精度かはわからないです笑」
だと実運用はできない

⇩

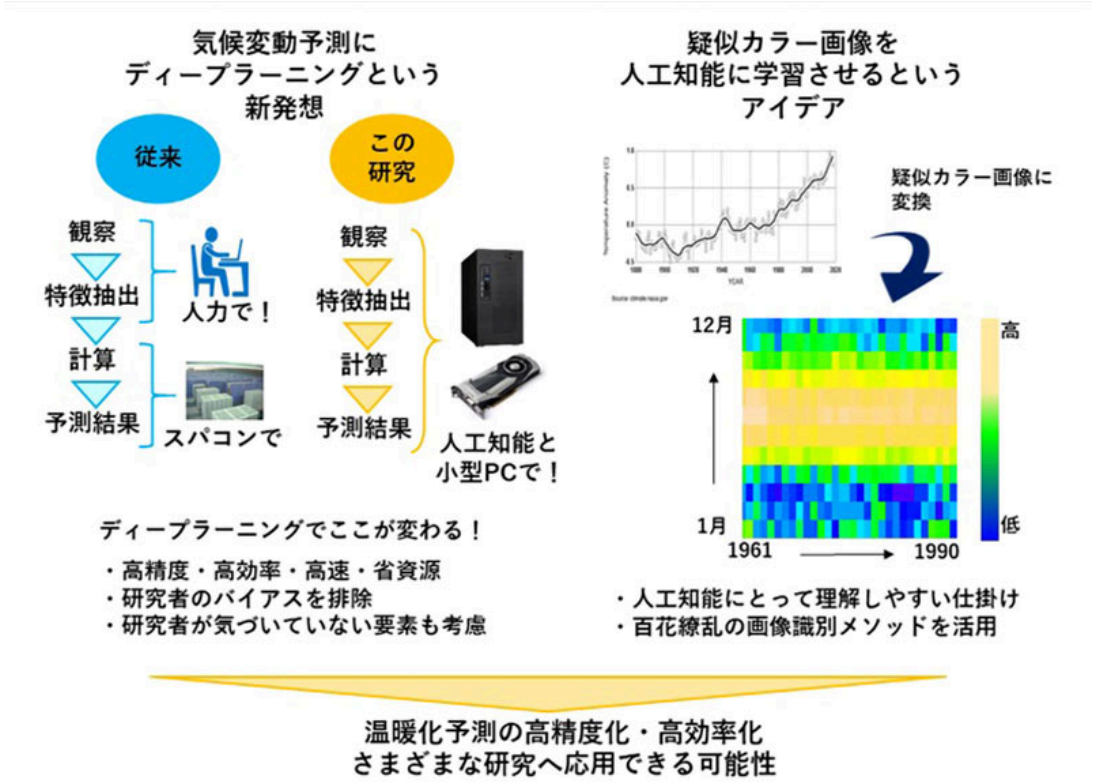
きちんとバリデーションを行うことで,
未知のデータに対する予測性能を評価することが大切.

逆に, 適切にバリデーションを行なっていないが故の嘘に気をつけよう！！

不適切なバリデーションの例

2019年の京大の研究 [1]

「過去の気温のデータから気温変化を NN で予測して, 検証用データで 97% の精度で上がるか下がるかを的中できるようになりました！」というもの



Ise, T., & Oba, Y. (2019). Forecasting Climatic Trends Using Neural Networks: An Experimental Study Using Global Historical Data. Frontiers in Robotics and AI, 6, 446979. https://doi.org/10.3389/frobt.2019.00032

不適切なバリデーションの例

Q. どこが不適切でしょう？

> ... Randomly selecting 25% of images for validation ....

不適切なバリデーションの例

A. 本来モデルが得るはずがない「未来の情報」が学習時に混入している！

バリデーションはなぜ未知のデータに対する予測性能を疑似的に計算できていたか？

⇔ 未知のデータを予測するときの状況を **擬似的に再現** していたから。

不適切なバリデーションの例

時系列なら **未知の情報に対する精度 ⇔ 2024年以降のデータに対する精度**

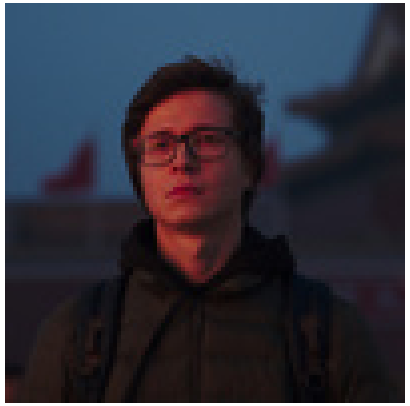
1990年のデータが検証用データに入っているなら 1991年以降のデータが学習データに入っていると不当に性能を高く見積もってしまう

バリデーションの重要性について

Kaggle をはじめとするデータ分析コンペは,「未知の情報」を予測するモデルの精度を競う

⇨ 試行錯誤している手法の「未知の情報を予測する能力」をきちんと評価することが大切！(詳しくは第七回)

バリデーションの重要性について



bestfitting はこう言っています

A good CV is half of success.

今日のまとめ

- ニューラルネットワークの学習は培われてきたいろいろな工夫があった
- バリデーションを行うことで未知のデータに対しての予測性能を評価することができる.
- バリデーションデータに対して行う評価は学習とは独立した作業なので, 微分可能であったり微分の性質が良い必要はなくいろいろな評価指標を用いることができる.
- 訓練データのみに過剰に適合した状態のことを「過学習」といい, 学習曲線に目を光らせるととでこれに気をつける必要があった
- 適切にバリデーションを行うのは **非常に重要**

機械学習講習会 第六回

-「ニューラルネットワークの実装」

traP Kaggle班

2024/07/10

今日すること

- PyTorch を使って実際にある情報を予測するニューラルネットワークを実装します
- データの読み込みからモデルの構築, 学習, 予測までを一通りやってみます
- **お題として今日から始めるコンペのデータを使います.**
  - **1 Sub まで一気に行きます！！**

はじめに

先に、コンペのルールなどの話をします

https://abap34.github.io/ml-lecture/supplement/competetion.pdf

(※ あとからこの資料を読んでいる人は飛ばしても大丈夫です)

今回のコンペのお題 ~ あらすじ ~

機械学習講習会用のオンラインジャッジを作った `@abap34` は困っていました.

攻撃はやめてくださいと書いてあるのにひっきりなしに攻撃が仕掛けられるからです.

部員の個人情報とサーバとモラルが心配になった `@abap34` は, 飛んでくる通信を機械学習を使って攻撃かを判定することで攻撃を未然に防ぐことにしました.

あなたの仕事はこれを高い精度でおこなえる機械学習モデルを作成することです.

※ 架空の話です. 僕の知る限りジャッジサーバへの攻撃は今のところきていないです.

データ

通信ログから必要そうな情報を抽出したもの (**詳細は Data タブから**)

- 接続時間
- ログイン失敗回数
- 過去2秒間の接続回数
- 特別なユーザ名 (`root` , `admin` `guest` とか) でログインしようとしたか？

⋮

データ

- train.csv
  - 学習に使うデータ
- train_tiny.csv (👈 **時間と説明の都合上 今日はこれを使います)**
  - 学習に使うデータの一部を取り出し,一部を削除
- test.csv
  - 予測対象のデータ
- test_tiny.csv (👈 **時間と説明の都合上 今日はこれを使います)**
  - 予測対象のデータの欠損値を埋めて,一部のカラムを削除
- sample_suboldsymbolission.csv
  - 予測の提出方式のサンプル (値はでたらめ)

