Лабораторная Работа №4

Методы стохастической оптимизации. Настройка гиперпараметров

Колтаков Максим М3234

Рязанова Екатерина М3234

Хайруллин Артур М3234

Оглавление

Метод отжига

Пусть имеется некоторая функция $f(x)$ от *состояния* x, которую мы хотим минимизировать.

Возьмем в качестве базового решения какое-то состояние x_0 и будем пытаться его улучшать.

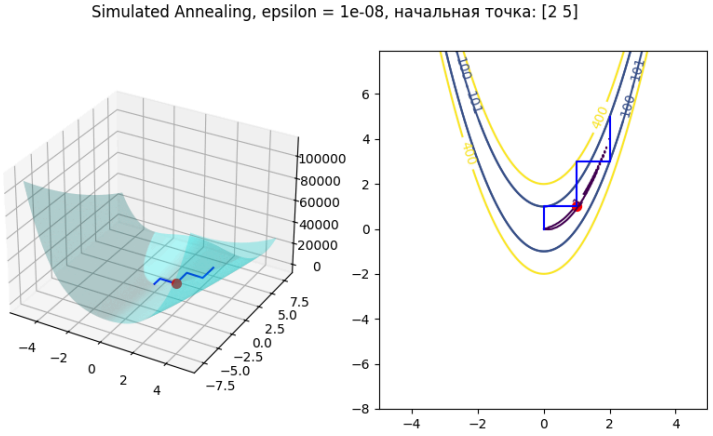
Введем *температуру* t — какое-то действительное число (изначально равное единице), которое будет изменяться в течение оптимизации и влиять на вероятность перейти в соседнее состояние.

Пока не придем к оптимальному решению или пока не закончится время, будем повторять следующие шаги:

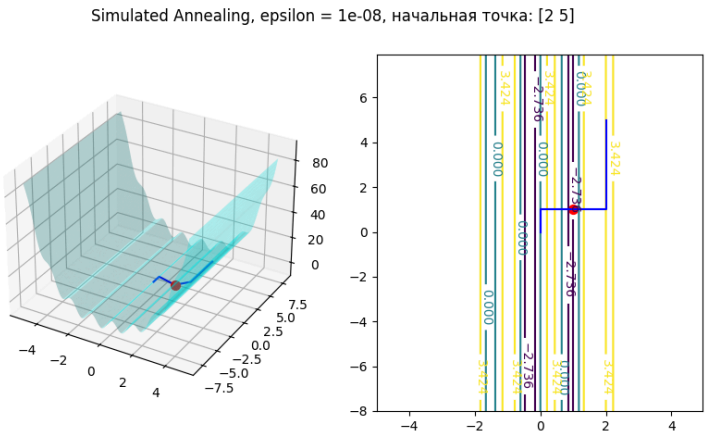
1) Уменьшим температуру $tk=T(tk-1)$.
2) Выберем случайного *соседа* x — то есть какое-то состояние y, которое может быть получено из x каким-то минимальным изменением.
3) С вероятностью $p(f(x),f(y),tk)$ сделаем присвоение $x \leftarrow y$.

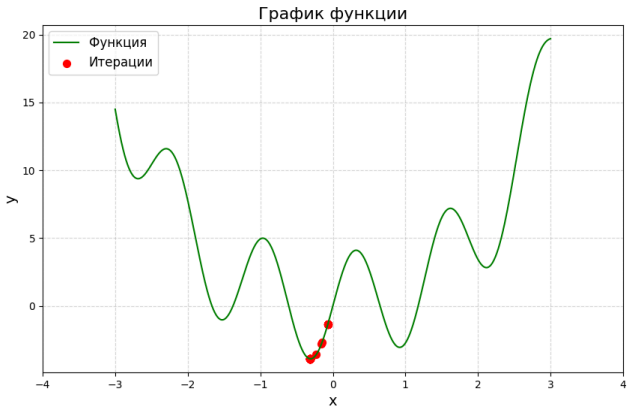
В каждом шаге есть много свободы при реализации. Основные эвристические соображения следующие:

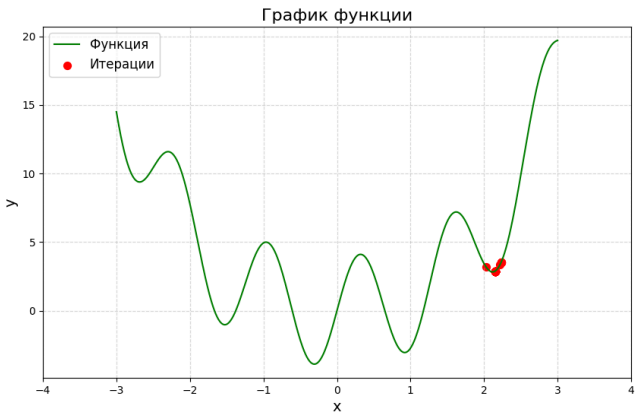
1. В начале оптимизации наше решение и так плохое, и мы можем позволить себе высокую температуру и риск перейти в состояние хуже. В конце наоборот — наше решение почти оптимальное, и мы не хотим терять прогресс. Температура должна быть высокой в начале и медленно уменьшаться к концу.
2. Алгоритм будет работать лучше, если функция $f(x)$ «гладкая» относительно этого изменения, то есть изменяется не сильно.
3. Вероятность должна быть меньше, если новое состояние хуже, чем старое. Также вероятность должна быть больше при высокой температуре.



