Mathematics in ML

Release 0.0.1

Victor Barbarich, Adelina Tsoi

Jun 20, 2022

NOTES

1 Installation 2

2 Examples 3

3 calculus 4

4 one_optimize 7

5 visualize.one_animation 12

6 multi_optimize 15

7 visualize.multi_animation 23

8 ml.classification 26

9 ml.regression 32

10 visualize.ml_animation 36

11 ml.optimize 39

12 ml.metrics 42

13 Indices and Tables 49

Python Module Index 50

Index 51

Hi! nueramic-mathml is a library for visualizing and using basic optimization algorithms in machine learning. We have an streamlit app with visualization and some interesting features, for example, regression on stock prices or viewing gradient descent steps.

CHAPTER

ONE

INSTALLATION

1.1 Installation via Pip

1. **The latest version of torch is required to install the package**

```
$ python -c "import torch; print(torch.__version__)"
1.11.0
```

2. **The package requires python version 3.7 or later**

```
$ python --version
Python 3.10.4
```

3. **Install**

```
$ pip install nueramic-mathml
```

CHAPTER

TWO

EXAMPLES

2.1 Notebook with examples

2.2 Applications (DEVELOPMENT)

1. One optimize
2. Multidimensional optimization
3. Inner point methods
4. Regression
5. Classification

CHAPTER

THREE

CALCULUS

gradient(*function*, *x0*, *delta_x=0.0001*)

Returns the gradient of the function at a specific point x_0. A two-point finite difference formula that approximates the derivative

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i - h, \ldots, x_n)}{2h} \tag{3.1}$$

Gradient

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}^\top \tag{3.2}$$

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – function which depends on n variables from x
- **x0** (*Tensor*) – n x 1 - dimensional array $\in \mathbb{R}^n$. dtype is torch.double (float64)
- **delta_x** (*float*) – precision of two-point formula above (delta_x = h)

Returns
vector of partial derivatives

Return type
Tensor

Note: If we make delta_x $\leq$ 1e-4 gradient will return values with large error rate

Examples

```
>>> # f(x, y)  = x ** 2 + y ** 2
>>> gradient(lambda x: (x ** 2).sum(), torch.tensor([1., 2.]))
    tensor([2., 4.], dtype=torch.float64)
```

hessian(*function*, *x0*, *delta_x=0.0001*)

Returns a hessian of function at point x_0

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{3.3}$$

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – function which depends on n variables from x
- **x0** (*Tensor*) – n - dimensional array
- **delta_x** (*float*) – precision of two-point formula above (delta_x = h)

Returns
the hessian of function

Return type
Tensor

Note: If we make delta_x $\leq$ 1e-4 hessian returns matrix with large error rate

---

Examples

```
>>> def paraboloid(x): return x[0] ** 2 + 2 * x[1] ** 2
>>> print(hessian(paraboloid, torch.tensor([1, 1])).round())
[[2. 0.]
 [0. 4.]]
```

jacobian(*f_vector*, *x0*, *delta_x=0.0001*)

Returns the Jacobian matrix of a sequence of m functions from f_vector by n variables from x.

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}_{m \times n} \tag{3.4}$$

Parameters

- **f_vector** (*Sequence[Callable[[Tensor], Tensor]]*) – a flat sequence, list or tuple or other containing m functions
- **x0** (*Tensor*) – an n-dimensional array. The specific point at which we will calculate the Jacobian
- **delta_x** (*float*) – precision of gradient

Returns
the Jacobian matrix according to the above formula. Matrix n x m

Return type
Tensor

Examples

```
>>> func_3 = [lambda x: x[0] ** 2 + x[1], lambda x: 2 * x[0] + 5 * x[1], lambda x:␣
↪x[0] * x[1]]
>>> print(jacobian(func_3, torch.tensor([-1, 2])).round())
tensor([[-2.,  1.],
        [ 2.,  5.],
        [ 2., -1.]], dtype=torch.float64)
```

CHAPTER

FOUR

ONE_OPTIMIZE

golden_section_search(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns the optimal point and history using the Golden Section search[2]

Constant: $\varphi = \dfrac{(1+\sqrt{5})}{2}$

Input: $f(x) -$ function ; $a, b -$ left and right bounds ; $\varepsilon -$ precision

---

while $|a-b| > \varepsilon$:
$x_1 = b - \dfrac{b-a}{\varphi}$
$x_2 = a + \dfrac{b-a}{\varphi}$
if $f(x_1) > f(x_2)$:
$a = x_1$
else :
$b = x_2$

Return: $\dfrac{a+b}{2}$

Note: If optimization fails golden_section_search will return the last point

---

References

Parameters

- **function** (*Callable[[float | torch.Tensor], float]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **bounds** (*Tuple[float, float]*) – tuple with two numbers. This is left and right bound optimization. [a, b]

[2] Press, William H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes with Source Code CD-ROM 3rd Edition: The Art of Scientific Computing (3rd ed.). Cambridge University Press. p.492-496

- **epsilon** (*float*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – ‘min’ / ‘max’ - type of required value
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[float | torch.Tensor, *HistoryGSS*]

Examples

```
>>> def func(x): return 2.71828 ** (3 * x) + 5 * 2.71828 ** (-2 * x)
>>> point, data = golden_section_search(func, (-10, 10), type_optimization='min',␣
↪keep_history=True)
```

successive_parabolic_interpolation(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns the optimal point and history using the Successive parabolic interpolation algorithm[3]

$$x_{i+1} = x_i + \frac{1}{2}\left[\frac{(x_{i-1}-x_i)^2(f_i-f_{i-2}) + (x_{i-2}-x_i)^2(f_{i-1}-f_i)}{(x_{i-1}-x_i)(f_i-f_{i-2}) + (x_{i-2}-x_i)(f_{i-1}-f_i)}\right] \tag{4.1}$$

Input: $f(x) -$ function; $a, b -$ left and right bounds; $\varepsilon -$ precision

---

$x_0 = a,\ f_0 = f(x_0); \qquad x_1 = b,\ f_1 = f(x_1); \qquad x_2 = \frac{a+b}{2},\ f_2 = f(x_2)$
while $|x_{i+1} - x_i| \geq \varepsilon$ or $|f(x_{i+1}) - f(x_i)| \geq \varepsilon$:
x_0, x_1, x_2 so that $f_2 \leq f_1 \leq f_0$
Calculate $x_{i+1}$with the formula (4.1)

Return: x_{i+1}

Parameters

- **function** (*Callable[[float | torch.Tensor], float]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs

[3] Press, William H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes with Source Code CD-ROM 3rd Edition: The Art of Scientific Computing (3rd ed.). Cambridge University Press. p.496-499

- **bounds** (*Tuple[float, float]*) – tuple with two numbers. This is left and right bound optimization. [a, b]
- **epsilon** (*float*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – 'min' / 'max' - type of required value
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[float | torch.Tensor, *HistorySPI*]

Examples

```
>>> def func1(x): return x ** 3 - x ** 2 - x
>>> successive_parabolic_interpolation(func1, (0, 1.5), verbose=True)
Iteration: 0    |       x2 = 0.750      |       f(x2) = -0.891
Iteration: 1    |       x2 = 0.850      |       f(x2) = -0.958
Iteration: 2    |       x2 = 0.961      |       f(x2) = -0.997
Iteration: 3    |       x2 = 1.017      |       f(x2) = -0.999
Iteration: 4    |       x2 = 1.001      |       f(x2) = -1.000
...

>>> def func2(x): return - (x ** 3 - x ** 2 - x)
>>> successive_parabolic_interpolation(func2, (0, 1.5), type_optimization='max',␣
↪verbose=True)
Iteration: 0    |       x2 = 0.750      |       f(x2) = 0.891
Iteration: 1    |       x2 = 0.850      |       f(x2) =  0.958
Iteration: 2    |       x2 = 0.961      |       f(x2) =  0.997
Iteration: 3    |       x2 = 1.017      |       f(x2) =  0.999
...
```

brent(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns the optimal point and history using the Brent's algorithm[1].

