

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 アルゴリズム班 Kaggle部

2023/7/3

振り返り

第一回: 学習

第二回: 勾配降下法

第三回: 自動微分

第四回: ニューラルネットワークの構造

第五回: ニューラルネットワークの学習と評価

第六回: ニューラルネットワークの実装

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！！

コンペについて ~ 基本情報 ~

- お題: このあと発表
- 開始: 7/09 18:00~ よりサブミット可能
- 終了: 7/16 23:59:59 (JST)
- サイト: ddaq-v2.trap.show

※  注意事項があります！この後ちゃんと読んでください！

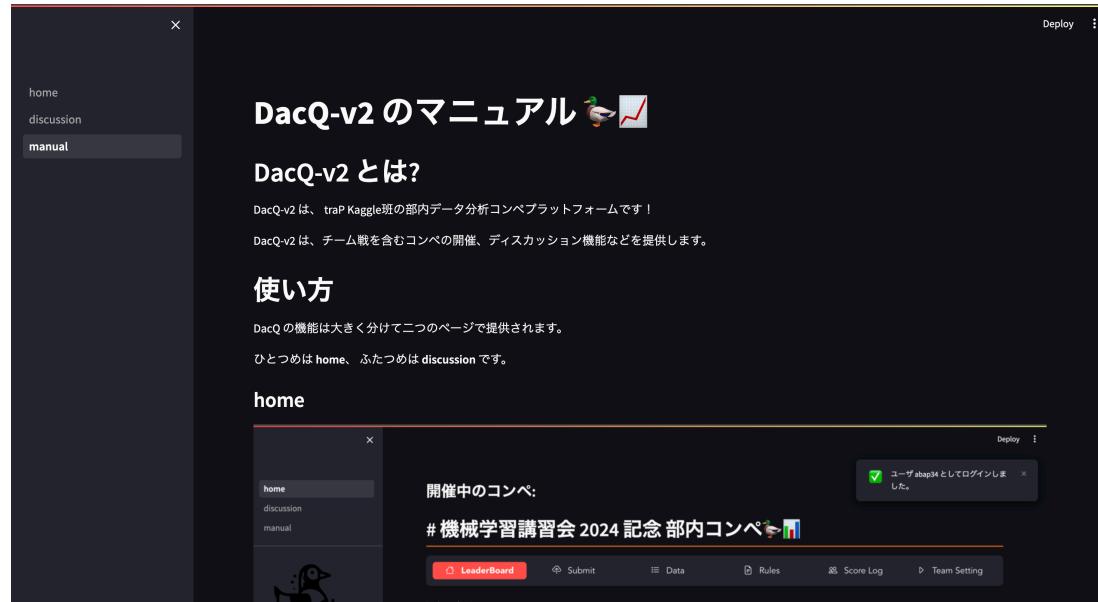
コンペについて : DacQ

The screenshot shows the DacQ competition platform's Team Setting page. At the top, it displays the current competition: "# 機械学習講習会 2024 記念 部内コンペ". The navigation bar includes links for LeaderBoard, Submit, Data, Rules, Score Log, and Team Setting (which is highlighted in red). Below the navigation, it shows the user is logged in as "abap34" and is part of "Team: team 41". The main section is titled "Team Setting" and shows the "Team Name: team 41". It features a "Current Icon" placeholder and a field to "Enter your team name" with "team 41" typed in. To the right, there is a "Members" table listing three users: abap34, noya2, and ebi, each with their respective profile icons.



< 部内データ分析コンペプラットフォームです.

DacQ の使い方: Manual ページ



< バグ報告待ってます. (@abap34 の DM までお願いします)

稼働状況やバグのアナウンスは #event/workshop/machine-learning/system-announcement で

Private LB と Public LB

⚠️ コンペ期間中に見える LB は **Public LB** とよばれる **暫定スコア** です.



Private LB と Public LB

期間中に見えるスコア = 最終的なスコア の場合...