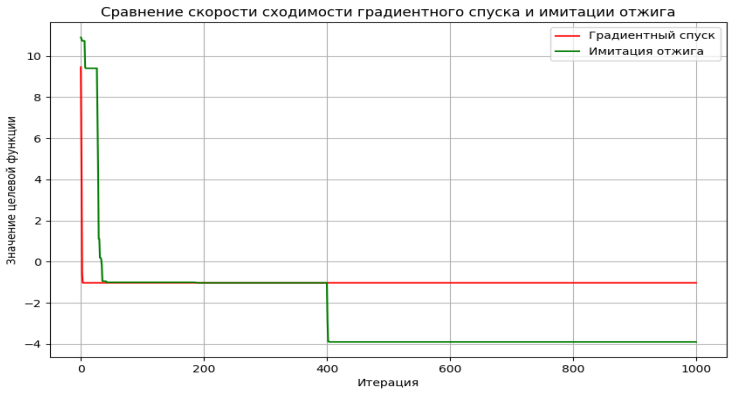
На графиках ниже визуализирована работа метода имитации отжига. С помощью метода отжига мы не всегда получаем именно глобальный минимум.

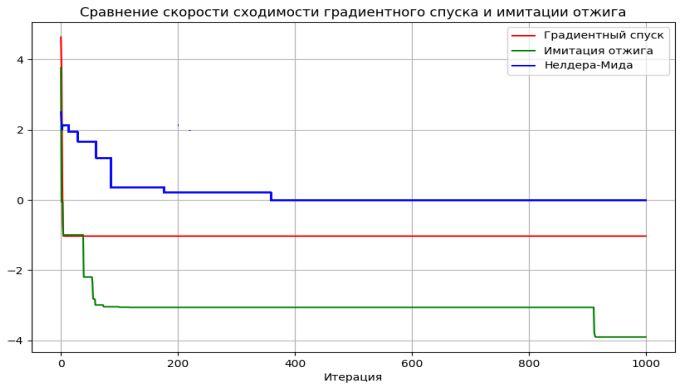






Сравнение





Метод	Найденное решение	Время (с)	Количество вызовов	Количество итераций
Градиентный спуск	-1.5256954130637503	0.091452419875614717	2001	1000
Имитация отжига	0.9200049336061964	0.020815610885620117	1001	1000
Нелдера-Мида	-0.3094036279247888	0.17584352493286133	2001	1000

Имитация отжига работает быстрее, чем методы из 1 лабораторной (Градиентный спуск, Нелдера-Мида), меньше и кол-во итераций, и вызовов функции. Однако, за счет

этого уменьшается и точность работы алгоритма. Градиентный спуск вычислил решение с очень точной оценкой.

Дополнительное задание 1

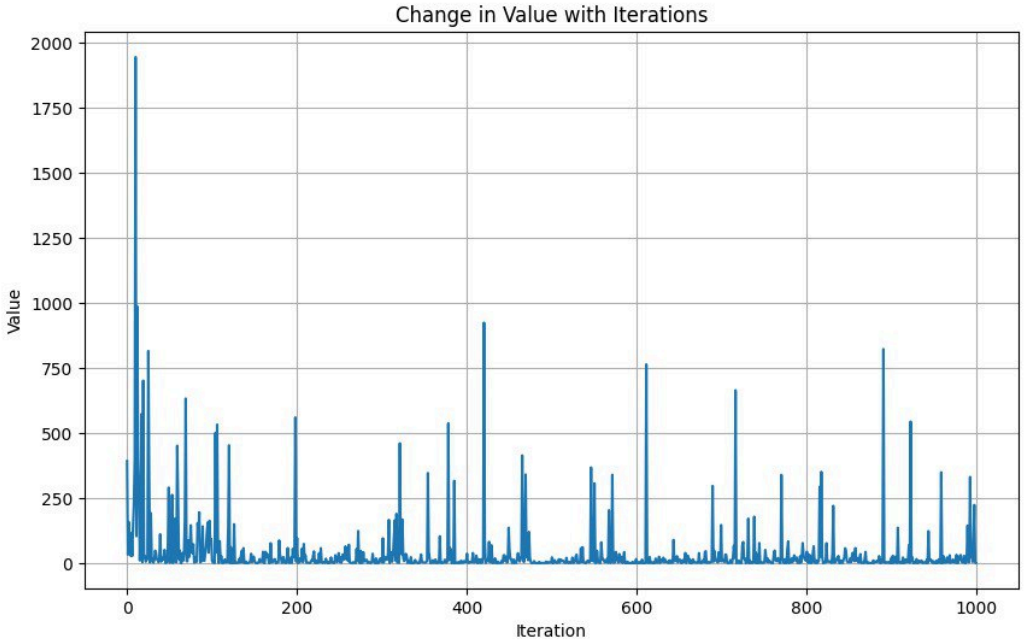
Выберем функцию Розенброка: $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$
Рассмотрим задачу: минимизировать значение функции f при её гиперпараметрах x и y. Для подбора гиперпараметров используем optuna.

```
def optimize(func):  2 usages (1 dynamic)  new *
    def objective(trial):  new *
        x = trial.suggest_float("x", -2.0, 2.0)
        y = trial.suggest_float("y", -2.0, 2.0)
        d = func(x, y)
        return d

    study = optuna.create_study(direction="minimize", sampler=optuna.samplers.TPESampler())
    study.optimize(objective, n_trials=1000)
    print(study.best_params)
    print(study.best_value)
```

Если ограничивать x и y отрезком [-10, 10], а количеством испытаний выбрать 10000, также сэмплером выбрать TPE, то результат будет таков:

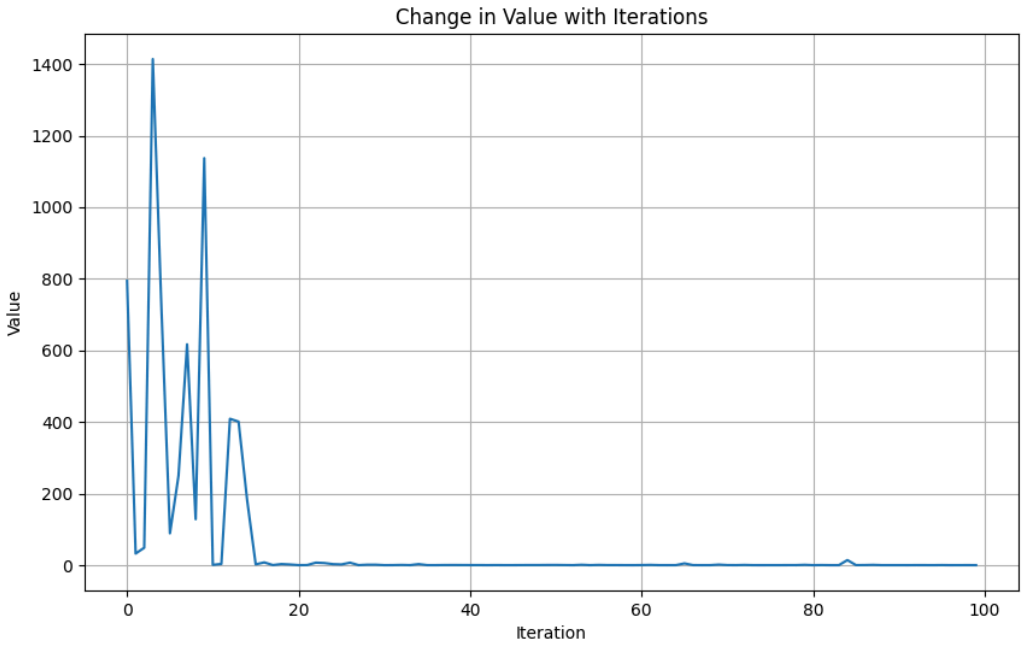
```
[I 2024-06-12 20:05:47,716] Trial 9997 finished with valu
[I 2024-06-12 20:05:47,819] Trial 9998 finished with valu
{'x': 1.00050295475855 95, 'y': 1.0003182429632056}
4.75762897315234 95e-05
[I 2024-06-12 20:05:47,916] Trial 9999 finished with valu
```

Что почти не отличается от ожидаемого (0 в точке 1, 1).
Если ограничить x числами -2, 2, а y числами -1, 3, количеством испытаний выбрать 1000, то получим результат:
{'x': 1.011987558327289, 'y': 1.0238277735057422}
0.00015217225659288592
Тоже хорошая точность. График показывает какое значение было получено на каждой итерации:

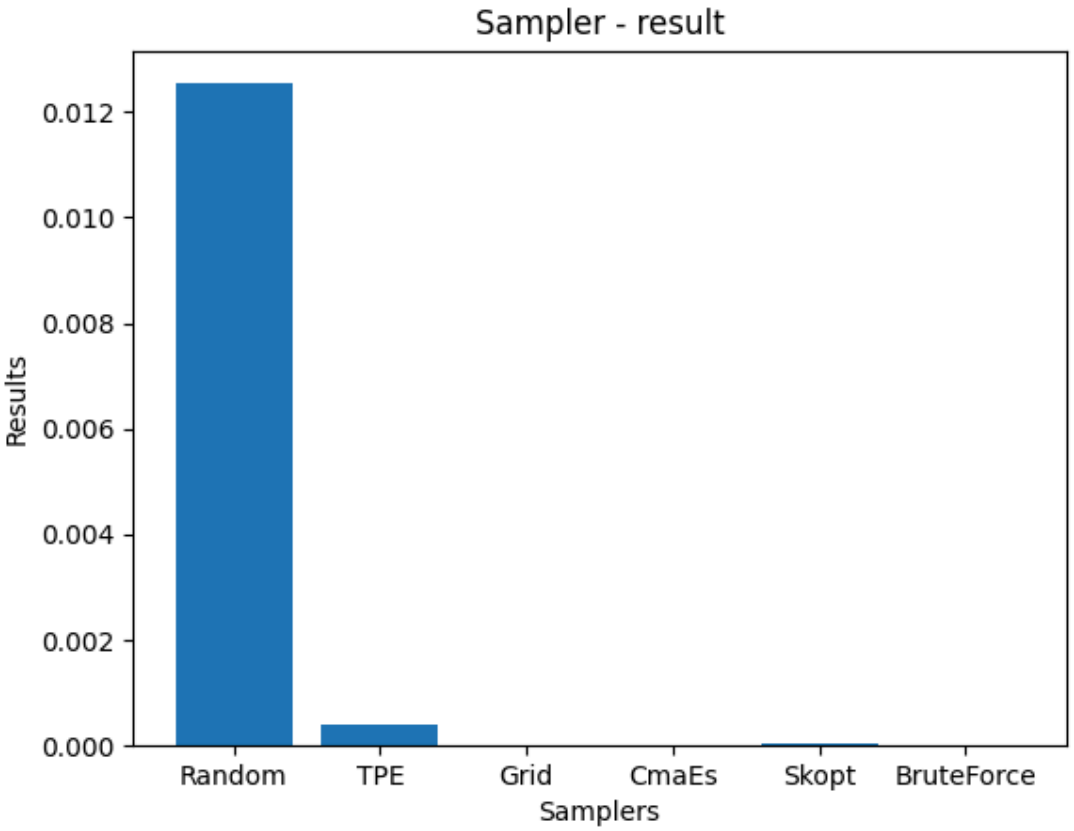


Бывают разные сэмплеры:
RandomSampler: Выбирает значения параметров случайным образом из заданных диапазонов.
TPESampler: Использует метод Tree-structured Parzen Estimator для эффективного подбора значений параметров.
CmaEsSampler: Использует алгоритм Covariance Matrix Adaptation Evolution Strategy (CMA-ES) для генерации новых точек испытания.
GridSampler: Проводит поиск по сетке значений параметров.
SkoptSampler: Использует библиотеку scikit-optimize для оптимизации.
BruteForceSampler: Представляет собой метод перебора всех возможных комбинаций значений параметров для поиска оптимального решения.

Рассмотрим разные сэмплеры при ограничениях на x и y от -2 до 2 и 1000 испытаниях.
RandomSampler: {'x': 1.0034842711659895, 'y': 1.0181714963387294}
0.012535571633143406
TPESampler: {'x': 0.9896399047567047, 'y': 0.9776485982583639}
0.000409584690101503
GridSampler: {'x': 1.0, 'y': 1.0}
0.0 (такой хороший результат, потому что сетка значений включала числа -2, 1 и 2 для каждого параметра).
CmaEsSampler: {'x': 0.9999999999999964, 'y': 0.999999999999992}
9.150786500563737e-29
SkoptSampler: {'x': 0.9953572891476115, 'y': 0.9903279865964727}
3.821311756924166e-05 (пришлось поставить 100 испытаний из-за времени выполнения, однако график распределения интересный)

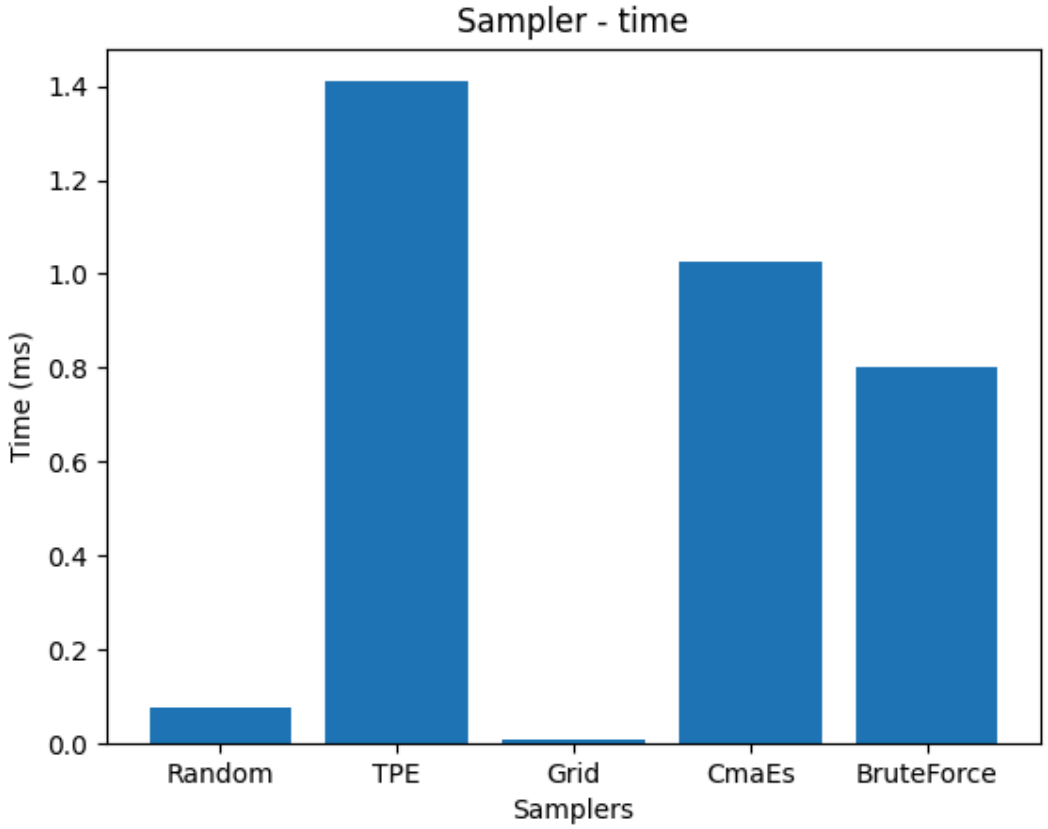


BruteForceMethod: {'x': 1.0, 'y': 1.0}
0.0 (при шаге в 0.12 было сделано 1156 испытаний)



Рассмотрим время работы при 100 испытаниях (в миллисекундах):
GridSampler: 0.008416414260864258
TPESampler: 1.4095995426177979

CmaEsSampler: 1.0267491340637207
BruteForceSampler: 0.8021633625030518 (при шаге 0.4 сделано 121 испытание)
RandomSampler: 0.07725787162780762
SkoptSampler: 106.05947279930115



На диаграмме нет SkoptSampler, так как на его фоне остальные данные были бы не видны.
По умолчанию используется TPESampler, как можно видеть, он довольно точный, но притом работает дольше среднего. Самый быстрый - GridSampler, но он не генерирует выборку сам, а берёт из данной. Дальше идёт RandomSampler - самый неточный из всех. Хорошую точность имеет CmaEsSampler, но скорость работы средняя. При использовании BruteForceSampler смутило то, что нужно указывать шаг (притом если количество испытаний выбрать самому, то их может оказаться мало для полного перебора). SkoptSampler работает чрезвычайно долго, но точность имеет неплохую. На мой взгляд, TPE и CmaES - хороший выбор.