全体の流れ

1. データの読み込み
2. モデルの構築
3. モデルの学習
4. 新規データに対する予測
5. 順位表への提出

全体の流れ1 ~モデルに入力するまで

1-0. データのダウンロード

⇩

1-1. データの読み込み

⇩

1-2. データの前処理

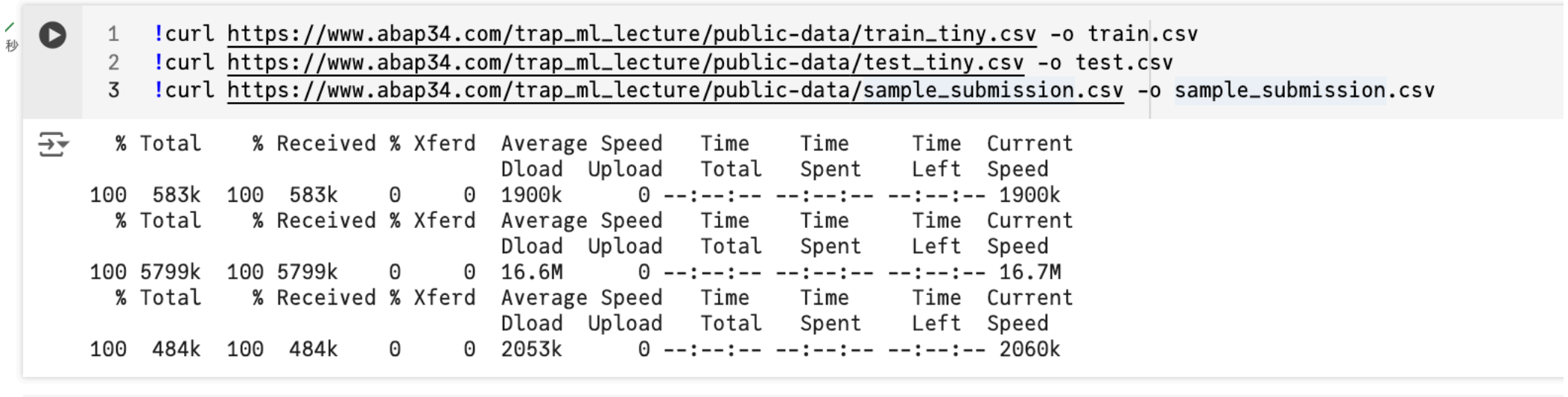
⇩

1-2. PyTorchに入力できる形に

1-0. データのダウンロード

✅ セルに以下をコピペして実行

```
!curl https://www.abap34.com/trap_ml_lecture/public-data/train_tiny.csv -o train.csv
!curl https://www.abap34.com/trap_ml_lecture/public-data/test_tiny.csv -o test.csv
!curl https://www.abap34.com/trap_ml_lecture/public-data/sample_submission.csv -o sample_submission.csv
```



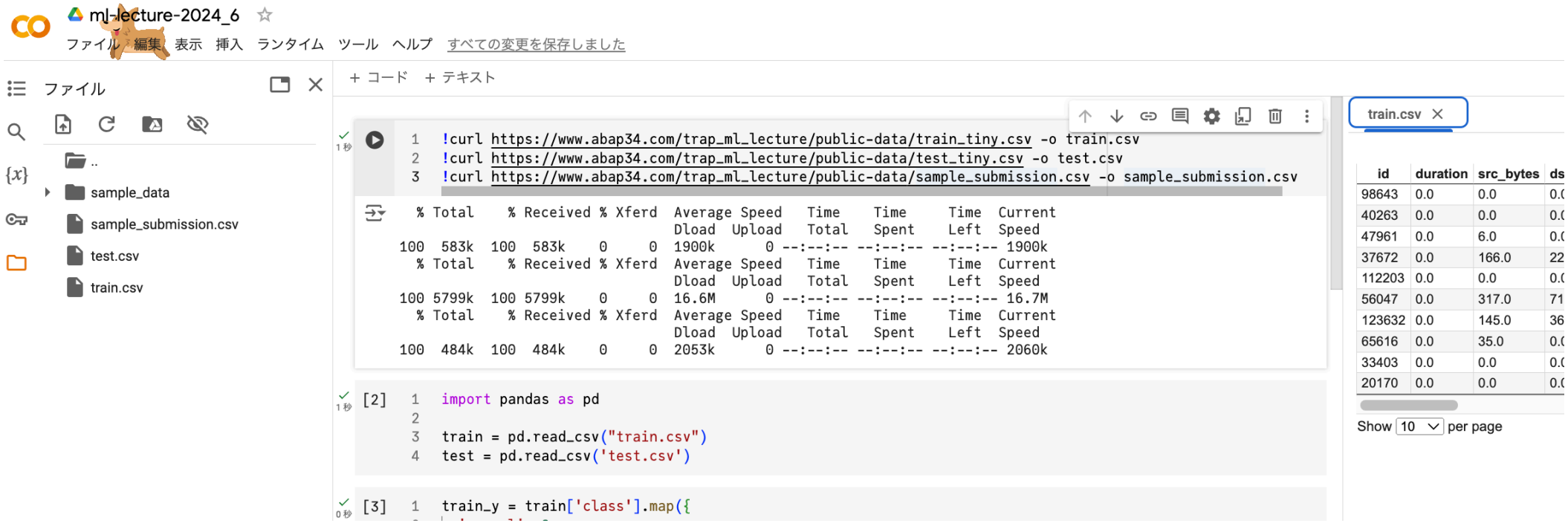
```
!curl https://www.abap34.com/trap_ml_lecture/public-data/train_tiny.csv -o train.csv
!curl https://www.abap34.com/trap_ml_lecture/public-data/test_tiny.csv -o test.csv
!curl https://www.abap34.com/trap_ml_lecture/public-data/sample_submission.csv -o sample_submission.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  583k  100  583k    0     0  1900k      0 --:--:-- --:--:-- --:--:-- 1900k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5799k  100 5799k    0     0  16.6M      0 --:--:-- --:--:-- --:--:-- 16.7M
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  484k  100  484k    0     0  2053k      0 --:--:-- --:--:-- --:--:-- 2060k
```

Jupyter Notebook では,先頭に `!` をつけることで,シェルコマンドを実行できます.

1-0. データのダウンロード

✅ 左の 📁 > train.csv, test.csv, sample_submission.csv で表が見えるようになっていたら OK！



今回のコンペのデータは ISCX NSL-KDD dataset 2009 [1] をもとに大きく加工したものを使用しています。

[1] M. Tavallaee, E. Bagheri, W. Lu, and A. Ghorbani, “A Detailed Analysis of the KDD CUP 99 Data Set,” Submitted to Second IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), 2009.

1-1. データの読み込み

✅ `pd.read_csv(path)` で, `path` にあるcsvファイルを読み込める

```
# pandas パッケージを `pd` という名前をつけてimport
import pandas as pd

# これによって, pandas の関数を `pd.関数名` という形で使えるようになる
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
```

パスはコンピュータ上のファイルやフォルダへの経路のことです.
今回は train.csv と test.csv がノートブックと同じ階層にあるので, train.csv と test.csv までの経路は,ファイル名をそのまま指定するだけで大丈夫です.
ほかにも たとえば `../train.csv` と指定すると ノートブックの一つ上の階層にある train.csv というファイルを読み込みます.

1-1. データの読み込み

```
[10]  1  import pandas as pd
      2
      3  train = pd.read_csv("train.csv")
      4  test = pd.read_csv('test.csv')
```

```
[11]  1  train
```

wrong_fragment	urgent	...	serror_rate	srv_serror_rate	rerror_rate	srv_rerror_rate	same_srv_rate	diff_srv_rate	srv_diff_host_rate	ds
0.0	0.0	...	0.000000	0.000000	0.970555	0.871439	0.135961	0.074646	0.000000	
0.0	0.0	...	1.024065	0.920154	0.000000	0.000000	0.095575	0.073942	0.000000	
0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	1.024575	0.000000	0.976209	
0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	0.953422	0.000000	0.000000	
0.0	0.0	...	0.959654	0.926067	0.000000	0.000000	0.074621	0.065885	0.000000	
...	...	...	...	...	...	...	...	...	...	
0.0	0.0	...	0.959072	0.994436	0.000000	0.000000	0.105512	0.069615	0.000000	

セルに単に変数をかくと中身を確認できます！ （Jupyter Notebook の各セルは最後に評価された値を表示するためです）
さっとデバッグするときに便利です. 中身がわからなくなったらとりあえず書いて実行してみましょう.

1-1. データの読み込み

今まで

⇩

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

def loss(a):
...
```

⇩

今回も入力と出力 (の目標) にわけておく

1-1. データの読み込み

```
train['カラム名']
```

で「カラム名」という名前の列を取り出せる 📝

⇩

今回の予測の目標は

```
train['class']
```

⇩

1-1. データの読み込み

```
train_y = train['class']
```

➪ `train_y` に攻撃? or 通常? の列が入る🙌

```
[6]  1  train_y = train['class']
```

```
[7]  1  train_y
```

```
0        attack
1        attack
2        attack
3        normal
4        attack
          ...
3623     attack
3624     attack
3625     attack
3626     attack
3627     normal
Name: class, Length: 3628, dtype: object
```

1-1. データの読み込み

機械学習モデルは **直接的には** 数以外は扱えないので数に変換しておく.