Input: $f(x) -$ function ; $a, b -$ left and right bounds ; $\varepsilon -$ precision

---

$$\varphi = \frac{(1+\sqrt{5})}{2}$$
$$x_{least} = a + \varphi \cdot (b - a)$$
$$x_{new} = x_{least}$$
$$tolerance = \varepsilon \cdot |x_{least}| + 10^{-9}$$

[1] Brent, R. P., Algorithms for Minimization Without Derivatives. Englewood Cliffs, NJ: Prentice-Hall, 1973 pp.72-80

while $|x_{least} - \frac{a+b}{2}| > 2 \cdot tolerance - \frac{b-a}{2}$:

if $|x_{new} - x_{least}| > tolerance$:
calculate parabolic $remainder$ by formula (4.1)
if $remainder < previous\ remainder$ & $x_{least} + remainder \in (a, b)$:

use “parabolic" $remainder$

else:
make “golden" $remainder$
use “golden" $remainder$
$x_{new} = x_{least} + remainder$

Return: x_{least}

References

Parameters

- **function** (*Callable[[float | torch.Tensor], float]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **bounds** (*Tuple[float, float]*) – tuple with two numbers. This is left and right bound optimization. [a, b]
- **epsilon** (*float*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – ‘min’ / ‘max’ - type of required value
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[float | torch.Tensor, *HistoryBrent*]

Examples

```
>>> brent(lambda x: x ** 3 - x ** 2 - x, (0,2), verbose=True)[0]
iteration 0     x = 0.763932,   f(x) = -0.901699        type : initial
iteration 1     x = 0.763932,   f(x) = -0.901699        type : golden
iteration 2     x = 0.763932,   f(x) = -0.901699        type : golden
iteration 3     x = 0.944272,   f(x) = -0.993962        type : golden
iteration 4     x = 0.944272,   f(x) = -0.993962        type : golden
iteration 5     x = 0.999120,   f(x) = -0.999998        type : parabolic
iteration 6     x = 0.999223,   f(x) = -0.999999        type : parabolic
iteration 7     x = 0.999223,   f(x) = -0.999999        type : golden
iteration 8     x = 0.999992,   f(x) = -1.000000        type : parabolic
iteration 9     x = 1.000002,   f(x) = -1.000000        type : parabolic
iteration 10    x = 1.000002,   f(x) = -1.000000        type : golden
iteration 11    x = 1.000002,   f(x) = -1.000000        type : parabolic
Searching finished. Successfully. code 0
1.0000016327177492
```

CHAPTER

FIVE

VISUALIZE.ONE_ANIMATION

gen_animation_gss(*func*, *bounds*, *history*, ***kwargs*)

Generates an animation of the golden-section search on *func* between the *bounds*

Parameters

- **func** (*Callable*) – callable that depends on the first positional argument
- **bounds** (*tuple[float, float]*) – tuple with left and right points on the x-axis
- **history** (*HistoryGSS*) – a history object. a dict with lists. keys iteration, f_value, middle_point, left_point, right_point

Returns

go.Figure with graph

Return type

Figure

```
>>> def f(x): return x ** 3 - x ** 2 - x
>>> _, h = golden_section_search(f, (0, 2), keep_history=True)
>>> gen_animation_gss(f, (0, 2), h)
```

gen_animation_spi(*func*, *bounds*, *history*)

Generate animation. Per each iteration we create a go.Frame with parabola plot passing through three points

Parameters

- **history** (*HistorySPI*) – a history object. a dict with lists. keys iteration, f_value, middle_point, left_point, right_point
- **bounds** (*[float, float]*) – tuple with left and right points on the x-axis
- **func** (*Callable[[float], float]*) – the functions for which the story was created

Return type

go.Figure

```
>>> def f(x): return x ** 3 - x ** 2 - x
>>> _, h = successive_parabolic_interpolation(f, (0, 2), keep_history=True)
>>> gen_animation_spi(f, (0, 2), h)
```

gen_animation_brent(*func*, *history*)

Returns a visualization of the Brent algorithm. Each iteration shows which iteration.

Parameters

- **func** (*Callable[[float], float]*) – callable that depends on the first positional argument
- **history** (*HistoryBrent*) – brent optimization history

Returns
animation of optimization

Return type
Figure

```
>>> def f(x): return x ** 3 - x ** 2 - x
>>> _, h = brent(f, (0, 2), keep_history=True)
>>> gen_animation_brent(f, h)
```

parabolic_coefficients(*x0*, *x1*, *x2*, *func*)

Returns a parabolic function passing through the specified points x0, x1, x2 coeficients

```
>>> parabolic_coefficients(0, 1, 2, lambda x: x ** 2)
(1.0, 0.0, 0.0)
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_0^2 & x_0 & 1 \\ x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} \tag{5.1}$$

Parameters

- **x0** (*float*) – first point
- **x1** (*float*) – second point
- **x2** (*float*) – third point
- **func** (*Callable[[float], float]*) – the functions for which the story was created

Returns
coefficients of the parabolic function

Return type
[float, float, float]

transfer_history_gss(*history*, *func*)

Generate data for plotly express with using animation_frame for animate

```
>>> def f(x): return x ** 2
>>> _, hist = golden_section_search(f, (-1, 2), keep_history=True)
>>> data_for_plot = transfer_history_gss(hist, f)
Searching finished. Successfully. code 0
```

```
>>> data_for_plot[::30]
```

	iteration	type	x	y	size
0	0	middle	0.500000	0.250000	3
30	4	left	-0.291796	0.085145	3
60	8	right	0.042572	0.001812	3

Parameters

- **history** (*HistoryGSS*) – a history object. a dict with lists. keys iteration, f_value, middle_point, left_point, right_point
- **func** – the functions for which the story was created

Returns
pd.DataFrame for gen_animation_gss. index - num of iteration.

Return type
DataFrame

CHAPTER

SIX

MULTI_OPTIMIZE

6.1 Gradient descent with constant step

Algorithm Flowchart

gd_constant(*function*, *x0*, *epsilon=1e-05*, *gamma=0.1*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns a tensor n x 1 with optimal point and history using Algorithm with constant step. The gradient of the function shows us the direction of increasing the function. The idea is to move in the opposite direction to x_{k+1} where $f(x_{k+1}) < f(x_k)$.

But, if we add a gradient to x_k without changes, our method will often diverge. So we need to add a gradient with some weight γ .

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – Torch tensor which is initial approximation
- **epsilon** (*float*) – optimization accuracy
- **gamma** (*float*) – gradient step
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[*Tensor*, *HistoryGD*]

Examples

```
>>> def func(x): return x[0] ** 2 + x[1] ** 2
>>> x_0 = torch.tensor([1, 2])
>>> solution = gd_constant(func, x_0)
>>> print(solution[0])
tensor([1.9156e-06, 3.8312e-06], dtype=torch.float64)
```

6.2 Gradient descent with fractional step

Algorithm Flowchart

---

gd_frac(*function*, *x0*, *epsilon=1e-05*, *gamma=0.1*, *delta=0.1*, *lambda0=0.1*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns a tensor n x 1 with optimal point and history using Algorithm with fractional step.

Requirements: $0 < \lambda_0 < 1$ is the step multiplier, $0 < \delta < 1$ influence on step size.

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – Torch tensor which is initial approximation
- **epsilon** (*float*) – optimization accuracy
- **gamma** (*float*) – gradient step
- **delta** (*float*) – value of the crushing parameter
- **lambda0** (*float*) – initial step
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[*Tensor*, *HistoryGD*]

Examples