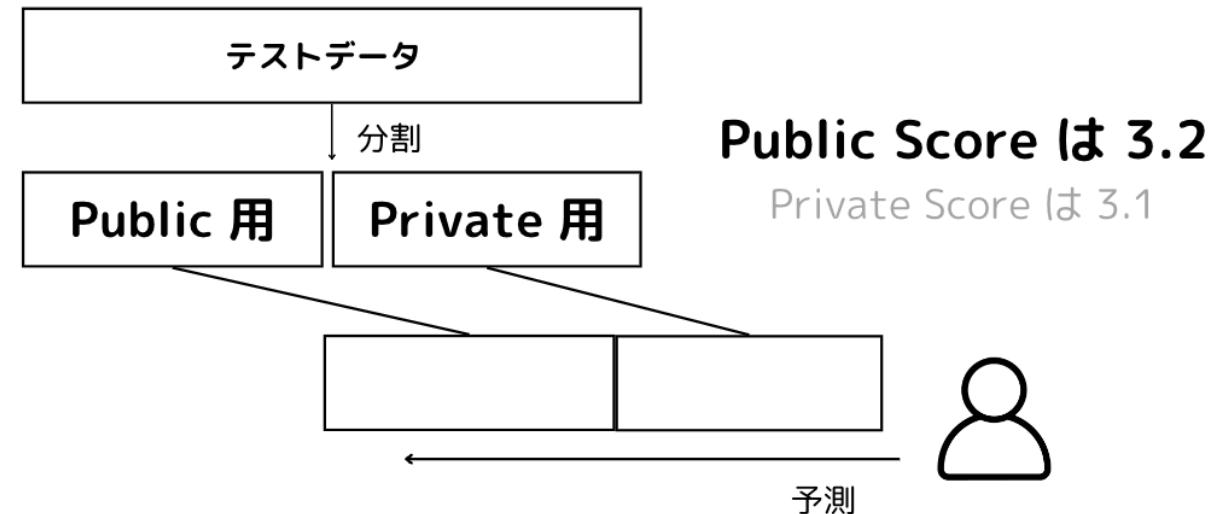
- 各行の値を変えたときのスコアを見ることで値を改善できてしまい不毛
- 上振れを引くために、サブミットしまくるのが最適戦略になってしまい不毛



コンペ中は暫定スコアを表示して、
最終結果は競技者が選んだサブミットに対して 裏で計算したスコア を使う

Private / Public LB まとめ

- 運営があらかじめデータを
Public 用 / Private 用に分割
 - ↓
 - それぞれに対してスコアを計算し、
Public Score だけが期間中見られる
 - ↓
 - サブミットのうち数個を競技者が
期間中に選んでおき、
その Private Score で順位が決定



DacQ における Private / Public LB

重要: DacQ では、最終スコアとして最後の 2つのサブミットが使われます。

スコアの計算例

投稿日	Public Score	Private Score
7/17	0.7	0.9
7/16	0.9	0.8
7/14	0.99	0.95
7/13	0.6	0.7
7/12	0.5	0.6