Дополнительное задание 2

Применение методов из optuna к гиперпараметрам из лаб. 2. Для исследования была выбрана функция Розенброка.
Для методов (кроме метода Ньютона с условием Вольфе) подбирался единственный гиперпараметр - начальная точка. eps всегда оставался 10^-8. Оптимизируемая метрика - норма разности между подсчитанным вручную минимум функции (точка 1, 1) и полученным на последней итерации алгоритма.

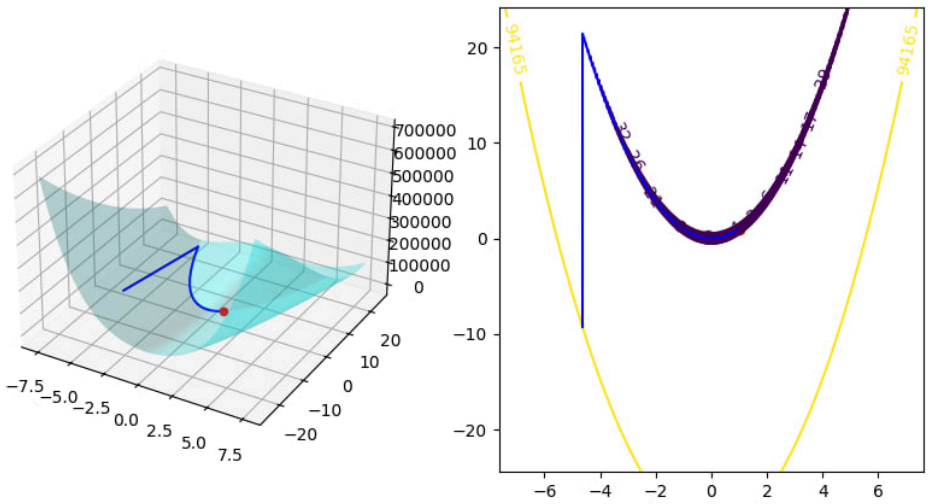
```python
def objective(trial):  1 usage  new *
    f = rosenbrock
    f_grad = grad_rosenbrock
    f_hessian = hessian_rosenbrock
    start_point_x = trial.suggest_float('start_point_x', -10.0, 10.0)
    start_point_y = trial.suggest_float('start_point_y', -10.0, 10.0)
    start_point = np.array([start_point_x, start_point_y])
    eps = 1e-8
    cur_x, stat, xs = method[0](f, f_grad, f_hessian, start_point, eps)
    d = np.linalg.norm(np.array([1, 1]) - cur_x)
    trial.set_user_attr('stat', stat)
    trial.set_user_attr('x', cur_x)
    trial.set_user_attr('xs', xs)
    return d
```

В запуске для каждого метода было проведено 15 испытаний. Выводились следующие результаты:

Метод Ньютона

```
================================================
Function name: rosenbrock_function

Method: Newton
start_point_x: -4.6280294955218215
start_point_y: -9.262460453395212
x:  [1. 1.]
y:  1.232595164407831e-30
time:            0.025997877120971 68
memory:          3663
function_calls:  944
gradient_calls:  22
hessian_calls:   22
iterations:      22
```

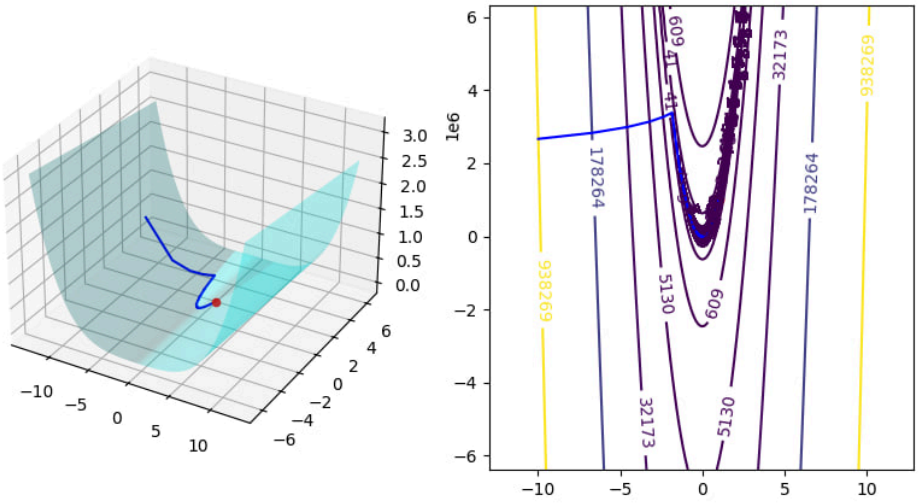
Newton для функции Rosenbrock, epsilon = 1e-08, начальная точка: [-4.6280295 -9.26246045]



Метод Ньютона с сопряжёнными градиентами:

```
==================================================
Function name: rosenbrock_function

Method: Newton-CG
start_point_x: -9.976275636634496
start_point_y: 2.668022834813715
x:  [1. 1.]
y:  8.3007336548207e-19
time:              0.3330252170562744
memory:            87176
function_calls:    640
gradient_calls:    2270
hessian_calls:     0
iterations:        616
```