```
train_y = train['class'].map({
  'normal': 0,
  'attack': 1
})
```

```
[4]  1  train_y = train['class'].map({
     2    'normal': 0,
     3    'attack': 1
     4  })
```

```
     1  train_y
```

```
0       1
1       1
2       1
3       0
4       1
       ..
3623    1
3624    1
3625    1
3626    1
3627    0
Name: class, Length: 3628, dtype: int64
```

1-1. データの読み込み

逆に, モデルに入力するデータは `train` から さっきの列 (と `id`) を除いたもの！

```
train.drop(columns=['カラム名'])
```

を使うと `train` から「カラム名」という名前の **列を除いたもの** を取り出せる

⇩

今回は `train.drop(columns=['id', 'class'])`

1-1. データの読み込み

```
train_x = train.drop(columns=['id', 'class'])
test_x = test.drop(columns=['id'])
```

➪ `train_x` にさっきの列と `id` を除いたもの，`test_x` に `id` を除いたものが入る🙌

```
[36]  1  train_x
```

	duration	src_bytes	dst_bytes	land	wrong_
0	0.0	0.0	0.0	0	
1	0.0	0.0	0.0	0	
2	0.0	6.0	0.0	0	
3	0.0	166.0	2256.0	0	
4	0.0	0.0	0.0	0	
...	...	...	...	...	
3623	0.0	0.0	0.0	0	
3624	0.0	695.0	0.0	0	
3625	0.0	1364.0	0.0	0	
3626	0.0	990.0	0.0	0	
3627	0.0	120.0	0.0	0	

3628 rows × 30 columns

1-1. データの読み込み

✅ **データの読み込みが完了！**

今の状況整理

- `train_x` ・・・モデルに入力するデータ(接続時間,ログイン失敗回数,etc...)
- `train_y` ・・・モデルの出力の目標(攻撃? 通常?)
- `test_x` ・・・予測対象のデータ

が入ってる

1-2. データの前処理

✅ **データをそのままモデルに入れる前に処理をすることで学習の安定性や精度を向上**

(極端な例... 平均が 10^{18} の列があったらすぐオーバーフローしてしまうので平均を引く)

今回は各列に対して「**標準化**」をします

1-2. データの前処理

標準化

$$x' = \frac{x - \mu}{\sigma}$$

(μ は平均, σ は標準偏差)

1. 平均 μ_1 のデータの全ての要素から μ_2 を引くと,平均は $\mu_1 - \mu_2$
2. 標準偏差 σ_1 のデータの全ての要素を σ_2 で割ると,標準偏差は σ_1/σ_2

⇨ 標準化で **平均を0,標準偏差を1** にできる

初期化の際の議論を思い出すとこのようなスケーリングを行うことは自然な発想だと思います.
NN の入力の標準化については, LeCun, Yann, et al. "E cient BackProp." Lecture Notes in Computer Science 1524 (1998): 5-50.　にもう少し詳しく議論が載っていたので気になる人は読んでみてください.

1-2. データの前処理

✅ `scikit-learn` というライブラリの `StandardScaler` クラスを使うと,
簡単に標準化できる！

```
# sklearn.preprocessing に定義されているStandardScalerを使う
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# 計算に必要な量 (平均,標準偏差) を計算
scaler.fit(train_x)

# 実際に変換
train_x = scaler.transform(train_x)
test_x = scaler.transform(test_x)
```

`scalar.fit` によって引数で渡されたデータの各列ごとの平均と標準偏差が計算され, `scalar` に保存されます. そして, `scalar.transform` によってデータが実際に標準化されます. 勘がいい人は「`test` に対しても `train_x` で計算した平均と標準偏差を使って標準化しているけど大丈夫なのか？」と思ったかもしれないですね. 結論から言うとそうなのですが意図しています. ここに理由を書いたら信じられないくらいはみ出てしまったので, 省略します. 興味がある人は「Kaggleで勝つデータ分析の技術」p.124 などを参照してみてください.

1-2. データの前処理

```
train_x
```

```
test_x
```

などを実行してみると,確かに何かしらの変換がされている！ ✊
(ついでに結果がテーブルから単なる二次元配列 (`np.ndarray`) に変換されてる)

最初のテーブルっぽい情報を持ったまま計算を進めたい場合は, `train_x[:] = scaler.transform(train_x)` のようにすると良いです.

1-2. データの前処理

ので `train_y` もここで中身を取り出して `np.ndarray` にしておく.

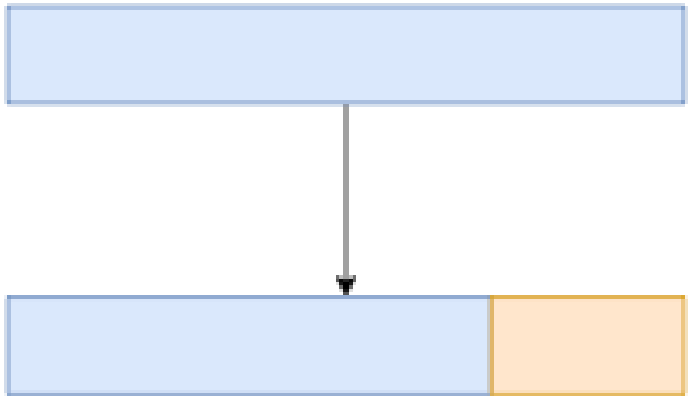
1. `train_y.values` で 中身の値を取り出せる.
2. `arr.reshape(-1, 1)` で `arr` を $N \times 1$ の形に変換できる

```
train_y = train_y.values.reshape(-1, 1)
```

`np.ndarray` のメソッド `reshape` はその名の通り配列の形を変えるメソッドです. そして `-1` は「他の次元の要素数から自動的に決定する」という意味です. 例えば, 3×4 の配列に対して `.reshape(-1, 2)` とすると 6×2 にしてくれます. (2次元目が 2 と確定しているので勝手に 6 と定まる)

1-2. データの前処理 - バリデーション

バリデーションのためにデータを分割しておく



バリデーションを前処理と呼ぶ人はいないと思いますがここでやっておきます.

1-2. データの前処理 - バリデーション

`sklearn.model_selection.train_test_split` による分割

```
train_test_split(train_x, train_y, test_size=0.3, random_state=34)
```

- `train_x`, `train_y`: 分割するデータ
- `test_size`: テストデータの割合
- `random_state`: **乱数のシード** 👈**重要！！**

1-2. データの前処理 - バリデーション

`scikit-learn` の `train_test_split` を使うと簡単にデータを分割できる！

```
from sklearn.model_selection import train_test_split
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size=0.3, random_state=34)
```

乱数シードを固定しよう！！

乱数に基づく計算がたくさん

⇩

実行するたびに結果が変わって，

めちゃくちゃ困る😢

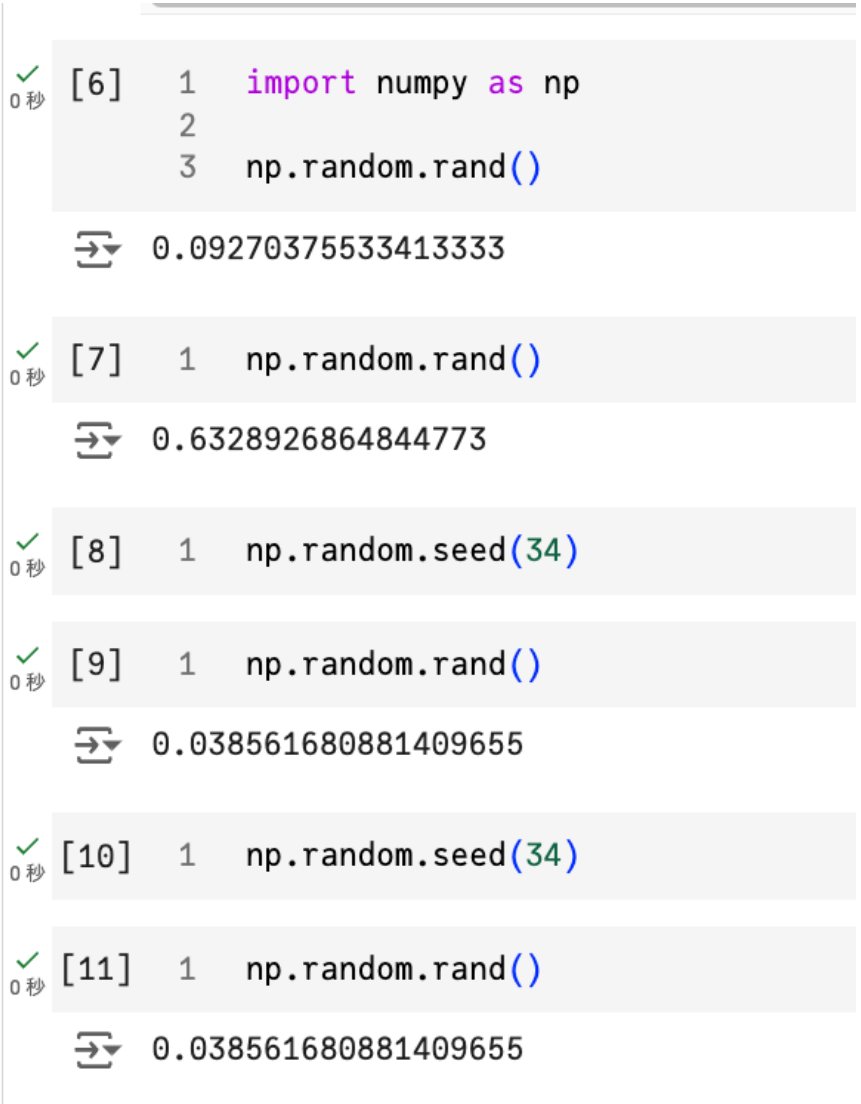
⇩

乱数シードを固定すると，

毎回同じ結果になって

再現性確保

実際はそんな素朴な世の中でもなく，環境差異であったり，並列処理をしたとき（とくに GPU が絡んだとき）には単に乱数シードを固定するような見た目のコードを書いても結果が変わりがちで，困ることが多いです．対処法もいろいろ考えられているので，気になる人は jax の乱数生成の仕組みなどを調べてみると面白いかもしれません。



1-2. データの前処理 - バリデーション

(`train_x` , `train_y`) を 学習データ:検証データ = 7:3 に分割

```
from sklearn.model_selection import train_test_split
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size=0.3, random_state=34)
```

結果を確認すると...