```
>>> def func(x): return x[0] ** 2 + x[1] ** 2
>>> x_0 = torch.tensor([1, 2])
>>> solution = gd_frac(func, x_0)
>>> print(solution[0])
tensor([1.9156e-06, 3.8312e-06], dtype=torch.float64)
```

6.3 Gradient descent with optimal step

Algorithm Flowchart

---

gd_optimal(*function*, *x0*, *epsilon=1e-05*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns a tensor n x 1 with optimal point and history using Algorithm with optimal step. The idea is to choose a gamma that minimizes the function in the direction $f'(x_k)$

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – Torch tensor which is initial approximation
- **epsilon** (*float*) – optimization accuracy
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns

tuple with point and history.

Return type

Tuple[*Tensor*, *HistoryGD*]

Examples

```
>>> def func(x): return -torch.exp(- x[0] ** 2 - x[1] ** 2)
>>> x_0 = torch.tensor([1, 2])
>>> solution = gd_optimal(func, x_0)
>>> print(solution[0])
tensor([9.2070e-08, 1.8405e-07], dtype=torch.float64)
```

6.4 Nonlinear conjugate gradient method

Algorithm Flowchart

---

nonlinear_cgm(*function*, *x0*, *epsilon=1e-05*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns a tensor n x 1 with optimal point and history. Algorithm works when the function is approximately quadratic near the minimum, which is the case when the function is twice differentiable at the minimum and the second derivative is non-singular there[1]

References

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – Torch tensor which is initial approximation
- **epsilon** (*float*) – optimization accuracy
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns

tuple with point and history.

Return type

Tuple[*Tensor*, *HistoryGD*]

[1] Nocedal, J., & Wright, S. J. (2006). 5.2 NONLINEAR CONJUGATE GRADIENT METHOD In Numerical optimization (pp. 121). essay, Springer.

Examples

Example for $f(x, y) = (x + 0.5)^2 + (y - 0.5)^2, \quad x = 1$

```
>>> def func(x): return 10 * x[0] ** 2 + x[1] ** 2 / 5
>>> x_0 = torch.tensor([1, 2])
>>> solution = nonlinear_cgm(func, x_0)
>>> print(solution[0])
tensor([6.9846e+25, 4.2454e+26], dtype=torch.float64)
```

6.5 BFGS (Broyden–Fletcher–Goldfarb–Shanno) algorithm

bfgs(*function*, *x0*, *tolerance=1e-08*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Returns a tensor n x 1 with optimal point and history using the BFGS method[1]

Broyden–Fletcher–Goldfarb–Shanno algorithm The algorithm does not use Wolfe conditions. Instead of wolfe, alg uses the optimal step.

Note: The algorithm only works for a flat x0, and the functions should depend on a flat array

References

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **x0** (*Tensor*) – start minimization point
- **tolerance** (*float*) – criterion of stop os l2 norm(grad f) < tolerance
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Return type
Tuple[*Tensor*, *HistoryGD*]

[1] Wright and Nocedal, 'Numerical Optimization', 1999; pp.136-140 BFGS algorithm.

Examples

```
>>> def func(x): return 10 * x[0] ** 2 + x[1] ** 2 / 5
>>> x_0 = torch.tensor([1, 2])
>>> solution = bfgs(func, x_0)
>>> print(solution[0])
tensor([3.4372e-14, 1.8208e-14], dtype=torch.float64)
```

6.6 Newton's method under equality constrains

constrained_lagrangian_solver(*function*, *x0*, *constraints*, *x_bounds=None*, *epsilon=0.0001*, *max_iter=250*, *keep_history=False*, *verbose=False*)

Returns a tensor n x 1 with optimal point and history of minimization by newton_eq_const. Alias of ''Newton's method under equality constrains"[1]

Example for $f(x, y) = (x + 0.5)^2 + (y - 0.5)^2, \quad x = 1$

References

Parameters

- **function** (*Callable[[float | torch.Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – some specific point x(Torch tensor)
- **constraints** (*Sequence[Callable[[float | torch.Tensor], Tensor]]*) – list of equality constraints
- **x_bounds** (*Union[Sequence[Tuple[float, float]], None, Tensor]*) – bounds on x. e.g. 0 <= x[i] <= 1, then x_bounds[i] = (0, 1)
- **epsilon** (*float*) – optimization accuracy
- **max_iter** (*int*) – maximum number of iterations
- **keep_history** (*bool*) – flag of return history
- **verbose** (*bool*) – flag of printing iteration logs

Returns

tuple with point and history.

Return type

Tuple[*Tensor*, *HistoryGD*]

[1] Nocedal, J., & Wright, S. J. (2006). 17.4 PRACTICAL AUGMENTED LAGRANGIAN METHODS. In Numerical optimization (pp. 519–521). essay, Springer.

Examples

```
>>> constrained_lagrangian_solver(lambda x: (x[0] + 0.5) ** 2 + (x[1] - 0.5) ** 2,
>>>                               torch.tensor([0.1, 0.1]),[lambda x: x[0] - 1]))
tensor([1.0540, 0.5000], dtype=torch.float64)
```

6.7 Log Barrier method

log_barrier_solver(*function*, *x0*, *inequality_constraints*, *epsilon=1e-05*, *max_iter=1000*, *keep_history=False*, *verbose=False*)

Returns optimal point of optimization with inequality constraints by Log Barrier method[1]

References

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – some specific point x(Torch tensor)
- **epsilon** (*float*) – optimization accuracy
- **inequality_constraints** (*Sequence[Callable[[Tensor], Tensor]]*) – $\mathcal{I}$ is set of inequality functions
- **max_iter** (*int*) – maximum number of iterations
- **keep_history** (*bool*) – flag of return history
- **verbose** (*bool*) – flag of printing iteration logs

Returns
tuple with point and history.

Return type
Tuple[*Tensor*, *HistoryGD*]

Examples

Example for $f(x, y) = (x + 0.5)^2 + (y - 0.5)^2, \quad 0 \leq x \leq 1, 0 \leq y \leq 1$

```
>>> log_barrier_solver(lambda x: (x[0] + 0.5) ** 2 + (x[1] - 0.5) ** 2, torch.
↪tensor([0.9, 0.1]),
>>>                    [lambda x: x[0], lambda x: 1 - x[0], lambda x: x[1], lambda␣
↪x: 1 - x[1]])
tensor([0.0032, 0.5000], dtype=torch.float64)
```

[1] Nocedal, J., & Wright, S. J. (2006). 19.6 THE PRIMAL LOG-BARRIER METHOD. In Numerical optimization (pp. 583–584). essay, Springer.

6.8 Primal-dual algorithm

primal_dual_interior(*function*, *x0*, *inequality_constraints*, *mu=0.0001*, *epsilon=1e-12*, *alpha=0.1*, *max_iter=200*, *verbose=False*, *keep_history=False*)

Returns point and history of minimization.[1]

AIM: minimize $f(x)$ subject to $c(x) \geqslant 0$; c from inequality_constraints

$B(x, \mu) = f(x) - \mu \sum_{i=1}^{m} \log(c_i(x)) \to \min$

Here μ is a small positive scalar, $\mu \to 0$

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **x0** (*Tensor*) – some specific point x(Torch tensor)
- **inequality_constraints** (*Sequence[Callable[[Tensor], Tensor]]*) – $\mathcal{I}$ is set of inequality functions
- **mu** (*float*) – is a small positive scalar, sometimes called the "barrier parameter"
- **epsilon** (*float*) – optimization accuracy
- **alpha** (*float*) – step length
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns
tuple with point and history.

Raises
ArithmeticError – if x0 is not in trust region.

Return type
Tuple[*Tensor*, *HistoryGD*]

Reference

Examples