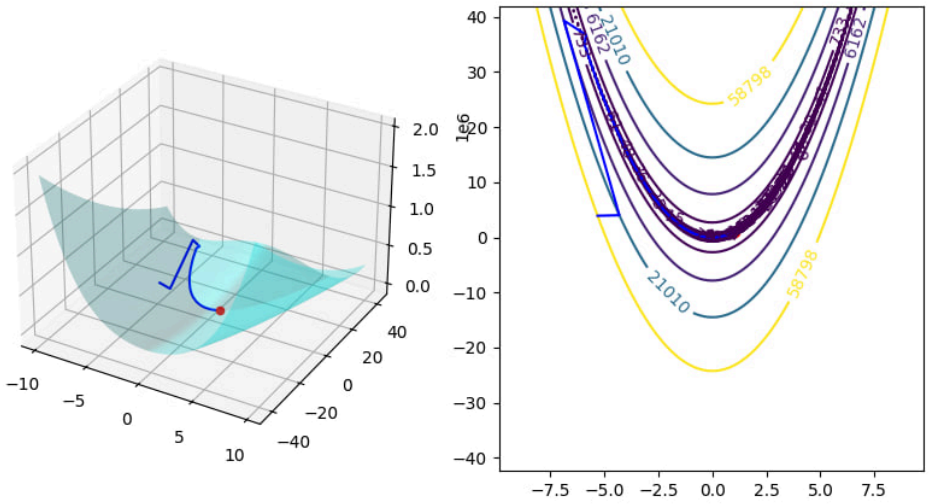
Newton-CG для функции Rosenbrock, epsilon = 1e-08, начальная точка: [-9.97627564 2.66802283]



BFGS:

```
==========================================
Function name: rosenbrock_function

Method: Quasinewton (BFGS)
start_point_x: -5.306293580533328
start_point_y: 3.916617599469909
x:  [1. 1.]
y:  2.099184753291565e-24
time:            0.024999856948852 54
memory:          12976
function_calls:  89
gradient_calls:  89
hessian_calls:   0
iterations:      71
```

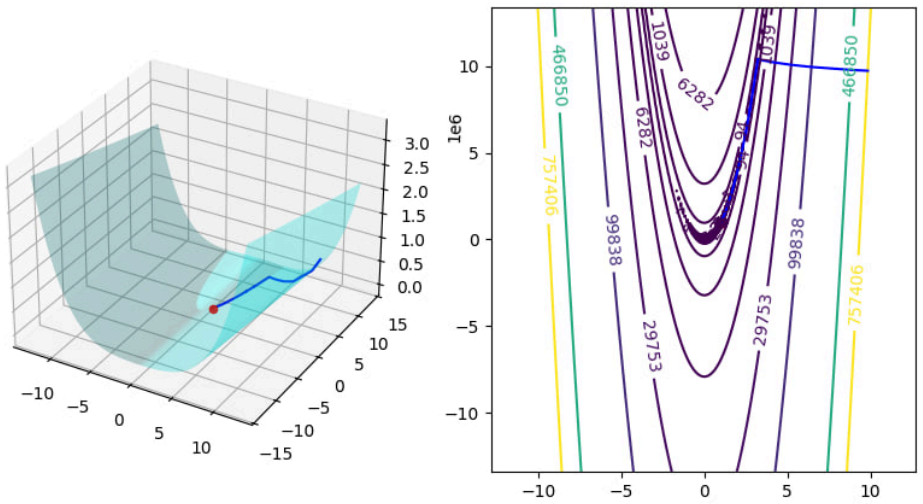
Quasinewton (BFGS) для функции Rosenbrock, epsilon = 1e-08, начальная точка: [-5.30629358 3.9166176]



L-BFGS-B:

```
================================================
Function name: rosenbrock_function

Method: Quasinewton (L-BFGS-B)
start_point_x: 9.837358001856087
start_point_y: 9.74901847954851
x:  [1.00000001 1.00000001]
y:  3.702591816601879e-16
time:            0.004999876022338867
memory:          19912
function_calls:  59
gradient_calls:  59
hessian_calls:   0
iterations:      46
```

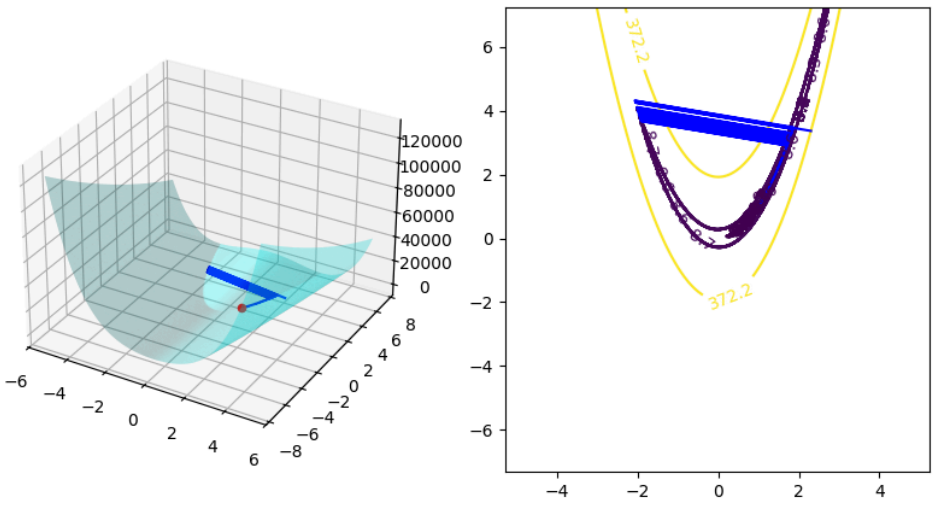
Quasinewton (L-BFGS-B) для функции Rosenbrock, epsilon = 1e-08, начальная точка: [9.837358 9.74901848]