```
train_x.shape
```

```
val_x.shape
```

確かに 7:3 くらいに分割されていることがわかる

1-3. PyTorchに入力できる形に

✅ PyTorchで扱える形にする

1-3. PyTorchに入力できる形に

数として **Tensor型** を使って自動微分などを行う

```
>>> x = torch.tensor(2.0, requires_grad=True)
>>> def f(x):
...     return x ** 2 + 4 * x + 3
...
>>> y = f(x)
>>> y.backward()
>>> x.grad
tensor(8.)
```

($f(x) = x^2 + 4x + 3$ の $x = 2$ における微分係数 8)

➪ **データをTensor型に直しておく必要あり**

再掲: `Tensor` 型のつくりかた

`torch.tensor(data, requires_grad=False)`

- `data`: 保持するデータ(配列**っぽい**ものならなんでも)
  - リスト,タプル, **Numpy配列**, スカラ....
- `requires_grad`: 勾配 (gradient)を保持するかどうかのフラグ
  - デフォルトは `False`
  - 勾配の計算(自動微分)を行う場合は `True` にする
  - このあとこいつを微分の計算に使いますよ〜という表明

1-3. PyTorchに入力できる形に

⚠️ 我々が勾配降下法で使うのは,

各 パラメータ の損失に対する勾配

⇩

入力データの勾配は不要なので `requires_grad=True` とする必要はないことに注意！

1-3. PyTorchに入力できる形に

✅ **単にこれで OK！**

```
import torch

train_x = torch.tensor(train_x, dtype=torch.float32)
train_y = torch.tensor(train_y, dtype=torch.float32)
val_x = torch.tensor(val_x, dtype=torch.float32)
val_y = torch.tensor(val_y, dtype=torch.float32)
test_x = torch.tensor(test_x, dtype=torch.float32)
```

全体の流れ1 ~モデルに入力するまで

✅ 1-0. データのダウンロード

⇩

✅ 1-1. データの読み込み

⇩

✅ 1-2. データの前処理

⇩

✅ 1-2. PyTorchに入力できる形に

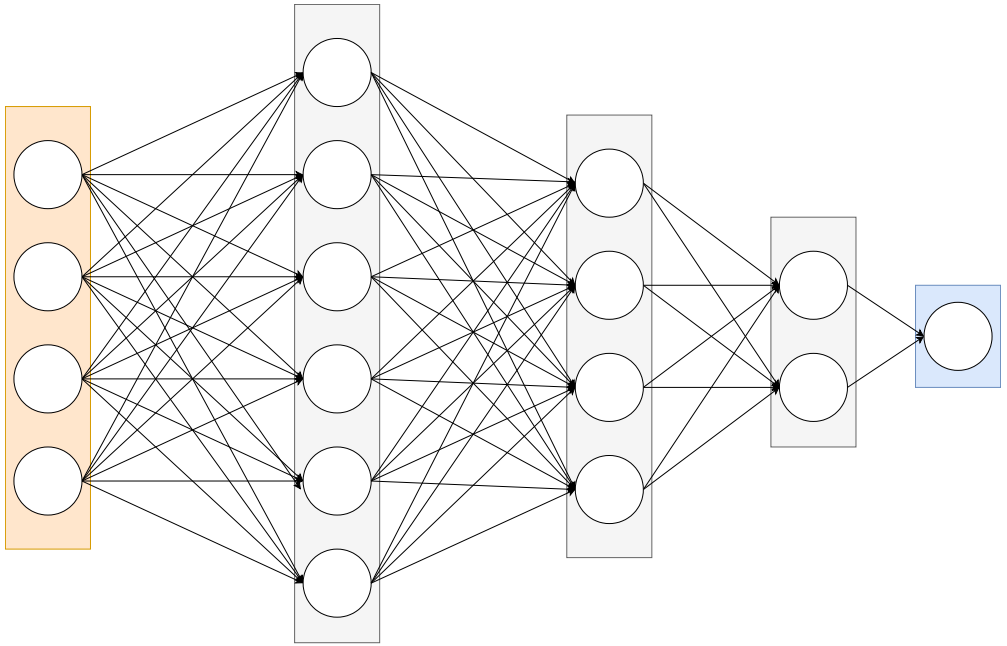
全体の流れ

1. データの読み込み
2. モデルの構築
3. モデルの学習
4. 新規データに対する予測
5. 順位表への提出

2. モデルの構築

今からすること...

$f(\boldsymbol{x};\boldsymbol{\theta})$ をつくる



2. モデルの構築

`torch.nn.Sequential` によるモデルの構築

✅ `torch.nn.Sequential` を使うと **一直線** のモデルを簡単に定義できる.

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(30, 32),
    nn.Sigmoid(),
    nn.Linear(32, 64),
    nn.Sigmoid(),
    nn.Linear(64, 1)
)
```

2. モデルの構築 ~ 二値分類の場合

二値分類の場合

➪ 最後に **シグモイド関数** をかけることで出力を $[0, 1]$ の中に収める.

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(30, 32),
    nn.Sigmoid(),
    nn.Linear(32, 64),
    nn.Sigmoid(),
    nn.Linear(64, 1),
    nn.Sigmoid() # <- ここ重要!
)
```

2. モデルの構築

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(30, 32),
    nn.Sigmoid(),
    nn.Linear(32, 64),
    nn.Sigmoid(),
    nn.Linear(64, 1),
    nn.Sigmoid()
)
```

⇨ すでにこの時点でパラメータの初期化などは終わっている

引数に層を順番に渡すことで,モデルを構築してくれる！

👈 「全結合層($W \in \mathbb{R}^{30,32}$) → シグモイド関数 → 全結合層 ($W \in \mathbb{R}^{32,64}$) → シグモイド関数 → 全結合層($W \in \mathbb{R}^{64,1}$)」という MLP の定義

2. モデルの構築

`model.parameters()` または
`model.state_dict()` で
モデルのパラメータを確認できる

```
model.state_dict()
```

各全結合層のパラメータ $W^{(i)}, \boldsymbol{b}^{(i)}$
が見える 👀 👉

```
0.038561680881409655
```

```
6秒  1  import torch.nn as nn
     2
     3  model = nn.Sequential(
     4      nn.Linear(16, 32),
     5      nn.Sigmoid(),
     6      nn.Linear(32, 64),
     7      nn.Sigmoid(),
     8      nn.Linear(64, 1)
     9  )
```

```
0秒 [13]  1  model.state_dict()
```

```
OrderedDict([('0.weight',
              tensor([[ 1.5935e-01,  1.1
                        7.4866e-02, -2.0
                        1.0282e-01,  4.6
                       -1.9607e-01],
                      [ 6.8658e-02, -2.4
                        2.4991e-02, -8.8
                       -9.1278e-02,  1.0
                        1.3355e-01],
                      [-1.5368e-01, -2.0
                       -1.1788e-01, -1.7
```

2. モデルの構築

✅ 構築したモデルは関数のように呼び出すことができる

```
import torch
dummy_input = torch.rand(1, 30)
model(dummy_input)
```

`torch.rand(shape)` で,形が `shape` のランダムな `Tensor` が作れる

➪ モデルに入力して計算できることを確認しておく！

(現段階では乱数でパラメータが初期化されたモデルに乱数を入力しているので値に意味はない)

2. モデルの構築

✅ $f(\boldsymbol{x};\boldsymbol{\theta})$ をつくる

⇩

あとはこれを勾配降下法の枠組みで学習させる！

⇩

思い出すシリーズ

確率的勾配降下法

全体の流れ

1. ✅ データの読み込み
2. ✅ モデルの構築
3. モデルの学習
4. 新規データに対する予測
5. 順位表への提出

全体の流れ3. モデルの学習

3-1. 確率的勾配降下法の準備

⇩

3-2. 確率的勾配降下法の実装

確率的勾配降下法

確率的勾配降下法 (SGD)

データの **一部** をランダムに選んで,
そのデータに対する勾配を使ってパラメータを更新する

3-1. 確率的勾配降下法の準備

整理: **我々がやらなきゃいけないこと**

👉 データをいい感じに選んで供給する仕組みを作る

3-1. 確率的勾配降下法の準備

 < 私がやります

✅ `torch.utils.data.Dataset`, `torch.utils.data.DataLoader` を

使うと簡単に実装できる！

3-1. 確率的勾配降下法の準備

現状確認☝️

`train_x`, `train_y`, `val_x`, `val_y`, `test_x` にデータが `Tensor` 型のオブジェクトとして格納されている.