の場合、 Public LB の値は

0.5→0.6→0.99→0.99→0.99

Private LB に乗る値は

0.9

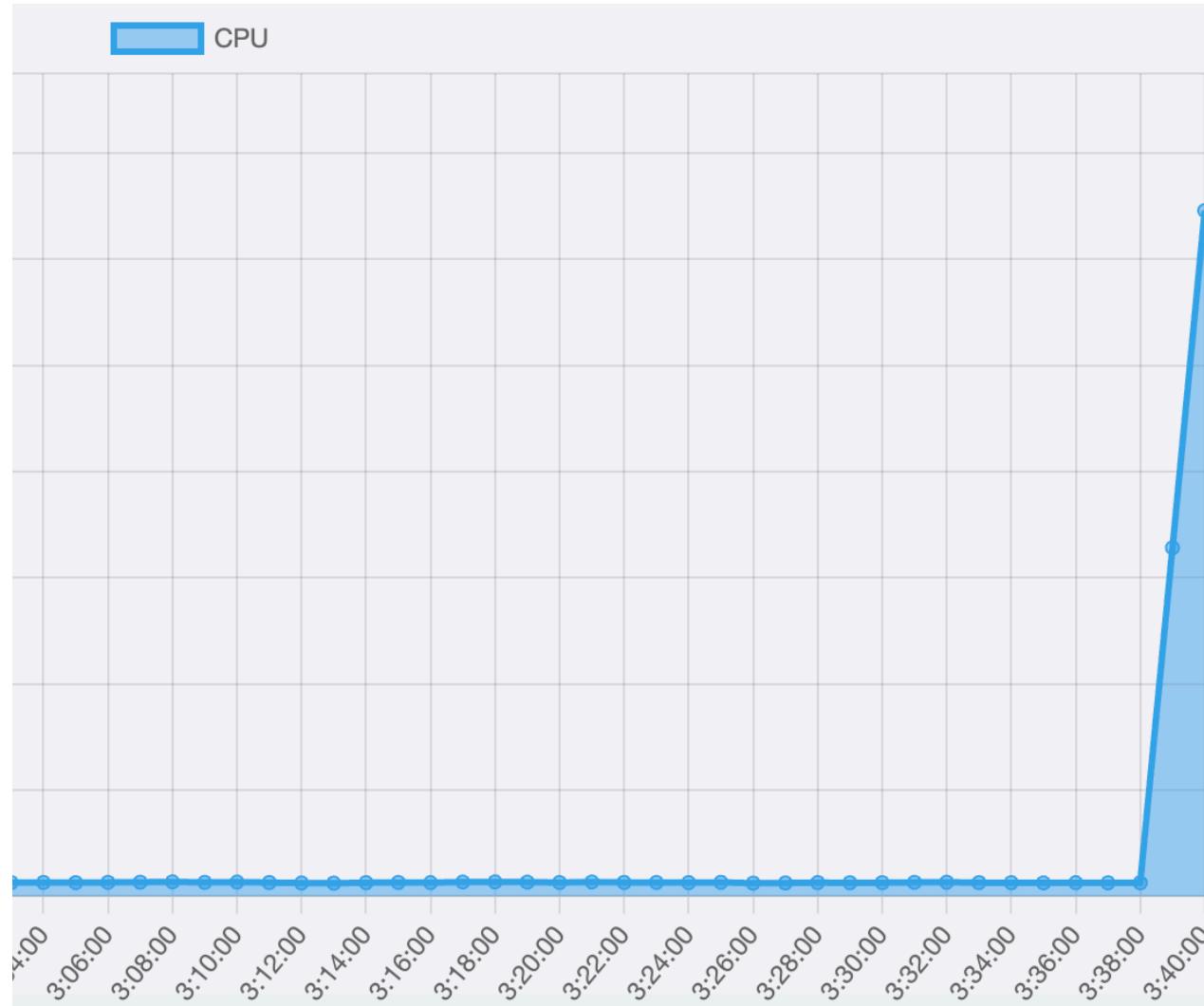
✳ 注意: 自信のある投稿を最後に再投稿することを忘れずに!!!!!! ✳

投稿に関する特別なルール

終了直前は、混雑によって
投稿ができない可能性があります



- なるべく余裕を持って提出してください
- 次善策として、終了 3 時間前からに限り、**終了時刻前に @abap34 のDMへ送信されたものも提出として認めます**



注意

.....

■最善を期しているつもりですが、不具合によって投稿データなどが失われる可能性があります。投稿ファイルは忘れずに、必ず手元にとっておいてください。 ■

そのほかの細かいルールや注意事項について

Rules タブを、絶対に読んでください！！！
不明なことをしたくなったら @abap34 まで必ず連絡してください！

開催中のコンペ:

機械学習講習会 2024 記念 部内コンペ

[LeaderBoard](#) [Submit](#) [Data](#) [Rules](#) [Score Log](#) [Team Setting](#)

Login as abap34
Team: team 41
[Rules](#)