```
>>> primal_dual_interior(lambda x: (x[0] + 0.5) ** 2 + (x[1] - 0.5) ** 2, torch.
↪tensor([0.9, 0.1]),
>>>                       [lambda x: x[0], lambda x: 1 - x[0], lambda x: x[1], lambda␣
↪x: 1 - x[1]])[0]
tensor([1.9910e-04, 5.0000e-01], dtype=torch.float64)
```

You can choose the best model using our flowchart

[1] https://en.wikipedia.org/wiki/Interior-point_method

CHAPTER

SEVEN

VISUALIZE.MULTI_ANIMATION

gen_simple_gradient(*function*, *history*, *cnt_dots=200*, *title='Contour plot with optimization steps'*, *showlegend=True*, *font_size=18*)

Return go.Figure with gradient steps under contour plot. Not animated

Parameters

- **function** (*Callable[[Tensor], Tensor]*) – callable that depends on the first positional argument
- **history** (*HistoryGD*) – History after some gradient method
- **cnt_dots** (*int*) – the numbers of point per each axis
- **title** (*str*) – title of chart
- **showlegend** (*bool*) – flag of showing legend
- **font_size** (*int*) – font size

Returns

go.Figure with contour and line of gradient steps

Return type

Figure

```
>>> def f(x): return x[0] ** 2 + x[1] ** 2 / 2
>>> x_opt, hist = gd_optimal(f, torch.tensor([8, 5]), keep_history=True)
>>> gen_simple_gradient(f, hist).show()
```

---

gen_animated_surface(*function*, *history*, *cnt_dots=100*, *title='Surface with optimization steps'*)

Return go.Figure with animation per each step of descent

Parameters

- **function** (*Callable[[Tensor], float]*) – callable that depends on the first positional argument
- **history** (*HistoryGD*) – History after some gradient method
- **cnt_dots** (*int*) – the numbers of point per each axis
- **title** (*str*) – how many frames will drawing. ~300 frames will be drawn for ~5-10 seconds

Returns

go.Figure with animation steps on surface

Return type
Figure

```
>>> def f(x): return x[0] ** 2 + x[1] ** 2 / 9
>>> _, h = bfgs(f, torch.tensor([10, 10]), keep_history=True)
```

---

make_contour(*function*, *bounds*, *cnt_dots=100*, *colorscale='teal'*, *showlegend=False*)

Return go.Contour for draw by go.Figure. Evaluate function per each point in the 2d grid

Parameters

- **function** (*Callable[[Tensor], float | torch.Tensor]*) – callable that depends on the first positional argument
- **bounds** (*tuple[tuple[float, float], tuple[float, float]]*) – two tuples with constraints for x- and y-axis
- **cnt_dots** (*int*) – number of point per each axis
- **colorscale** – plotly colorscale for go.Contour
- **showlegend** (*bool*) – show legend flag

Returns
go.Contour

Return type
Contour

```
>>> def f(x): return x[0] ** 2 + x[1] ** 2 / 2
>>> make_contour(f, ((0, 1), (0, 1)), cnt_dots=4)
```

```
Contour({
    'colorscale': [[0.0, 'rgb(209, 238, 234)'], [0.16666666666666666, 'rgb(168,
                   219, 217)'], [0.3333333333333333, 'rgb(133, 196, 201)'], [0.5,
                   'rgb(104, 171, 184)'], [0.6666666666666666, 'rgb(79, 144,
                   166)'], [0.8333333333333334, 'rgb(59, 115, 143)'], [1.0,
                   'rgb(42, 86, 116)']],
    'name': 'f(x, y)',
    'showlegend': False,
    'showscale': False,
    'x': array([0.        , 0.33333334, 0.6666666 , 1.        ], dtype=float32),
    'y': array([0.        , 0.33333334, 0.6666666 , 1.        ], dtype=float32),
    'z': array([[0.        , 0.11111112, 0.4444444 , 1.        ],
                [0.05555556, 0.16666669, 0.49999994, 1.0555556 ],
                [0.2222222 , 0.3333333 , 0.66666657, 1.2222222 ],
                [0.5       , 0.6111111 , 0.9444444 , 1.5       ]], dtype=float32)
})
```

make_surface(*function*, *bounds*, *cnt_dots=100*, *colorscale='teal'*, *showlegend=False*)

Return go.Surface for draw by go.Figure. Evaluate function per each point in the 2d grid

Parameters

- **function** (*Callable[[Tensor], float | torch.Tensor]*) – callable that depends on the first positional argument

- **bounds** (*tuple[tuple[float, float], tuple[float, float]]*) – two tuples with constraints for x- and y-axis
- **cnt_dots** (*int*) – number of point per each axis
- **colorscale** – plotly colorscale for go.Contour
- **showlegend** (*bool*) – showlegend flag

Returns
go.Surface

Return type
Surface

```
>>> def f(x): return x[0] ** 2 + x[1] ** 2 / 2
>>> make_surface(f, ((0, 1), (0, 1)), cnt_dots=4)
```