3-1. 確率的勾配降下法の準備

1. Datasetの作成 (`Dataset`)

- データセット (データの入出力のペア $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$) を表すクラス

3-1. 確率的勾配降下法の準備

`TensorDataset` に

- モデルの入力データ (`train_x`)と
- 出力の目標データ (`train_y`) を渡すことで `Dataset` のサブクラスである `TensorDataset` が作れる！

```
from torch.utils.data import TensorDataset

# データセットの作成

# 学習データのデータセット
train_dataset = TensorDataset(train_x, train_y)
# 検証データのデータセット
val_dataset = TensorDataset(val_x, val_y)
```

実際は `torch.utils.data.Dataset` を継承したクラスを作ることでも `Dataset` のサブクラスのオブジェクトを作ることができます.
この方法だと非常に柔軟な処理が行えるためふつうはこれを使います (今回は簡単のために `TensorDataset` を使いました)

3-1. 確率的勾配降下法の準備

1. DataLoaderの作成 (`DataLoader`)

- `Dataset` から一部のデータ (ミニバッチ) を取り出して供給してくれるオブジェクト

つまり....

> 整理: **我々がやらなきゃいけないこと**
>
> 👉 データをいい感じに選んで供給する仕組みを作る

をやってくれる

3-1. 確率的勾配降下法の準備

1. DataLoaderの作成（`DataLoader`）

- `Dataset`からミニバッチを取り出して供給してくれるオブジェクト

`DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)`

```
from torch.utils.data import DataLoader

batch_size = 32
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

⇨ これを for文で回すことでデータを取り出すことができる

3-1. 確率的勾配降下法の準備

1. DataLoaderの作成(`DataLoader`型)

```
for inputs, targets in train_dataloader:
    print('inputs.shape', inputs.shape)
    print('targets.shape', targets.shape)
    print('-------------')
```

⇩

```
inputs.shape torch.Size([32, 30])
targets.shape torch.Size([32, 1])
-------------
inputs.shape torch.Size([32, 30])
targets.shape torch.Size([32, 1])
...
```

✔ データセットを一回走査するまでループが回ることを確認しよう！

3-1. 確率的勾配降下法の準備

✅ DatasetとDataLoaderの作成