Description

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、飛んでくる通信について、機械学習を使って攻撃を試みる通信かどう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。ご協力ありがとうございます。

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用います)
 - 学習に使うデータの一部
- test.csv
 - 予測対象のデータ
- test-filna.csv ( 時間と説明の都合上今日はこれを用います)
 - 予測対象のデータの欠損値(※後述)を埋めたもの
- 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



1-3. 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
```

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

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



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** ml-lecture-2024_6
- File Explorer:** Shows files: sample_data, sample_submission.csv, test.csv, and train.csv.
- Code Cell 1:** Contains three curl commands to download CSV files:

```
1 !curl https://www.abap34.com/trap_ml_lecture/public-data/train_tiny.csv -o train.csv
2 !curl https://www.abap34.com/trap_ml_lecture/public-data/test_tiny.csv -o test.csv
3 !curl https://www.abap34.com/trap_ml_lecture/public-data/sample_submission.csv -o sample_submission.csv
```
- Code Cell 2:** Imports pandas and reads the CSV files:

```
[2] 1 import pandas as pd
2
3 train = pd.read_csv("train.csv")
4 test = pd.read_csv('test.csv')
```
- Code Cell 3:** Starts with `train_y = train['class'].map({`.
- Output Area:** Shows the contents of `train.csv` as a table:

id	duration	src_bytes	ds
98643	0.0	0.0	0.0
40263	0.0	0.0	0.0
47961	0.0	6.0	0.0
37672	0.0	166.0	22
112203	0.0	0.0	0.0
56047	0.0	317.0	71
123632	0.0	145.0	36
65616	0.0	35.0	0.0
33403	0.0	0.0	0.0
20170	0.0	0.0	0.0

Show 10 per page

1-1. データの読み込み

✓ `pd.read_csv(path)` で、`path` にあるcsvファイルを読み込む

```
# 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")
```

パスとは、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	rror_rate	rv_rrror_rate	same_srv_rate	diff_srv_rate	rv_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）
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']
```

と入力して実行

⇒ **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 **さっきの列を除いたもの！**
(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

セルに、

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

と入力して実行

⇒ `train_x` にさっきの列を除いたデータが入る

[36] 1 train_x

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

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可能

(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、平均を引くなど)

• • •

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. "Efficient 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 = scaler.transform(test)
```

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

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に二次元配列 (`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

なので、`train_y` もここで中身を取り出して二次元配列にしておく。

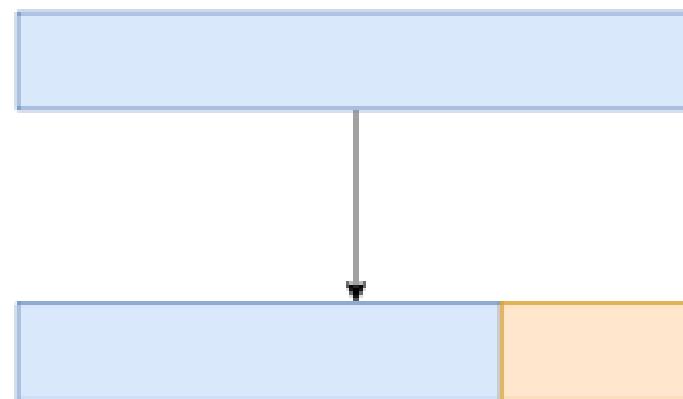
1. `train_y.values` で 中身の値を取り出せる。
2. `arr.reshape(-1, 1)` で `arr` を $N \times 1$ の形に変換できる

```
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。`-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 してくれます。

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まあ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

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)
```

1-2. データの前処理 - バリデーション

train_test_split 関数による分割