```
Surface({
    'colorscale': [[0.0, 'rgb(209, 238, 234)'], [0.16666666666666666, 'rgb(168,
                   219, 217)'], [0.3333333333333333, 'rgb(133, 196, 201)'], [0.5,
                   'rgb(104, 171, 184)'], [0.6666666666666666, 'rgb(79, 144,
                   166)'], [0.8333333333333334, 'rgb(59, 115, 143)'], [1.0,
                   'rgb(42, 86, 116)']],
    'name': 'f(x, y)',
    'opacity': 0.75,
    'showlegend': False,
    'x': array([0.        , 0.33333334, 0.6666666 , 1.        ], dtype=float32),
    'y': array([0.        , 0.33333334, 0.6666666 , 1.        ], dtype=float32),
    'z': array([[0.        , 0.11111112, 0.4444444 , 1.        ],
                [0.05555556, 0.16666669, 0.49999994, 1.0555556 ],
                [0.2222222 , 0.3333333 , 0.66666657, 1.2222222 ],
                [0.5       , 0.6111111 , 0.9444444 , 1.5       ]], dtype=float32)
})
```

CHAPTER

EIGHT

ML.CLASSIFICATION

class BaseClassification

Bases: Module

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

metrics_tab(*x*, *y*)

Returns metrics dict with recall, precision, accuracy, f1, auc roc scores

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value

Param
dict with recall, precision, accuracy, f1, auc roc scores

Returns
dict with 5 metrics

Return type
dict

class LogisticRegression

Bases: *BaseClassification*

Binary classification model

Let $x \in \mathbb{R}^{n \times m},\ w \in \mathbb{R}^{m \times 1},\ I = [1]_{n \times 1}$, x_i – is a row and $x_i \in \mathbb{R}^{1 \times m}$

Model:

$$\mathbb{P}(y_i = 1|w) = \frac{1}{1 + \exp(x_i \cdot w + b)} \tag{8.1}$$

__init__(*kernel='linear'*)

Parameters
kernel (*Literal['linear', 'perceptron']*) – 'linear' or 'perceptron'. linear - basic logistic regression, perceptron - nn with 2 hidden layer with dim1 = 1024, dim2 = 512

fit(*x*, *y*, *epochs=1000*, *l1_lambda=0*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model Logistic Regression

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** – max number of sgd implements
- **l1_lambda** (*float*) – l1 regularization weight
- **show_epoch** (*int*) – amount of showing epochs
- **print_function** (*Callable*) – print or streamlit.write

Returns
trained model

Return type
Module

forward(*x*)

Returns confidence probabilities of first class

Parameters
x (*Tensor*) – training set

Returns
probabilities

Return type
Tensor

init_weights(*x*)

Initialization weights

Parameters
x (*Tensor*) – input torch tensor

predict(*x*)

Returns binary class 0 or 1 instead of probabilities

Parameters
x – some tensor with shape[1] = n_features

Returns

class LogisticRegressionRBF

Bases: *BaseClassification*

This is a logistic regression, but before we make a basic linear prediction and apply the sigmoid, we transfer x to another space using radial basis functions. The dimension of this space depends on the basis matrix x (x_basis)[1]

Radial basis functions

1. gaussian $\varphi(x, x_b) = e^{-\|x-x_b\|^2}$
2. linear $\varphi(x, x_b) = \|x - x_b\|$
3. multiquadratic $\varphi(x, x_b) = \sqrt{1 + \|x - x_b\|^2}$

[1] https://en.wikipedia.org/wiki/Radial_basis_function

References

__init__(*x_basis*, *rbf='gaussian'*)

Parameters

- **x_basis** (*Tensor*) – centers of basis functions
- **rbf** (*Literal['linear', 'gaussian', 'multiquadratic']*) – type of rbf function. Available: ['linear', 'gaussian']

fit(*x*, *y*, *epochs=100*, *l1_lambda=0*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model Logistic Regression with RBF

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** – max number of sgd implements
- **l1_lambda** (*float*) – l1 regularization weight
- **show_epoch** (*int*) – amount of showing epochs
- **print_function** (*Callable*) – e.g. print or streamlit.write

Returns
trained model

Return type
Module

forward(*x=None*, *phi_matrix=None*)

Returns a "probability" (confidence) of class 1

Parameters

- **x** (*Optional[Tensor]*) – 2D array
- **phi_matrix** (*Optional[Tensor]*) – 2D array

Returns
1D array

Return type
Tensor

make_phi_matrix(*x*)

Returns n x k array with calculated phi(x_i, x_basis_j). n is number of observation from x (x.shape[0]) k is number of basis from initialization.

$$\begin{bmatrix} \varphi(x_1, x_1^{\text{basis}}) & \varphi(x_1, x_2^{\text{basis}}) & \dots & \varphi(x_1, x_k^{\text{basis}}) \\ \varphi(x_2, x_1^{\text{basis}}) & \varphi(x_2, x_2^{\text{basis}}) & \dots & \varphi(x_2, x_k^{\text{basis}}) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi(x_n, x_1^{\text{basis}}) & \varphi(x_n, x_2^{\text{basis}}) & \dots & \varphi(x_n, x_k^{\text{basis}}) \end{bmatrix} \tag{8.2}$$

Parameters
x (*Tensor*) – Array k x m dimensional. k different x_i and m features

Return type
Tensor

predict(*x*)

Returns binary class 0 or 1 instead of -1; 1

Parameters
x – some tensor with shape[1] = n_features

Returns

class SVM

Bases: *BaseClassification*

Binary classification model. Method predict: SVM.predict(x) –> original names

Mathematical model:

$$\hat{y} = \text{sign}(x \cdot w - b \cdot I) \tag{8.3}$$

$x \in \mathbb{R}^{n \times m},\ w \in \mathbb{R}^{m \times 1},\ I = [1]_{n \times 1}$

And search of best w, b calculates by minimization of Hinge loss

$$\lambda \|\mathbf{w}\|^2 + \left[\frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - y_i (x_i \cdot w - b)\right) \right] \longrightarrow \min \tag{8.4}$$

or PEGASOS algorithm

Variables

- **scale** – for the best training and prediction, the model will standard normalize the input x data. The first time you call model, std and mean will be saved and in the future use the parameters for scaling. x = (x is the average value) / std
- **weights** – parameters of model. Initialize after first calling

__init__()

Initialization of SVM

_fit_pegasos(*x*, *y*, *epochs=20*, *lambda_reg=0.95*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model SVM[2]

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** – max number of sgd implements
- **lambda_reg** (*float*) – regularization parameter
- **show_epoch** (*int*) – amount of showing epochs
- **print_function** (*Callable*) – print or streamlit.write

Returns
trained model

Return type
Module

[2] Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. Shai Shalev-Shwartz; Yoram Singer; Nathan Srebro; Andrew Cotter

References

_fit_sgd(*x*, *y*, *epochs=500*, *l2_lambda=0*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model SVM

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** – max number of sgd implements
- **l2_lambda** (*float*) – l2 regularization weight
- **show_epoch** (*int*) – amount of showing epochs
- **print_function** (*Callable*) – print or streamlit.write

Returns
trained model

fit(*x*, *y*, *method='sgd'*, *epochs=100*, *lambda_reg=0.1*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model SVM

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value. binary classes
- **method** (*Literal['pegasos', 'sgd']*) – optimization method. Available PEGASOS or sgd
- **epochs** – max number of sgd and pegasos steps
- **lambda_reg** (*float*) – l2 regularization weight
- **show_epoch** (*int*) – amount of showing epochs
- **print_function** (*Callable*) – print or streamlit.write

Returns
trained model

forward(*x*)

Returns x @ w + b

$$f(x) = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_m \cdot x_m \tag{8.5}$$

Parameters
x (*Tensor*) – input observations, tensor n x m (n is the number of observations that have m parameters)

Returns
regression value (yes, no classification, for binary classes call predict)

Return type
Tensor

`init_weights`(*x*)

Initialization weights

Parameters
x (*Tensor*) – input torch tensor

`predict`(*x*)

Returns binary class from the first call, in training or just call

Parameters
x – some tensor with shape[1] = n_features

Returns

`scaler`(*x*)

Returns the scaled value of x. Standard x scaling and storing settings

Parameters
x (*Tensor*) – torch.Tensor

Returns

Return type
Tensor

CHAPTER

NINE

ML.REGRESSION

class BaseRegressionModel

Bases: Module

Base model for regression.

Variables

- **w** – weights of model
- **_best_state** – _best_state while model training
- **_best_loss** – _best_loss while model training

__init__()

Initialization of base model for different regression

Return type
None

fit(*x*, *y*, *epochs=2000*, *lr=0.0001*, *l1_constant=0.0*, *l2_constant=0.0*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model of Regression

Target function

Training happens by minimizing loss function:

$$\mathcal{L}(w) = \lambda_1 \|w\|_1 + \lambda_2 \|w\|_2 + \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \longrightarrow \min_w \tag{9.1}$$

$x_i \in \mathbb{R}^{1\times m}, w \in \mathbb{R}^{m\times 1}, y_i \in \mathbb{R}^1$

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** (*int*) – max number of sgd implements
- **lr** (*float*) – Adam optimizer learning rate
- **show_epoch** (*int*) – amount of showing epochs
- **l1_constant** (*float*) – parameter of l1 regularization
- **l2_constant** (*float*) – parameter of l2 regularization

- **print_function** (*Callable*) – a function that will print verbose

Returns
trained model

Return type
Module

forward(*x*, *transformed=True*)

Returns transform(x) @ w, w is the weights for each parameter, transform(x) is some transformed matrix.

1. Linear transformed - same matrix
2. Polynomial transform check polynomial
3. Exponential transform check exponential

For linear returns x @ w + b, w is the weights for each parameter, and b is the bias

$$\hat{Y}_{n\times 1} = X_{n\times m} \cdot W_{m\times 1} + b \cdot I_{n\times 1} = \begin{bmatrix} w_1 x_{1,1} + w_2 x_{1,2} + \cdots + w_m + x_{1,m} + b \\ \vdots \\ w_1 x_{n,1} + w_2 x_{n,2} + \cdots + w_m + x_{n,m} + b \end{bmatrix} \tag{9.2}$$

For non linear:

$$\hat{Y}_{n\times 1} = X_{\text{transformed}} \cdot W \tag{9.3}$$

Parameters

- **x** (*Tensor*) – input observations, tensor n x m (n is the number of observations that have m parameters)
- **transformed** (*bool*) – the flag of the converted x. if true, x will not be converted

Returns
regression value

Return type
Tensor

init_weights(*x*)

Initializing weights

Parameters
x (*Tensor*) – input observations, tensor n x m (n is the number of observations that have m parameters)

Return type
None

metrics_tab(*x*, *y*)

Returns metrics dict with r2, mae, mse, mape

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value of regression

Returns
r2, mae, mse, mape

Return type
dict

static transform(*x*)

Returns transformed x. Default is return x without changes

Parameters
x (*Tensor*) – torch tensor

Returns
transformed torch tensor

Return type
Tensor

class LinearRegression

Bases: *BaseRegressionModel*

Model:

$$\hat{y}(x) = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_m \cdot x_m \tag{9.4}$$

__init__(*bias=True*)

Initialization of base model for different regression

Parameters
bias (*bool*) –

Return type
None

class PolynomialRegression

Bases: *BaseRegressionModel*

Polynomial regression model:

$$\hat{y}(x) = \sum_{\alpha_1 + \cdots + \alpha_m \leq k} w_i \cdot x_1^{\alpha_1} \circ x_2^{\alpha_2} \cdot \cdots \circ x_m^{\alpha_m} \tag{9.5}$$

$\alpha_i \in \mathbb{Z}_+, x_i - i$ column from x matrix

$x_i, y, \hat{y} \in \mathbb{R}^{n \times 1}, \circ$ - hadamard product (like np.array * np.array)

__init__(*degree*)

Parameters
degree (*int*) – degree of polynomial regression

Return type
None

transform(*x*)

Returns poly-transformed data

Parameters
x (*Tensor*) – torch tensor

Returns
transformed tensor

Return type
Tensor

class ExponentialRegression

Bases: *BaseRegressionModel*

Exponential regression

$$\hat{y}_i = \exp(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_m \cdot x_m) \tag{9.6}$$

__init__()

Initialization of base model for different regression

fit(*x*, *y*, *epochs=5000*, *lr=0.0001*, *l1_constant=0.0*, *l2_constant=0.0*, *show_epoch=0*, *print_function=<built-in function print>*)

Returns trained model of exponential Regression

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** (*int*) – max number of sgd implements
- **lr** (*float*) – Adam optimizer learning rate
- **show_epoch** (*int*) – amount of showing epochs
- **l1_constant** (*float*) – parameter of l1 regularization
- **l2_constant** (*float*) – parameter of l2 regularization
- **print_function** (*Callable*) – a function that will print verbose

Returns
trained model

Return type
Module

forward(*x*, *transformed=True*)

Returns exponential regression function

$$\hat{y} = \exp(x \cdot w + b) \tag{9.7}$$

Parameters

- **x** (*Tensor*) – input observations, tensor n x m (n is the number of observations that have m parameters)
- **transformed** (*bool*) – flag of transformed x

Returns
regression value

Return type
Tensor

CHAPTER

TEN

VISUALIZE.ML_ANIMATION

gen_classification_plot(*x_tensor*, *y_true*, *model=None*, *threshold=0.5*, *cnt_points=1000*, *k=0.1*, *title=None*, *epsilon=0.0001*, *insert_na=False*)

Returns a graph with a distribution and an optional line. If dim(x) = 2, then you can get model. If dim(x) > 2, then returns graph of TSNE from sklearn.manifold with default settings. dim(x) is not support

Note: if model os linear and have one layer, simple activation function, then visualization will faster

Warning: if the model is heavy, then you should reduce cnt_points, but the probability of missing points is higher, and the visualization will be rather incorrect. You can increase the gap by increasing the epsilon.

Parameters

- **x_tensor** (*Tensor*) – training tensor
- **y_true** (*Tensor*) – target tensor. array with true values of binary classification
- **model** (*Optional[Module]*) – some model that returns a torch tensor with class 1 probabilities using the call: model(x)
- **threshold** (*float*) – if model(xi) >= threshold, then yi = 1
- **cnt_points** (*int*) – number of points on each of the two axes when dim(x) = 2
- **k** (*float*) – constant for draw on section: [x.min() - (x.max() - x.min()) * k, x.max() + (x.max() - x.min()) * k]
- **title** (*Optional[str]*) – title of plots
- **epsilon** (*float*) – contour line points: $\{x \in \mathbb{R}^2 \,|\, \text{threshold} - \text{epsilon} \leq \text{model}(x) \leq \text{threshold} + \text{epsilon}\}$
- **insert_na** (*bool*) – na insertion flag when two points too far away

Returns

scatter plot go.Figure

Return type

Figure