```
from torch.utils.data import TensorDataset, DataLoader

# データセットの作成
train_dataset = TensorDataset(train_x, train_y)
val_dataset = TensorDataset(val_x, val_y)

# データローダの作成
batch_size = 32
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

3-1. 確率的勾配降下法の準備

整理: **我々がやらなきゃいけないこと**

👉 データをいい感じに選んで供給する仕組みを作る

✅ **Done!**

3.2 確率的勾配降下法の実装

✅ データは回るようになった

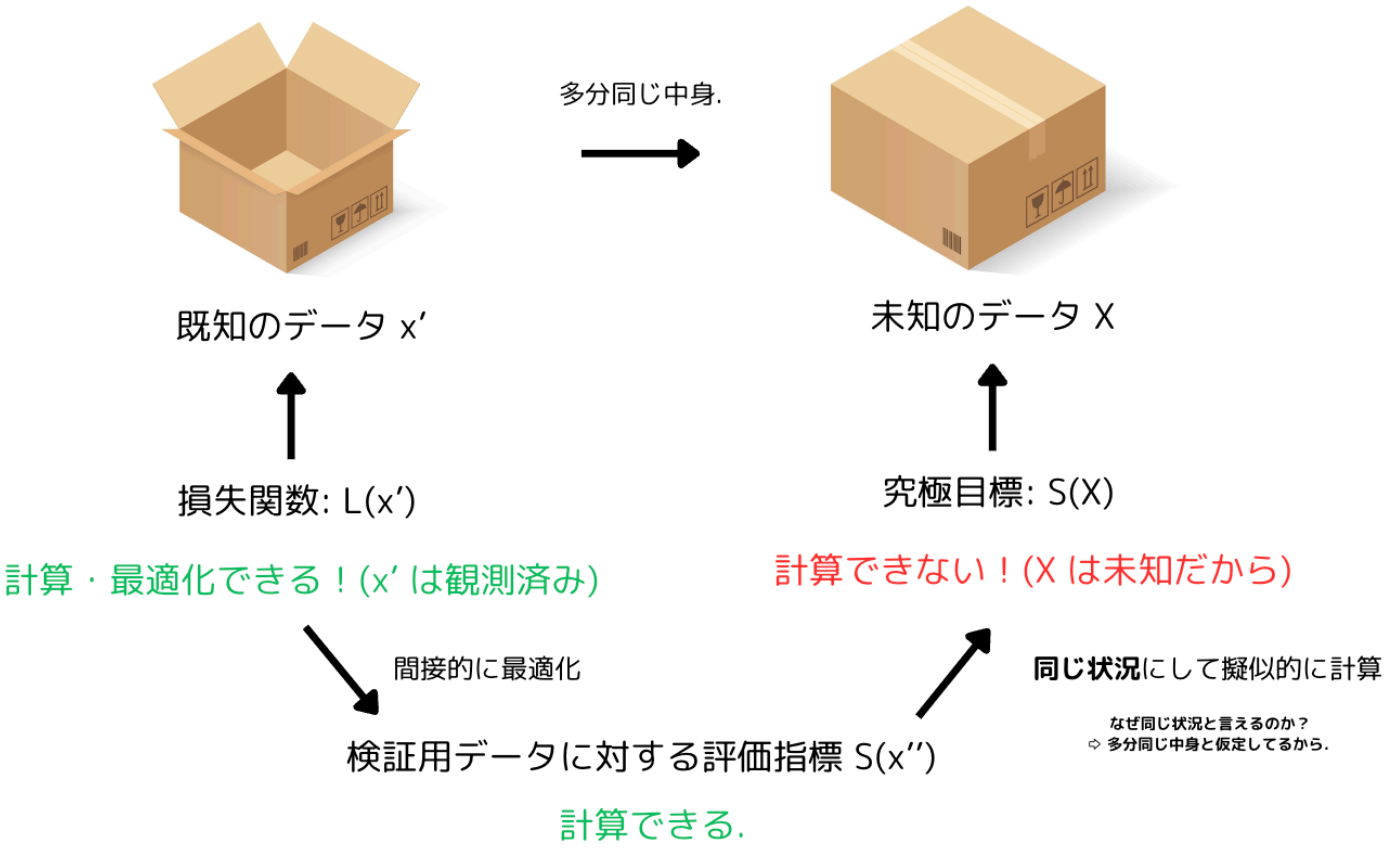
➪ あとは学習を実装すればOK！

TODOリスト

1. 損失関数を設定する
2. 勾配の計算を行う
3. パラメータの更新を行う

3.2 確率的勾配降下法の実装: 損失関数の設定

1. 損失関数は何のためにあるのか？



3.2 確率的勾配降下法の実装: 損失関数の設定

今回の評価指標 👉 **正解率！**

3.2 確率的勾配降下法の実装: 損失関数の設定

今までは評価指標もすべて平均二乗和誤差だった

⇩

平均二乗誤差は微分可能なのでこれを **損失関数** として勾配降下法で最適化すれば

評価指標である 平均二乗誤差も最適化できた

3.2 確率的勾配降下法の実装: 損失関数の設定

正解率は直接最適化できる？

⇨ **No!!**

正解率の微分

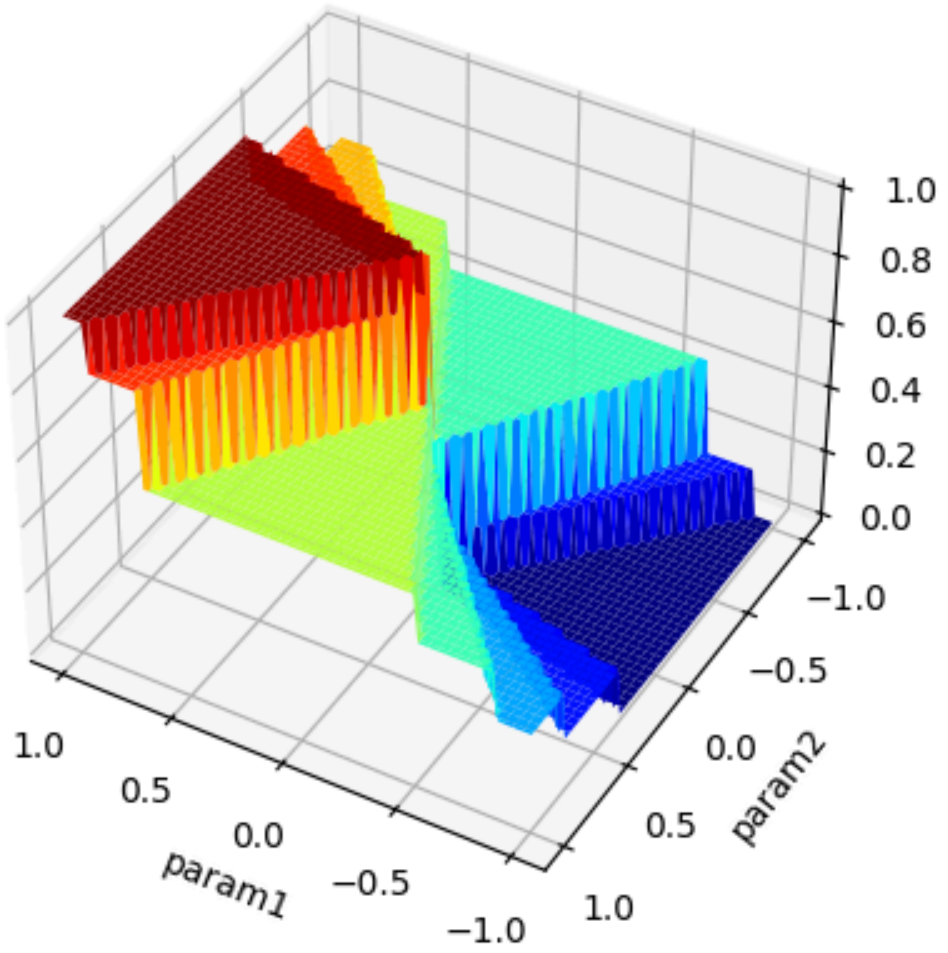
パラメータを微小に変化させても
正解率は変化しない！

⇨ 正解率は,

- **ほとんどの点で微分係数 0**
- **変わるところも微分不可能**

⇩

勾配降下法で最適化できない



右のグラフは, 適当に作った二値分類 ($\mathbb{R}^2 \to \{0, 1\}$) のタスクをロジスティック回帰というモデルで解いたときの、パラメータ平面上の正解率をプロットしてみたものです。これを見ればほとんどのところが微分係数が 0 ($\leftrightarrow$ 平坦) で、変わるところも微分不可 ($\leftrightarrow$ 鋭い) ことがわかります。

正解率を間接的に最適化する

どうするか？

⇨ こういう分類を解くのに向いている損失関数を使って **間接的に** 正解率を上げる.

Binary Cross Entropy Loss

二値交差エントロピー誤差 (Binary Cross Entropy Loss)

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \log(f(x_i)) + (1-y_i)\log(1-f(x_i))$$

Binary Cross Entropy Loss

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \log(f(x_i)) + (1-y_i)\log(1-f(x_i))$$

確認してほしいこと:

- 正解 y_i と予測 $f(x_i)$ が近いほど値は小さくなっている.
  ($y_i \in \{0,1\}$　なのでそれぞれの場合について考えてみるとわかる)
- 微分可能である

👉 **なので、損失関数として妥当**

これもやはり二乗和誤差のときと同様に同様に尤度の最大化として **導出** できます.

Binary Cross Entropy Loss

✅ PyTorch では，`torch.nn.BCELoss` で使える！

```
import torch

criterion = torch.nn.BCELoss()

y = torch.tensor([0.0, 1.0, 1.0])
pred = torch.tensor([0.1, 0.9, 0.2])

loss = criterion(pred, y)
print(loss)   # ⇒ tensor(0.6067)
```

3.2 確率的勾配降下法の実装

TODOリスト

- ☑ 1. 損失関数を設定する
- 2. 勾配の計算を行う
- 3. パラメータの更新を行う

2. 勾配の計算を行う

やりかたは....？

3.2 確率的勾配降下法の実装

定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義
定義→計算→backward(), 定義→計算→backward(), 定義→計算→backward(), 定義

損失に対するパラメータの勾配の計算例

```
# ここから
model = nn.Sequential(
    nn.Linear(30, 32),
    ...
)
# ここまでが "定義"

dummy_input = torch.rand(1, 30)
dummy_target = torch.rand(1, 1)

# "計算"
pred = model(dummy_input)
loss = criterion(pred, dummy_target)

# "backward()"
loss.backward()
```

3.2 確率的勾配降下法の実装

✔ チェックポイント

1. `loss` に対する勾配を計算している

```
# backward
loss.backward()
```

2. 勾配は **パラメータ** に対して計算される

```
for param in model.parameters():
    print(param.grad)
```

(`dummy_input` , `dummy_target` は `requires_grad=False` なので勾配は計算されない)

3.2 確率的勾配降下法の実装

TODOリスト

- ☑ 1. 損失関数を設定する
- ☑ 2. 勾配の計算を行う
- 3. パラメータの更新を行う

3.2 確率的勾配降下法の実装

```
for epoch in range(epochs):
    for inputs, targets in train_dataloader:
        # 計算
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        # backward
        loss.backward()

        # ----------------------
        # ....
        # ここにパラメータの更新を書く
        # ....
        # ----------------------
```

3.2 確率的勾配降下法の実装

これまでは,我々が手動(?)で更新するコードを書いていた

< 私がやります

✅ **torch.optimのオプティマイザを使うことで簡単にいろいろな最適化アルゴリズムを使える**

3.2 確率的勾配降下法の実装

(⚠️: 完成版ではない)

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

# 学習ループ
for epoch in range(epochs):
    for inputs, targets in train_dataloader:
        # 勾配の初期化
        optimizer.zero_grad()
        # 計算
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        # backward
        loss.backward()

        # パラメータの更新
        optimizer.step()
```

3.2 確率的勾配降下法の実装

✅ `optimizer = optim.SGD(params)` **のようにすることで**
`params` **を勾配降下法で更新するオプティマイザを作成できる！**

たとえば Adam が使いたければ `optimizer = optim.Adam(params)` とするだけでOK！

⇩

勾配を計算したあとに `optimizer.step()` を呼ぶと，
各 `Tensor` に載っている勾配の値を使ってパラメータを更新してくれる

3.2 確率的勾配降下法の実装

⚠️ **注意** ⚠️

`optimizer.step()` で一回パラメータを更新するたびに
`optimizer.zero_grad()` で勾配を初期化する必要がある！

(これをしないと前回の `backward` の結果が残っていておかしくなる)

⇩ 次のページ...

学習の全体像を貼ります！！！

3.2 確率的勾配降下法の実装

```
from torch import nn


model = nn.Sequential(
    nn.Linear(30, 32),
    nn.Sigmoid(),
    nn.Linear(32, 64),
    nn.Sigmoid(),
    nn.Linear(64, 1),
    nn.Sigmoid()
)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
criterion = torch.nn.BCELoss()

n_epoch = 100
for epoch in range(n_epoch):
    running_loss = 0.0

    for inputs, targets in train_dataloader:
        # 前の勾配を消す
        optimizer.zero_grad()

        # 計算
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        # backwardで勾配を計算
        loss.backward()

        # optimizerを使ってパラメータを更新
        optimizer.step()

        running_loss += loss.item()

    val_loss = 0.0
    with torch.no_grad():
        for inputs, targets in val_dataloader:
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            val_loss += loss.item()

    # エポックごとの損失の表示
    train_loss = running_loss / len(train_dataloader)
    val_loss = val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1} - Train Loss: {train_loss:.4f} - Val Loss: {val_loss:.10f}')
```

各行の解説 (for文以降)

- 1行目. `for epoch in range(n_epoch)` .... データ全体を `n_epoch` 回まわす
- 2行目. `running_loss = 0.0` 1エポックごとの訓練データの損失を計算するための変数
- 4行目. `for inputs, targets in train_dataloader` .... 訓練データを1バッチずつ取り出す(`DataLoader` の項を参照してください！)
- 6行目. `optimizer.zero_grad()` .... 勾配を初期化する. 二つ前のページのスライドです！
- 9, 10行目. `outputs = ...` .... 損失の計算をします.

3.2 確率的勾配降下法の実装

- 13行目. `loss.backward()` .... 勾配の計算です.これによって`model`のパラメータに**損失に対する** 勾配が記録されます
- 16行目. `optimizer.step()` .... `optimizer`が記録された勾配に基づいてパラメータを更新します.
- 18行目. `running_loss += loss.item()` .... 1バッチ分の損失を`running_loss`に足しておきます.
- 20行目~25行目. 1エポック分の学習が終わったらバリデーションデータでの損失を計算します. バリデーションデータの内容は学習に影響させないので勾配を計算する必要がありません.したがって`torch.no_grad()`の中で計算します.

3.2 確率的勾配降下法の実装

- 28行目〜30行目. 1エポック分の学習が終わったら, 訓練データと検証データの損失を表示します. `len(train_dataloader)` は訓練データが何個のミニバッチに分割されたかを表す数, `len(val_dataloader)` は検証データが何個のミニバッチに分割されたかを表す数です. これで割って平均の値にします.
- 32行目. 損失を出力します.

3.2 確率的勾配降下法の実装

TODOリスト

- ☑ 1. 損失関数を設定する
- ☑ 2. 勾配の計算を行う
- ☑ 3. パラメータの更新を行う

バリデーション

バリデーションデータで 今回の評価指標である正解率がどのくらいになっているか計算しておく！

👉 これがテストデータに対する予測精度のめやす.

正解率の計算

1. 0.5 以上なら異常と予測する.