```
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` : 亂数のシード ➡️重要！！

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



再現性が確保 😊

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、並列計算、とくに GPU が絡んでくると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実行毎に結果が変わりがちで、人は苦しんでいます。

```
✓ [6] 1 import numpy as np  
2  
3 np.random.rand()
```

```
↳ 0.09270375533413333
```

```
✓ [7] 1 np.random.rand()
```

```
↳ 0.6328926864844773
```

```
✓ [8] 1 np.random.seed(34)
```

```
✓ [9] 1 np.random.rand()
```

```
↳ 0.038561680881409655
```

```
✓ [10] 1 np.random.seed(34)
```

```
✓ [11] 1 np.random.rand()
```

```
↳ 0.038561680881409655
```

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

- ✓ このあとこれらをPyTorchで扱うので、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

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)
train_y = torch.tensor(train_y)
val_x = torch.tensor(val_x)
val_y = torch.tensor(val_y)
test = torch.tensor(test)
```

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



✓ 1-3. PyTorchに入力できる形に

全体の流れ

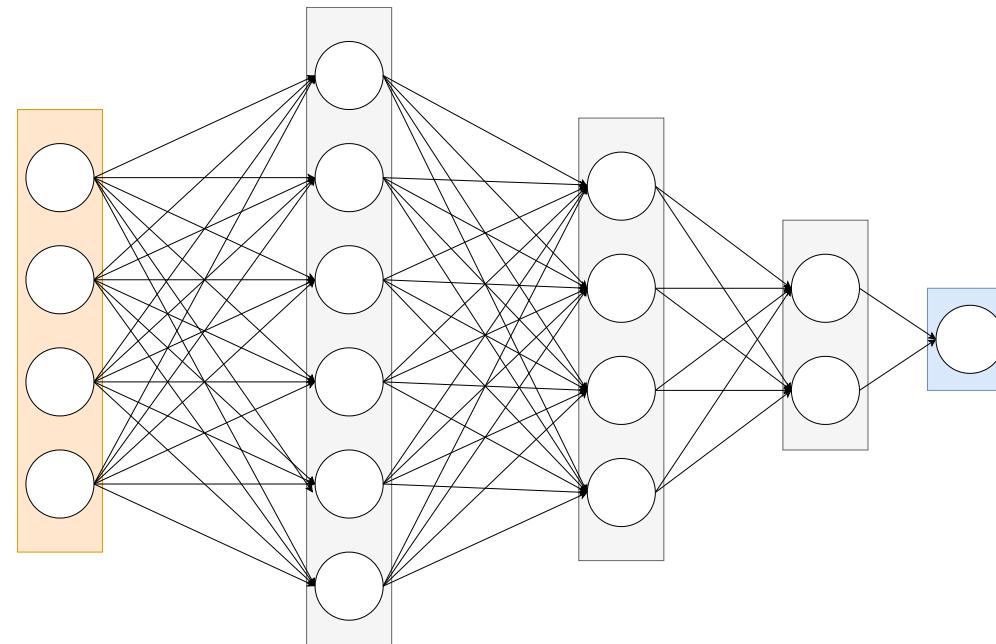
.....

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...

$f(x; \theta)$ 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)
)
```

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

⇒ 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

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, 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 )
```

✓ [13] 1 model.state_dict()

➡️ `OrderedDict([('0.weight', tensor([[1.5935e-01, 1.1
7.4866e-02, -2.6
1.0282e-01, 4.6
-1.9607e-01],
[6.8658e-02, -2.4
2.4991e-02, -8.8
-9.1278e-02, 1.6
1.3355e-01],
[-1.5368e-01, -2.6
-1.1788e-01, -1.7
1.5117e-01, -1.2`

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みる！

(現段階では、乱数で初期化されたモデルに乱数を入力して、謎の出力が得られる(?))

2. モデルの構築

✓ $f(x; \theta)$ 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

➊ <私がやろう

✓ `torch.utils.data.Dataset` と `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` にデータセットが `Tensor` 型で入っている

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 $\mathcal{D} = \{(x_i, y_i)\}$) を表現するクラス

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この方法が主流です。(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, 16])  
targets.shape torch.Size([32, 1])  
-----  
inputs.shape torch.Size([32, 16])  
targets.shape torch.Size([32, 1])  
...
```