```
>>> from sklearn.datasets import make_moons
>>> torch.random.manual_seed(7)
>>> x, y = make_moons(1000, noise=0.15, random_state=7)
```

(continues on next page)

(continued from previous page)

```
>>> x, y = torch.tensor(x), torch.tensor(y)

>>> lr_rbf = LogisticRegressionRBF(x[:50])
>>> lr_rbf.fit(x, y, epochs=5000)

>>> lr_rbf.metrics_tab(x, y)
```

```
{'recall': 0.9980000257492065,
 'precision': 0.9842209219932556,
 'accuracy': 0.9909999966621399,
 'f1': 0.9910625822119956,
 'auc_roc': 0.9995800006320514}
```

```
>>> gen_classification_plot(x, y, model, threshold=0.5, epsilon=0.001)
```

---

roc_curve_plot(*y_true*, *y_prob*, *fill=False*)

Return figure with plotly.Figure ROC curve

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_prob** (*Tensor*) – array of probabilities of confidence of belonging to the 1st class
- **fill** (*bool*) – flag for filling the area under the curve

Returns
go.Figure

Return type
Figure

```
>>> yt = torch.tensor([1, 1, 0, 0, 1, 0])
>>> yp = torch.tensor([0.7, 0.6, 0.3, 0.5, 0.4, 0.4])
>>> roc_curve_plot(yt, yp)
```

gen_regression_plot(*x_tensor*, *y_tensor*, *model=None*, *title='Scatter plot'*)

Returns a graph with a regression and scatter of initial distribution.

Note: Support 1d x_tensor. If x_tensor n_d method applied t-SNE

Parameters

- **x_tensor** (*Tensor*) – training tensor
- **y_tensor** (*Tensor*) – target tensor. array with true regression values
- **model** (*Optional[Module]*) – some model that returns a torch tensor with class 1 probabilities using the call: model(x)
- **title** (*Optional[str]*) – title of plots

Returns
scatter plot go.Figure and line of regression

Return type
Figure

```
>>> from sklearn.datasets import make_regression
>>> x, y = make_regression(200, 1, noise=20, random_state=21)
>>> x, y = torch.tensor(x), torch.tensor(y)
>>> regression = LinearRegression().fit(x, y)
>>> gen_regression_plot(x, y, regression)
```

```
>>> # Let's create 4-dimensional data and perform a linear regression.
>>> # After that, t-sne will show the data on the plane

>>> x, y = make_regression(200, 4, noise=20, random_state=21)
>>> x, y = torch.tensor(x), torch.tensor(y)
>>> regression = LinearRegression().fit(x, y)
>>> gen_regression_plot(x, y, regression)
```