```
val_pred = model(val_x) > 0.5
```

2. `torch.Tensor` から `numpy.ndarray` に変換する

```
val_pred_np = val_pred.numpy().astype(int)
val_y_np = val_y.numpy().astype(int)
```

2. `sklearn.metrics` の `accuracy_score` を使って正解率を計算する

```
from sklearn.metrics import accuracy_score
accuracy_score(val_y_np, val_pred_np) # ⇒ (乞うご期待. これを高くできるように頑張る)
```

3. 学習が完了！！！

+ オプション　学習曲線を書いておこう

1. 各エポックの損失を記録する配列を作っておく

```
train_losses = []
val_losses = []
```

1. 先ほどの学習のコードの中に,損失を記録するコードを追加する

```
train_loss = running_loss / len(train_dataloader)
val_loss = val_loss / len(val_dataloader)
train_losses.append(train_loss) # これが追加された
val_losses.append(val_loss)  # これが追加された
print(f'Epoch {epoch + 1} - Train Loss: {train_loss:.4f} - Val Loss: {val_loss:.10f}')
```

(各 エポックで正解率も計算するとより実験がしやすくなるので実装してみよう)

3. 学習が完了！！！

+ オプション　学習曲線を書いておこう

`matplotlib` というパッケージを使うことでグラフが書ける

```
# matplotlib.pyplot を pltという名前でimport
import matplotlib.pyplot as plt
```

```
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.legend()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

➪ いい感じのプロットが見れる

全体の流れ

1. データの読み込み
2. モデルの構築
3. モデルの学習
4. 新規データに対する予測
5. 順位表への提出

4. 新規データに対する予測

そういえば 💡

`test_x` に予測したい未知のデータが入っている

```
model(test_x)
```

➪ 予測結果が出る

5. 順位表への提出

```
import csv

def write_pred(predictions, filename='submit.csv'):
    pred = predictions.squeeze().tolist()
    assert set(pred) == set([True, False])
    pred_class =  ["attack" if x else "normal" for x in pred]
    sample_submission = pd.read_csv('sample_submission.csv')
    sample_submission['pred'] = pred_class
    sample_submission.to_csv('submit.csv', index=False)
```

をコピペ

→

5. 順位表への提出

予測結果（`True`, `False` からなる `Tensor`）

```
pred = model(test_x) > 0.5
```

を作って,

```
write_pred(pred)
```

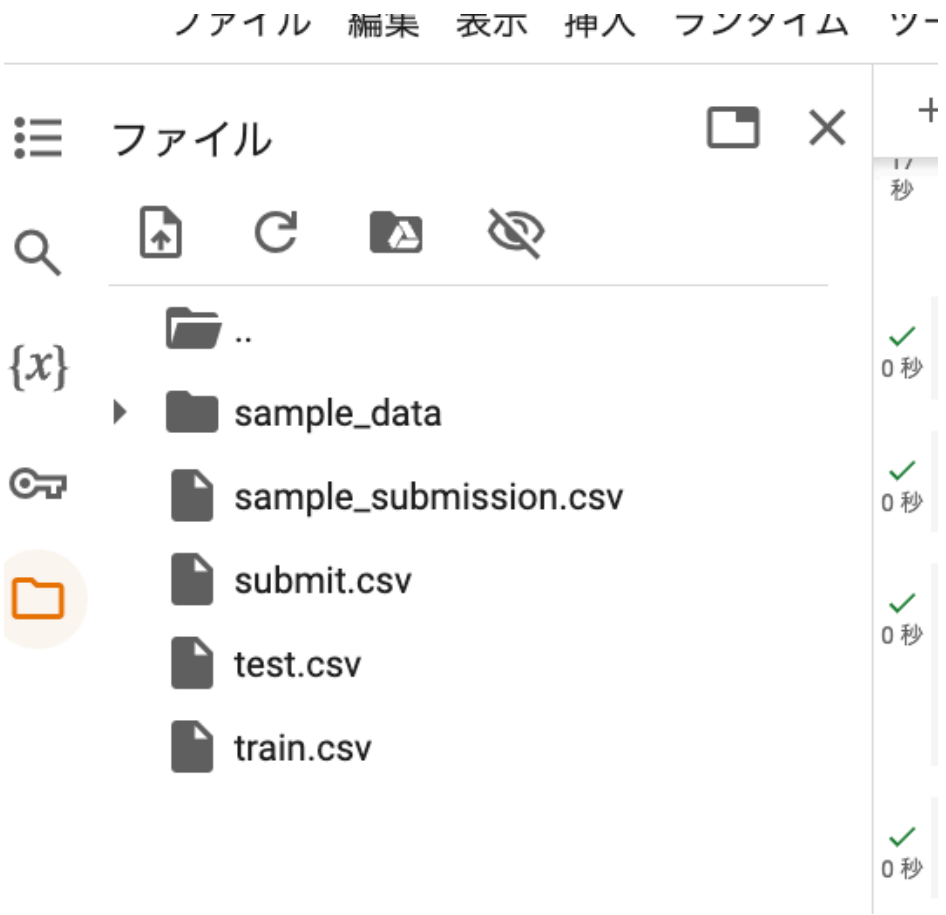
すると,

5. 順位表への提出

📂 > submit.csv

ができる！

👉 ダウンロードして, submit から投稿！ **順位表に乗ろう!**



5. 順位表への提出

めざせ　No.1！

機械学習講習会 第七回

- 「機械学習の応用,データ分析コンペ」

traP Kaggle班

2024/07/17

今日の内容

- **コンペの結果発表** 🥳
- データ分析コンペという競技について
- ポエム

コンペの結果発表 

⇨ https://abap34.github.io/ml-lecture/supplement/competetion-result.html

データ分析コンペという競技について

Q. 今回のコンペでどんな取り組み方をしましたか？

コンペの戦い方 1.EDA

✅ データ分析コンペにおける勝敗を分けるポイントのひとつ

⇨ データへの 理解度

コンペの戦い方 1.EDA

あたり前に確認すべきこと...

1. データはどのくらいあるのか？
2. どういう形式なのか？

+ **どのような情報が予測に役立つのか？**

EDA: 探索的データ分析 (Exploratory Data Analysis)

EDA: 探索的データ分析 (Exploratory Data Analysis)

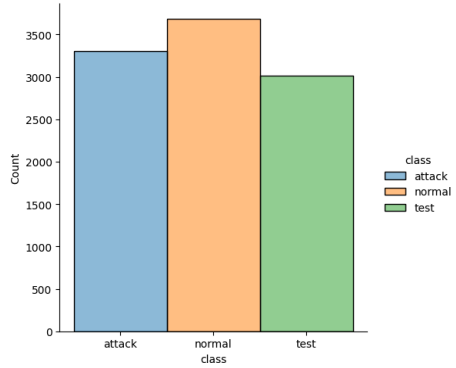
事前に仮説やモデルを仮定せず,データの特徴や構造を理解する分析.

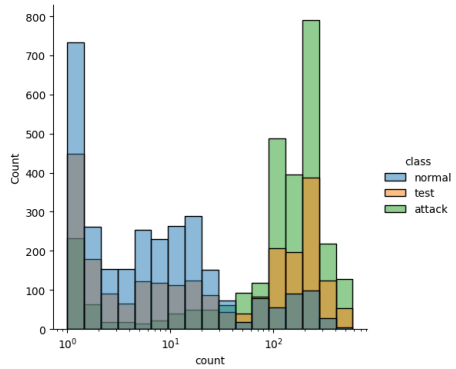
例) データの分布,欠損値の確認,各変数の組の相関係数　などなど...