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

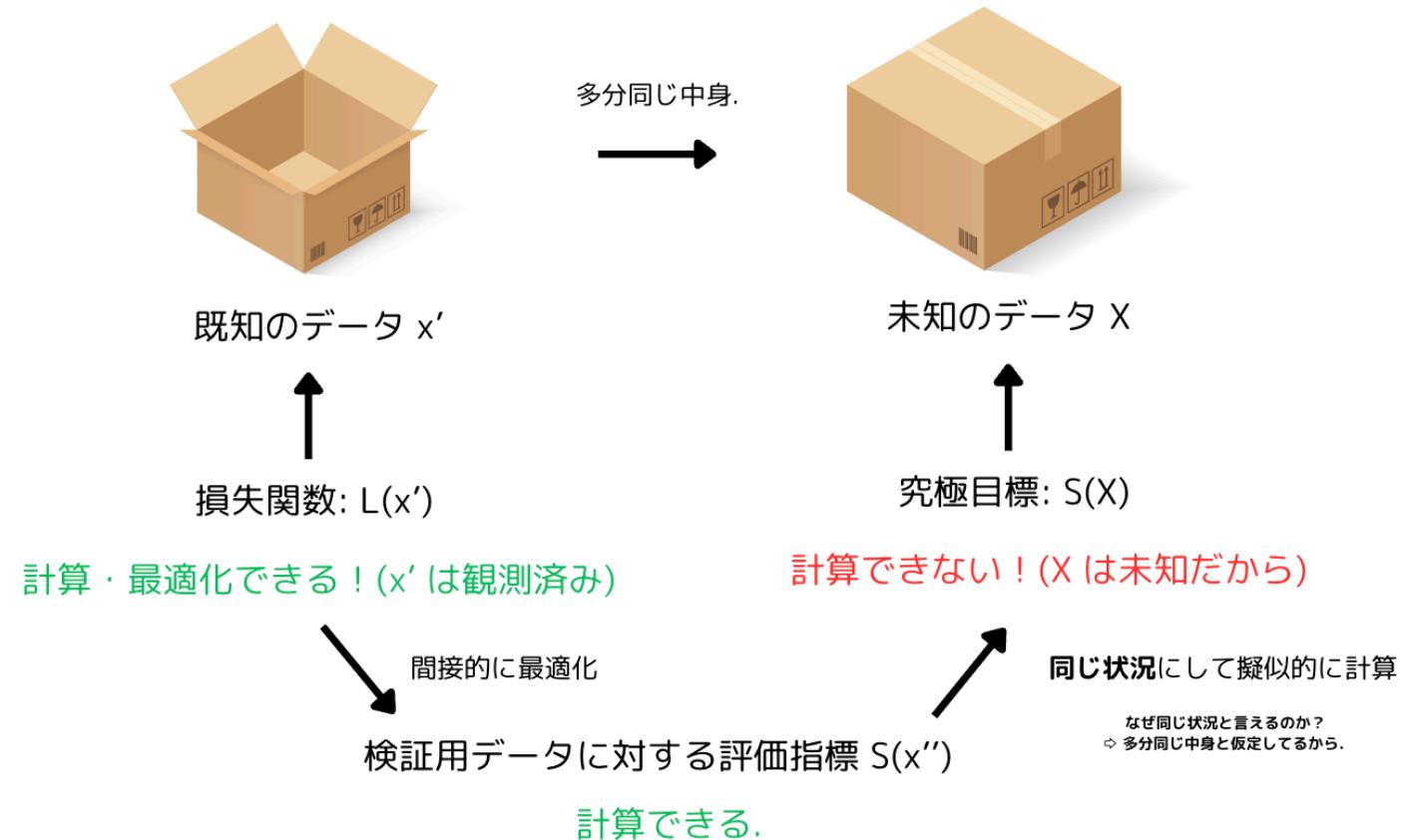
⇒ あとは学習を実装すればOK！

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

今まで評価指標も損失関数も、すべて平均二乗和誤差だった

事実: $f(x)$ を最適化すると、 $f(x)$ が最適化できる。 !!!