Градиентный спуск:

```
==================================================
Function name: rosenbrock_function

Method: Gradient descend
start_point_x: 2.301186270981491
start_point_y: 3.3706072446821294
x:  [1.00000001 1.00000002]
y:  1.1046075484558254e-16
time:            0.30100464820861816
memory:          73536
function_calls:  30016
gradient_calls:  535
hessian_calls:   0
iterations:      535
```

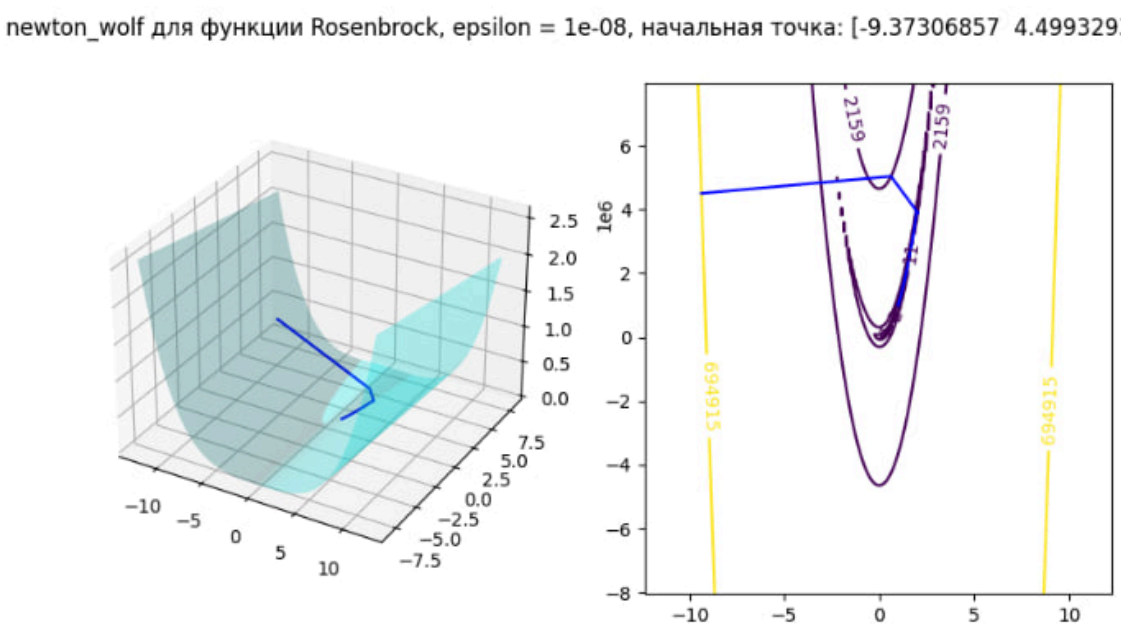
Gradient descend для функции Rosenbrock, epsilon = 1e-08, начальная точка: [2.30118627 3.37060724]



Для **метода Ньютона с условием Вольфе** подбиралась начальная точка и параметры c1 (от 0 до 1), c2 (от c1 до 1), beta (от 0.5 до 1) в условии Вольфе:

```python
def objective_wolf(trial):  1 usage  new *
    f = rosenbrock
    f_grad = grad_rosenbrock
    f_hessian = hessian_rosenbrock
    start_point_x = trial.suggest_float('start_point_x', -10.0, 10.0)
    start_point_y = trial.suggest_float('start_point_y', -10.0, 10.0)
    start_point = np.array([start_point_x, start_point_y])
    c1 = trial.suggest_float('c1', 0.0, 1.0)
    c2 = trial.suggest_float('c2', c1, 1.0)
    beta = trial.suggest_float('beta', 0.5, 1.0)
    eps = 1e-8
    cur_x, stat, xs = newton_with_wolf(f, f_grad, f_hessian, start_point, c1, c2, beta, eps)
    d = np.linalg.norm(np.array([1, 1]) - cur_x)
    trial.set_user_attr('stat', stat)
    trial.set_user_attr('x', cur_x)
    trial.set_user_attr('xs', xs)
    return d
```

```
==========================================================
Function name: rosenbrock_function

Method: Newton with wolf
start_point_x: -4.890984957077878
start_point_y: -5.520089003457668
c1: 0.32766160897644026
c2: 0.7970902834605907
beta: 0.6267190820988444
x:  [1. 1.]
y:  1.1482856551623353e-28
time:             0.007998704910278320
memory:           4631
function_calls:   166
gradient_calls:   190
hessian_calls:    24
iterations:       24
```