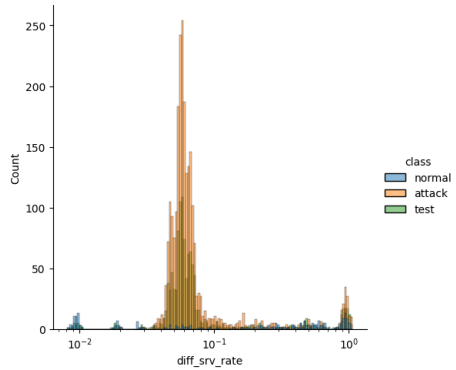
EDA: 今回のコンペを例に

ものすごく簡単な例:
(abap34.com/ml-lecture/supplement/EDA.html)

```
In [11]: for col in numeric_columns:
             sns.displot(data=all_df.sample(10000), x=col, hue='class', legend=col, log_scale=True)
             plt.show()
```







コンペの戦い方 2.バリデーションについて

Trust Your CV ... Cross Validation CV を信じよという有名な信仰.

バリデーション

Q. Public LeaderBoard に大量の提出を繰り返すとどうなる？

⇨ Public LB でのスコアが上振れる.

Q. するとどうなる？

⇨ **shake で死ぬ.**

shake

Public LB と Private LB の順位が大きく異なる現象

919	An Dang-Hieu		0.83320	2
726	Jura Moshkov		0.83320	57
462	Bohao XU		0.83320	7
311	按59		0.83320	41
69	Wannabeee		0.83319	29
353	Zaakcii Ru		0.83319	243
495	fanaoyu		0.83314	22
874	yhataktaro		0.83312	10
708	stakahashi		0.83310	81
419	yukiZ		0.83310	397
74	Aindriú		0.83306	172
392	Yuki1213ya		0.83305	8

写真はつい先日終わった Learning Agency Lab - Automated Essay Scoring 2.0 というコンペの順位表です. こちらのリンク (https://kaggle.com/competitions/learning-agency-lab-automated-essay-scoring-2/leaderboard) から見れます.恐怖.

shake の波を乗り切るにはどうするか？

✅ **Public LB に振り回されないために**

1. スコアのブレの程度を把握しておく
    i. テストと同じくらいのサイズのバリデーションデータをとり,そのスコアのブレを見るなど
    ii. **Public Score の上振れを引いても Private Score は上がらないので CV を上げることに専念**
2. バリデーションデータとテストデータの分布の乖離に気を付ける
    i. たいていのコンペでは参加者同士が CV と LB のスコアを比較するディスカッションが立っていがち. **これを必ず 確認する！**
    ii. 分布の違いの原因を調べて, よりテストデータに近いバリデーションデータを作る方法を考える (例: adversarial validation)

Public LB との向き合い方

ただ, Public LB も **重要な情報**

👀 **(ふつうの) 機械学習の枠組みでは絶対見られないテストスコアの一部が見られる**

⇩　以下のケースでは Public LB も **重要なスコアの指針**

1. **Public LB 用のデータが学習データと同じ分布で同程度のサイズ**
2. **時系列で学習データとテストデータが分割されている**
    i. **Public / Private 間はランダムに分割 ← とくに重要な指針になる**
    ii. **Public / Private も時系列で分割**

コンペの戦い方 3. ハイパーパラメータの調整

ハイパーパラメータ(学習率, 木の深さ, ... などの学習時の設定) の調整は大事！
だけど

⚠ 最初からハイパーパラメータの調整に時間をかけすぎない ⚠

ハイパーパラメータの調整

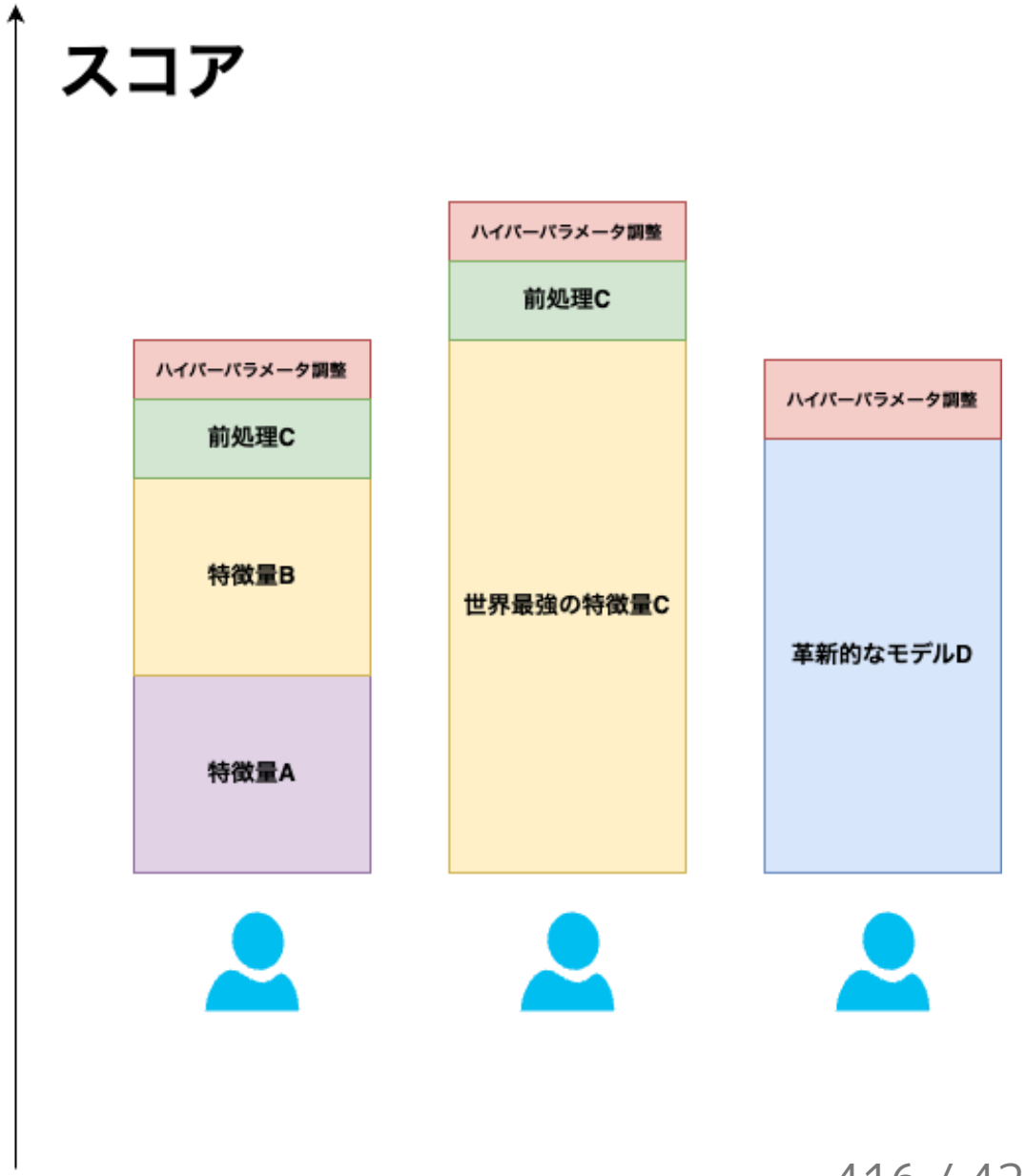
✅ **ハイパーパラメータの調整は決定的な差別化ポイントになりづらい！**

➪ 調整はそこそこに

- データの理解
- 特徴量エンジニアリング

に時間を費やすのが 🦁

(もちろん, 確実にスコアを上げられる手段なので**終盤にはちゃんと調整**する)



テーブルコンペの全体的な流れ

1. まず与えられたデータに対して EDA を行い, データの基本的な性質や予測に役立つ情報を把握する
2. 信頼できるバリデーションの仕組みを構築する
3. 特徴量エンジニアリングを行い, 学習
4. 提出
5. ディスカッションを参考にしつつ, スコアの信頼性などを確かめる.終盤ならハイパーパラメータの調整などをしても良いかも.
6. 3 に戻る↩

ポエム

- この講習会で扱わなかったこと

この講習会で扱わなかったこと

> **この講習会は機械学習の洞窟を全て探検することを目指しているのではなく、一旦ガイド付きで洞窟の最深部まで一気に駆け抜けることで二回目以降の探検をしやすくすることを目指しています。**

（前がきより）

この講習会で扱わなかったこと

✅ **機械学習の世界はめちゃくちゃ広い！**

関連する

- 数学
- コンピュータサイエンス

の話題もたくさん (本当にたくさん)

解ける面白い問題もたくさん！

➪ **必ず興味があるものに遭遇するはず!**

➪ Kaggle 班で色々やりましょう！お疲れ様でした！