```
>>> regression.metrics_tab(x, y)
```

```
{'r2': 0.9711183309555054,
 'mae': 15.044872283935547,
 'mse': 365.99530029296875,
 'mape': 55.71377182006836}
```

CHAPTER

ELEVEN

ML.OPTIMIZE

class HistorySA

Bases: TypedDict

iteration: list

loss: list

point: Optional[list]

type_ball: tuple

class NueSGD

Bases: object

__init__(*model*, *lr=0.0001*)

Implementation of classic SGD (stochastic gradient descent) optimization algorithm.

Parameters

- **model** (*Module*) – pytorch model that can be called and have a ".loss" method
- **lr** (*float*) – learning rate. Multiplier of gradient step: x = x - lr * grad(x)

optimize(*x*, *y*, *epochs=1*, *batch_size=-1*, *num_verbose=0*, *lamb=0.3*, *print_function=<built-in function print>*)

Function apply MySGD optimizer, and train model.

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value
- **epochs** (*int*) – max number of sgd implements
- **batch_size** (*int*) – size of batch for each epoch. default is -1 - all data
- **num_verbose** (*int*) – number of iterations to be printed
- **lamb** (*float*) – rate of history loss evaluation
- **print_function** (*Callable*) – e.g. print or streamlit.write or something else

Returns

trained model and history

Return type

[<class 'torch.nn.modules.module.Module'>, <class 'dict'>]

step()

Update parameters data

W = W - lr * Grad(W)

Returns
None

Return type
None

zero_grad()

Make the gradients equal to zero

Returns
None

Return type
None

class SimulatedAnnealing

Bases: object

__init__(*model*, *type_center='neighborhood'*, *init_temp=1000000*, *radius=1*, *temp_multiplier=0.95*)

Initialization of SimulatedAnnealing algorithm. Minimize real number models (non-discrete)

Parameters

- **model** (*Module*) – some pytorch model
- **type_center** (*Literal['zero', 'neighborhood']*) – if type_center is zero, new point (x_k+1) would be chosen from Uniform[-radius, radius] for each parameter, elif neighborhood, new point would be chosen from Uniform[x_k - radius, x_k + radius).
- **init_temp** (*float*) – initial temperature. Default is 10_000
- **radius** (*float*) – ball's radius
- **temp_multiplier** (*float*) –

optimize(*x*, *y*)

Parameters

- **x** (*Tensor*) –
- **y** (*Tensor*) –

Return type
[<class 'torch.nn.modules.module.Module'>, <class 'nueramic_mathml.ml.optimize.HistorySA'>]

optimize_generator(*x*, *y*)

Generator of Simulated Annealing steps.[1]

$c = x_{pre}$ if type area is 'neighborhood' else $c = \theta - \text{zero}$
$x_{cur} \sim \mathcal{U}(c, r) \qquad p \sim \mathcal{U}[0, 1]$

if $f(x_{cur}) < f(x_{best})$:
$x_{pre} = x_{best} = x_{cur}$

[1] Van Laarhoven, P. J. M., & Aarts, E. H. L. (1987). Simulated annealing: Theory and applications (1987th ed.). Kluwer Academic. pp.10-11

$$\text{elif } \exp\left(\frac{f(x_{pre}) - f(x_{cur})}{T}\right) > p:$$
$$x_{pre} = x_{cur}$$

$$T = T \cdot \delta$$

Parameters

- **x** (*Tensor*) – training set
- **y** (*Tensor*) – target value

Returns
verbose strign with iteration and loss

Return type
str

```
>>> torch.random.manual_seed(7)

>>> xr = torch.rand(100, 3)
>>> w = torch.tensor([[1., 2., 3.]]).T
>>> yr = xr @ w + 2

>>> model = torch.nn.Sequential(torch.nn.Linear(3, 1))
>>> model.loss = lambda _x, _y: torch.nn.MSELoss()(model(_x), _y)

>>> optimizer = SimulatedAnnealing(model, temp_multiplier=0.01)

>>> for verbose in optimizer.optimize(xr, yr):
>>>     print(verbose)
iteration:    1 | loss: 97.5745
iteration:    2 | loss: 231.5806
iteration:    3 | loss: 3.4633
iteration:    4 | loss: 3.7009
iteration:    5 | loss: 26.9238
iteration:    6 | loss: 6.5509
iteration:    7 | loss: 21.4261