⇒ 微分可能なので、損失関数を平均二乗誤差にすればよい。

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！

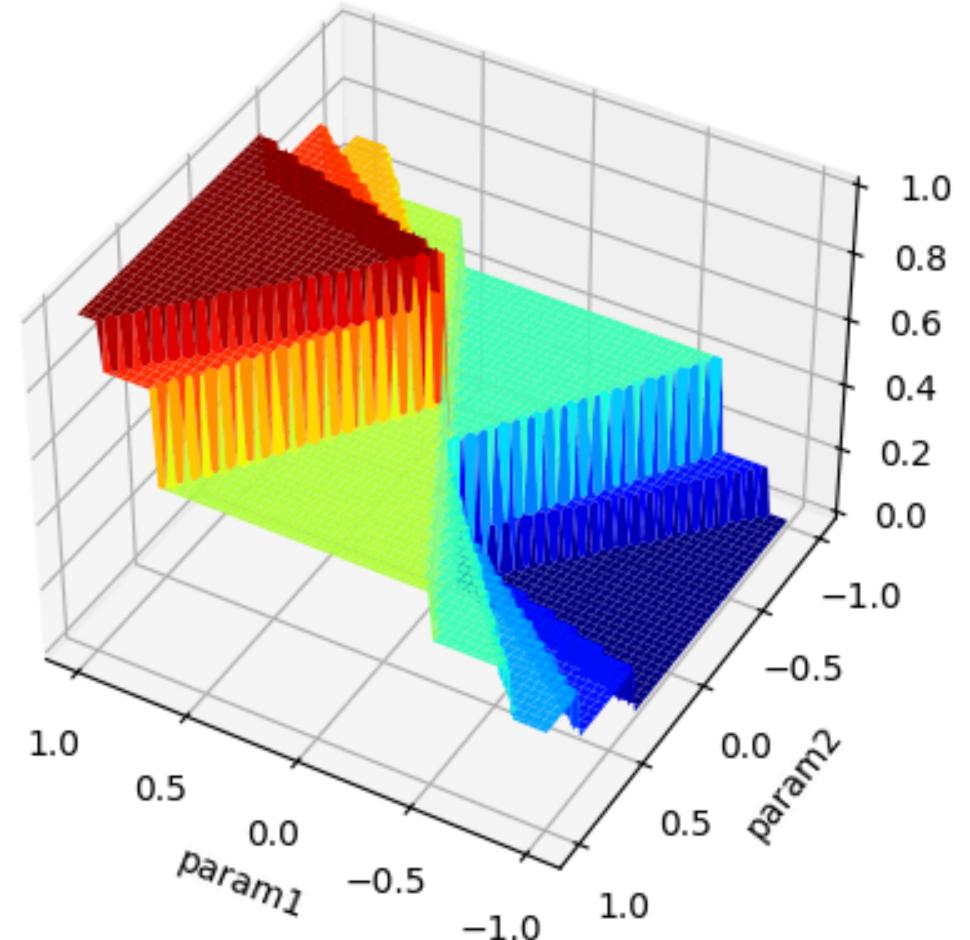
⇒ 正解率は、

- 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をロジスティック回帰で解いたときのパラメータ平面上の正解率のプロットです。ほとんど平坦(勾配が 0)なのがよくわかると思います。



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損失 (Binary Cross Entropy Loss)

$$-\frac{1}{N} \sum_{i=1}^N y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Binary Cross Entropy Loss

二値交差エントロピー損失 (Binary Cross Entropy Loss)

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

満たされていること

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)

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

3.2 確率的勾配降下法の実装

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

走義 → 計算 → backward(), 走義 → 計算 → backward(), 走義 → 計算 → backward(), 走義
走義 → 計算 → backward(), 走義 → 計算 → backward(), 走義 → 計算 → backward(), 走義

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

完善、計算、backward() 完善、計算、backward() 完善、計算、backward() 完善、計算、backward()

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

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 → 計算 → backward()

走義 > 計算 > backward(), 走義 > 計算 > backward(), 走義 > 計算 > backward(), 走義
走義 > 計算 > backward(), 走義 > 計算 > backward(), 走義 > 計算 > backward(), 走義

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(16, 32),
    ...
    nn.Linear(64, 1)
)
# ここまでが "定義"

dummy_input = torch.rand(1, 16)
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、我々が手動(?)で更新するコードを書いていた

➡  < 私がやろう

✓ `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, lr=lr)` 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！（`lr` は学習率）

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



勾配を計算したあと、`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

⚠ 注意点

`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、検証データでの損失を計算します。検証用データの内容は、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。
- 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.

バリデーション

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)
```

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つておく

```
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！)

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える。

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` に予測したい未知のデータが入っている

```
model(test)
```

⇒ 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！**

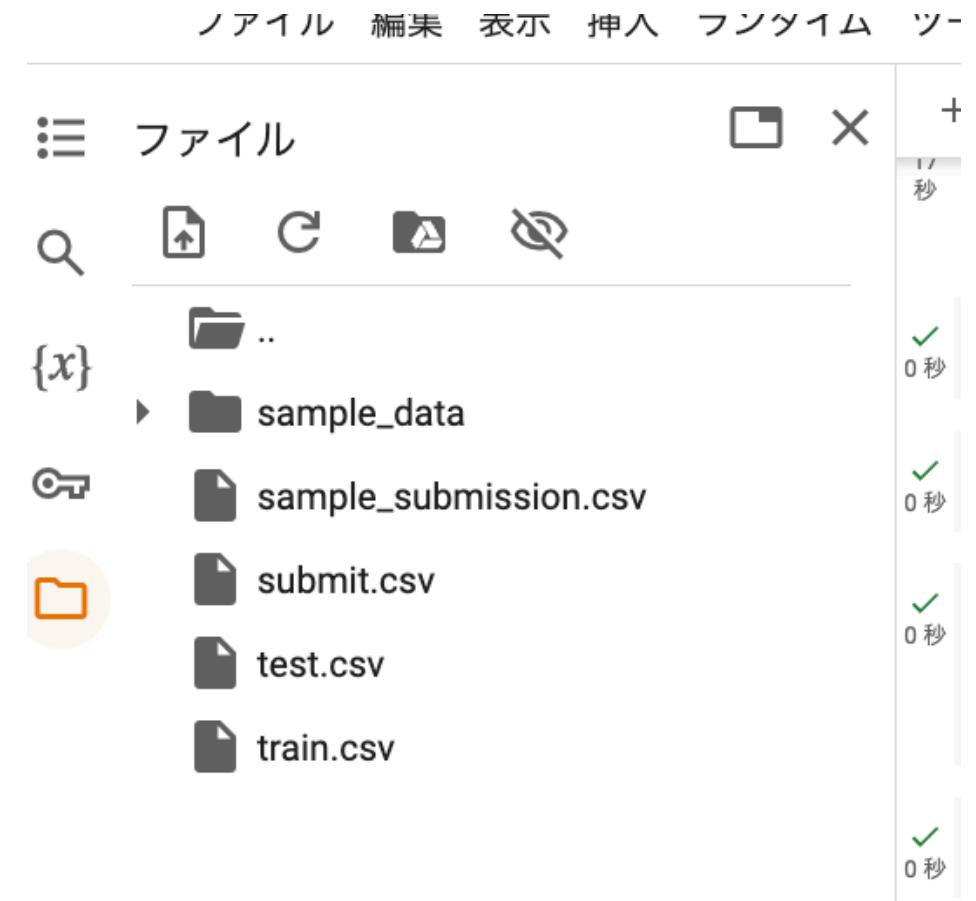
開催中のコンペ:

機械学習講習会 2024 記念 部内コンペ  

[LeaderBoard](#) [Submit](#) [Data](#) [Rules](#) [...](#)

Login as abap34

Team: team 41



5. 順位表への提出

めざせ No.1 !