>>> model.loss(xr, yr)
tensor(3.4633, grad_fn=<MseLossBackward0>)
```

References

CHAPTER

TWELVE

ML.METRICS

12.1 Classification

recall	Return True Positive Rate.
fpr	Return False Positive Rate.
precision	Return Positive Predictive Value .
accuracy	Return accuracy.
f_score	Return F_score.
auc_roc	Return area under curve ROC (AUC-ROC metric)
roc_curve	Return dict with points at TPR - FPR coordinates
binary_classification_report	Returns dict with recall, precision, accuracy, f1, auc roc scores
best_threshold	Returns best threshold by metric by linear search

12.1.1 nueramic_mathml.ml.metrics.recall

recall(*y_true*, *y_pred*)

Return True Positive Rate. TPR = TP / P = TP / (TP + FN). Alias is Recall

Note: if P == 0, then TPR = 0

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification

Returns

Return type
float

12.1.2 nueramic_mathml.ml.metrics.fpr

fpr(*y_true*, *y_pred*)

Return False Positive Rate. FPR = FP / N = FP / (FP + TN).

Note: if N == 0, then FPR = 0

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification

Returns

Return type
float

12.1.3 nueramic_mathml.ml.metrics.precision

precision(*y_true*, *y_pred*)

Return Positive Predictive Value . PPV = TP / (TP + FP)

Note: if TP + FP == 0, then PPV = 0

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification

Returns

Return type
float

12.1.4 nueramic_mathml.ml.metrics.accuracy

accuracy(*y_true*, *y_pred*)

Return accuracy. ACC = (TP + TN) / (P + N) = (TP + TN) / (TP + FP + TN + FN)

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification

Returns

Return type
float

12.1.5 nueramic_mathml.ml.metrics.f_score

f_score(*y_true*, *y_pred*, *beta=1*)

Return F_score. https://en.wikipedia.org/wiki/F-score

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}. \tag{12.1}$$

Note: if beta ** 2 * _precision + _recall == 0, then f_score = 0

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification
- **beta** (*float*) – is chosen such that recall is considered beta times as important as precision

Returns

Return type
float

12.1.6 nueramic_mathml.ml.metrics.auc_roc

auc_roc(*y_true*, *y_prob*, *n_thresholds=500*)

Return area under curve ROC (AUC-ROC metric)

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_prob** (*Tensor*) – array of probabilities of confidence of belonging to the 1st class
- **n_thresholds** (*int*) – if len(y_true) is too large, you can limit the number of threshold values

Returns
float value of area under roc-curve

Return type
float

12.1.7 nueramic_mathml.ml.metrics.roc_curve

roc_curve(*y_true*, *y_prob*, *n_thresholds=None*)

Return dict with points at TPR - FPR coordinates

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_prob** (*Tensor*) – array of probabilities of confidence of belonging to the 1st class
- **n_thresholds** (*Optional[int]*) – if len(y_true) is too large, you can limit the number of threshold values

Returns
dict with values of TPR and FPR

Return type
Dict

12.1.8 nueramic_mathml.ml.metrics.binary_classification_report

binary_classification_report(*y_true*, *y_pred*, *y_prob=None*)

Returns dict with recall, precision, accuracy, f1, auc roc scores

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification
- **y_prob** (*Optional[Tensor]*) – array of probabilities of confidence of belonging to the 1st class

Returns
dict with 5 metrics

Return type
dict

12.1.9 nueramic_mathml.ml.metrics.best_threshold

best_threshold(*x*, *y_true*, *model*, *metric='f1'*, *step_size=0.01*)

Returns best threshold by metric by linear search

Parameters

- **x** (*Tensor*) – training tensor
- **y_true** (*Tensor*) – target tensor. array with true values of binary classification
- **model** (*Module*) – some model that returns a torch tensor with class 1 probabilities using the call: model(x)
- **metric** (*Literal['f1', 'by_roc']*) – name of the target metric that we need to maximize. by_roc - difference between TPR and FPR
- **step_size** (*float*) – step size of linear search

Returns

12.2 Regression

r2_score	Return R2 metric of regression
mse	Returns MSE
mae	Returns MAE
mape	Returns MAPE
regression_report	Returns dict with recall, precision, accuracy, f1, auc roc scores

12.2.1 nueramic_mathml.ml.metrics.r2_score

r2_score(*y_true*, *y_pred*)

Return R2 metric of regression

$$\mathbf{R}^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(\hat{y}_i - \overline{y})^2} \tag{12.2}$$

Note: if std(y_true) = 0, then r2 = 0

Parameters

- **y_true** (*Tensor*) – array with true values of regression
- **y_pred** (*Tensor*) – array with prediction values of regression

Returns
r2 metric in float number

Return type
float

12.2.2 nueramic_mathml.ml.metrics.mse

mse(*y_true*, *y_pred*)

Returns MSE

$$\mathbf{MSE}^2 = \tag{12.3}$$

rac{1}{n}sum_{i=1}^{n}(hat y_i - y_i)^2}

param y_true
array with true values of regression

param y_pred
array with prediction values of regression

return
mse metric in float number

Parameters

- **y_true** (*Tensor*) –
- **y_pred** (*Tensor*) –

Return type
float

12.2.3 nueramic_mathml.ml.metrics.mae

mae(*y_true*, *y_pred*)

Returns MAE

$$\mathbf{MSE}^2 = \tag{12.4}$$

rac{1}{n}sum_{i=1}^{n}|hat y_i - y_i|

param y_true
array with true values of regression

param y_pred
array with prediction values of regression

return
mae metric in float number

Parameters

- **y_true** (*Tensor*) –
- **y_pred** (*Tensor*) –

Return type
float

12.2.4 nueramic_mathml.ml.metrics.mape

mape(*y_true*, *y_pred*)

Returns MAPE

$$\sum_{i=1}^{n} \left| \frac{A_i - F_i}{A_t} \right| \tag{12.5}$$

Note: All values in y_true that are less than 1e-10 in absolute value will be replaced by 1e-10

Parameters

- **y_true** (*Tensor*) – array with true values of regression
- **y_pred** (*Tensor*) – array with prediction values of regression

Returns
mape metric in float number

Return type
float

12.2.5 nueramic_mathml.ml.metrics.regression_report

regression_report(*y_true*, *y_pred*)

Returns dict with recall, precision, accuracy, f1, auc roc scores

Parameters

- **y_true** (*Tensor*) – array with true values of binary classification
- **y_pred** (*Tensor*) – array with prediction values of binary classification

Returns
dict with 4 metrics

Return type
dict

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

n

nueramic_mathml.calculus, 4
nueramic_mathml.ml.classification, 26
nueramic_mathml.ml.optimize, 39

INDEX

Symbols

__init__() (*BaseClassification method*), 26
__init__() (*BaseRegressionModel method*), 32
__init__() (*ExponentialRegression method*), 35
__init__() (*LinearRegression method*), 34
__init__() (*LogisticRegression method*), 26
__init__() (*LogisticRegressionRBF method*), 28
__init__() (*NueSGD method*), 39
__init__() (*PolynomialRegression method*), 34
__init__() (*SVM method*), 29
__init__() (*SimulatedAnnealing method*), 40
_fit_pegasos() (*SVM method*), 29
_fit_sgd() (*SVM method*), 30

A

accuracy() (*in module nueramic_mathml.ml.metrics*), 43
auc_roc() (*in module nueramic_mathml.ml.metrics*), 44

B

BaseClassification (*class in nueramic_mathml.ml.classification*), 26
BaseRegressionModel (*class in nueramic_mathml.ml.regression*), 32
best_threshold() (*in module nueramic_mathml.ml.metrics*), 45
bfgs() (*in module nueramic_mathml.multi_optimize*), 19
binary_classification_report() (*in module nueramic_mathml.ml.metrics*), 45
brent() (*in module nueramic_mathml.one_optimize*), 9

C

constrained_lagrangian_solver() (*in module nueramic_mathml.multi_optimize*), 20

E

ExponentialRegression (*class in nueramic_mathml.ml.regression*), 34

F

f_score() (*in module nueramic_mathml.ml.metrics*), 44
fit() (*BaseRegressionModel method*), 32
fit() (*ExponentialRegression method*), 35
fit() (*LogisticRegression method*), 26
fit() (*LogisticRegressionRBF method*), 28
fit() (*SVM method*), 30
forward() (*BaseRegressionModel method*), 33
forward() (*ExponentialRegression method*), 35
forward() (*LogisticRegression method*), 27
forward() (*LogisticRegressionRBF method*), 28
forward() (*SVM method*), 30
fpr() (*in module nueramic_mathml.ml.metrics*), 43

G

gd_constant() (*in module nueramic_mathml.multi_optimize*), 15
gd_frac() (*in module nueramic_mathml.multi_optimize*), 16
gd_optimal() (*in module nueramic_mathml.multi_optimize*), 17
gen_animated_surface() (*in module nueramic_mathml.visualize.multi_animation*), 23
gen_animation_brent() (*in module nueramic_mathml.visualize.one_animation*), 12
gen_animation_gss() (*in module nueramic_mathml.visualize.one_animation*), 12
gen_animation_spi() (*in module nueramic_mathml.visualize.one_animation*), 12
gen_classification_plot() (*in module nueramic_mathml.visualize.ml_animation*), 36
gen_regression_plot() (*in module nueramic_mathml.visualize.ml_animation*), 37
gen_simple_gradient() (*in module nueramic_mathml.visualize.multi_animation*), 23
golden_section_search() (*in module nueramic_mathml.one_optimize*), 7

gradient() (*in module nueramic_mathml.calculus*), 4

H

hessian() (*in module nueramic_mathml.calculus*), 4
HistorySA (*class in nueramic_mathml.ml.optimize*), 39

I

init_weights() (*BaseRegressionModel method*), 33
init_weights() (*LogisticRegression method*), 27
init_weights() (*SVM method*), 30
iteration (*HistorySA attribute*), 39

J

jacobian() (*in module nueramic_mathml.calculus*), 5

L

LinearRegression (*class in nueramic_mathml.ml.regression*), 34
log_barrier_solver() (*in module nueramic_mathml.multi_optimize*), 21
LogisticRegression (*class in nueramic_mathml.ml.classification*), 26
LogisticRegressionRBF (*class in nueramic_mathml.ml.classification*), 27
loss (*HistorySA attribute*), 39

M

mae() (*in module nueramic_mathml.ml.metrics*), 47
make_contour() (*in module nueramic_mathml.visualize.multi_animation*), 24
make_phi_matrix() (*LogisticRegressionRBF method*), 28
make_surface() (*in module nueramic_mathml.visualize.multi_animation*), 24
mape() (*in module nueramic_mathml.ml.metrics*), 47
metrics_tab() (*BaseClassification method*), 26
metrics_tab() (*BaseRegressionModel method*), 33
module
    nueramic_mathml.calculus, 4
    nueramic_mathml.ml.classification, 26
    nueramic_mathml.ml.optimize, 39
mse() (*in module nueramic_mathml.ml.metrics*), 46

N

nonlinear_cgm() (*in module nueramic_mathml.multi_optimize*), 18
nueramic_mathml.calculus
    module, 4
nueramic_mathml.ml.classification
    module, 26
nueramic_mathml.ml.optimize
    module, 39
NueSGD (*class in nueramic_mathml.ml.optimize*), 39

O

optimize() (*NueSGD method*), 39
optimize() (*SimulatedAnnealing method*), 40
optimize_generator() (*SimulatedAnnealing method*), 40

P

parabolic_coefficients() (*in module nueramic_mathml.visualize.one_animation*), 13
point (*HistorySA attribute*), 39
PolynomialRegression (*class in nueramic_mathml.ml.regression*), 34
precision() (*in module nueramic_mathml.ml.metrics*), 43
predict() (*LogisticRegression method*), 27
predict() (*LogisticRegressionRBF method*), 28
predict() (*SVM method*), 31
primal_dual_interior() (*in module nueramic_mathml.multi_optimize*), 22

R

r2_score() (*in module nueramic_mathml.ml.metrics*), 46
recall() (*in module nueramic_mathml.ml.metrics*), 42
regression_report() (*in module nueramic_mathml.ml.metrics*), 48
roc_curve() (*in module nueramic_mathml.ml.metrics*), 44
roc_curve_plot() (*in module nueramic_mathml.visualize.ml_animation*), 37

S

scaler() (*SVM method*), 31
SimulatedAnnealing (*class in nueramic_mathml.ml.optimize*), 40
step() (*NueSGD method*), 39
successive_parabolic_interpolation() (*in module nueramic_mathml.one_optimize*), 8
SVM (*class in nueramic_mathml.ml.classification*), 29

T

transfer_history_gss() (*in module nueramic_mathml.visualize.one_animation*), 13
transform() (*BaseRegressionModel static method*), 33
transform() (*PolynomialRegression method*), 34
type_ball (*HistorySA attribute*), 39

Z

zero_grad() (*NueSGD method*), 40