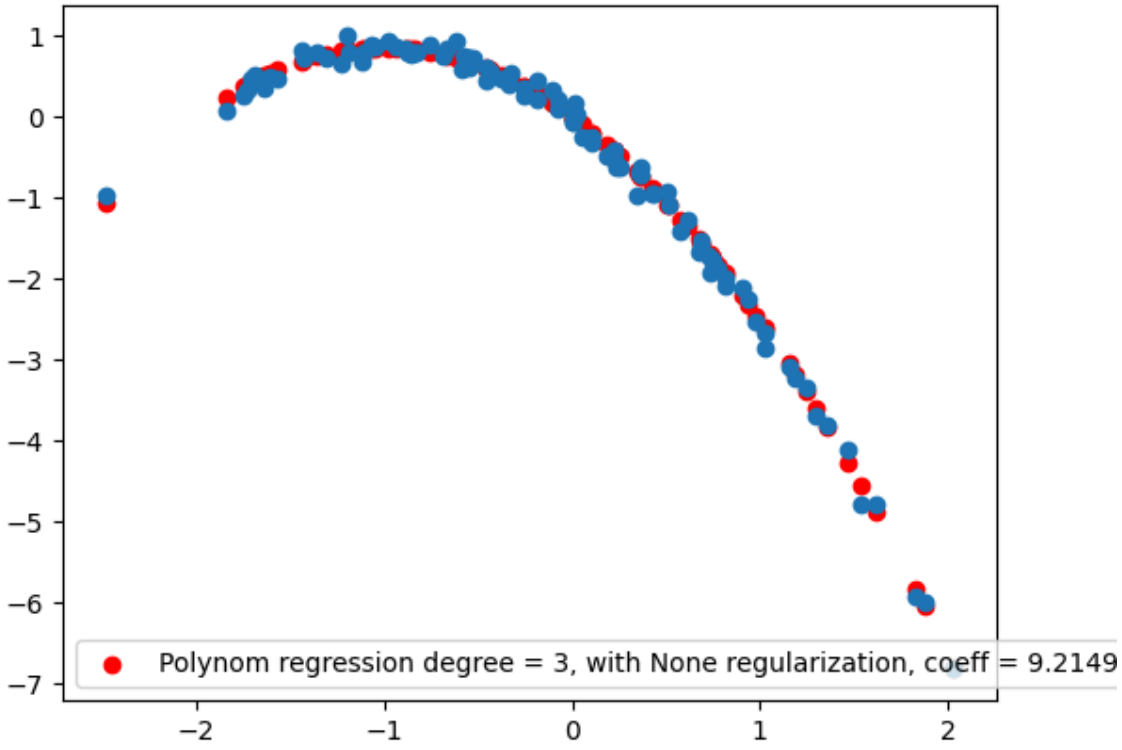
Можно заметить, что подобранные гиперпараметры себя оправдывают - результаты и правда хорошие. Особенно у метода Ньютона с условием Вольфе - время сократилось на порядок и число итераций уменьшилось в 7.75 раз (раньше параметры были c1=0.01, c2=0.5, beta=0.9).

Применение методов из optuna к гиперпараметрам из лаб. 3.
Метод полиномиальной регрессии, гиперпараметры которые мы будет изменять:

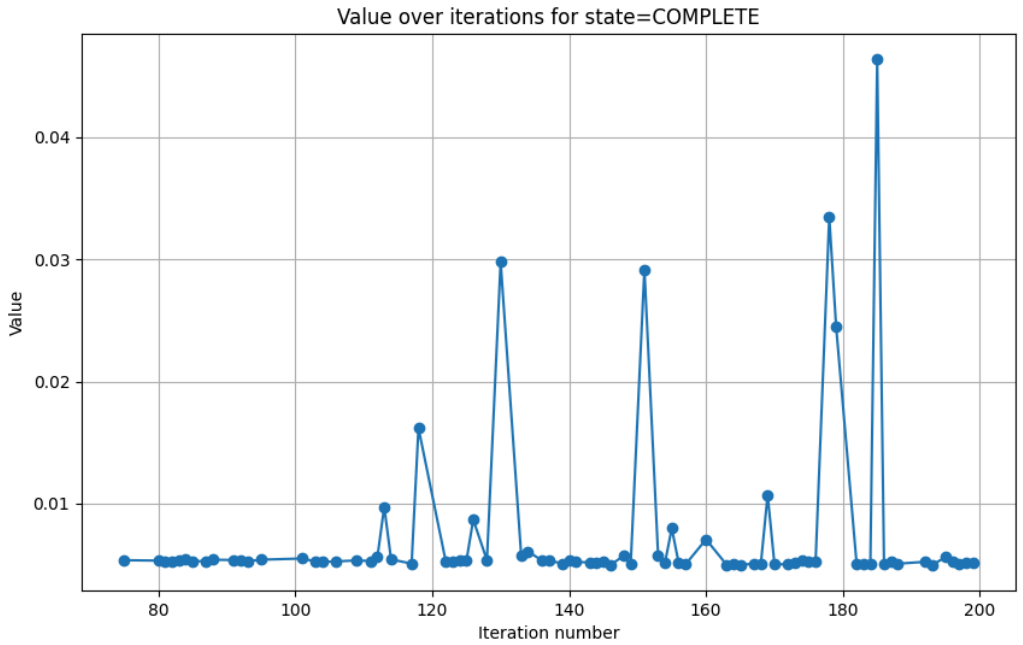
learning rate, regularization coefficient, batch size, power, type of regularization.

```
def objective(trial):  1 usage
    global m_power
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-1, log=True)
    regularization_coeff = trial.suggest_float('regularization_coeff', 0.0, 30.0)
    batch_size = trial.suggest_int('batch_size', 1, 100)
    m_power = trial.suggest_int('power', 1, 10)
    reg_name = trial.suggest_categorical('regularization', ['L1', 'L2', 'Elastic', 'None'])
    reg = regularization[reg_name]

    start = np.array([2.0] * m_power)
    theta, cost_history, _ = polynomial_regression(X, y, start, learning_rate, iterations: 1000,
                                                   reg, regularization_coeff, batch_size, m_power)

    return cost_history[-1]
```

Когда точки распределяются квадратично, получили:



Best hyperparameters: {'learning_rate': 0.014302513164071292, 'regularization_coeff': 9.214938027599523, 'batch_size': 69, 'power': 3, 'regularization': 'None'}
The best value for the objective function is: 0.005257646217146001

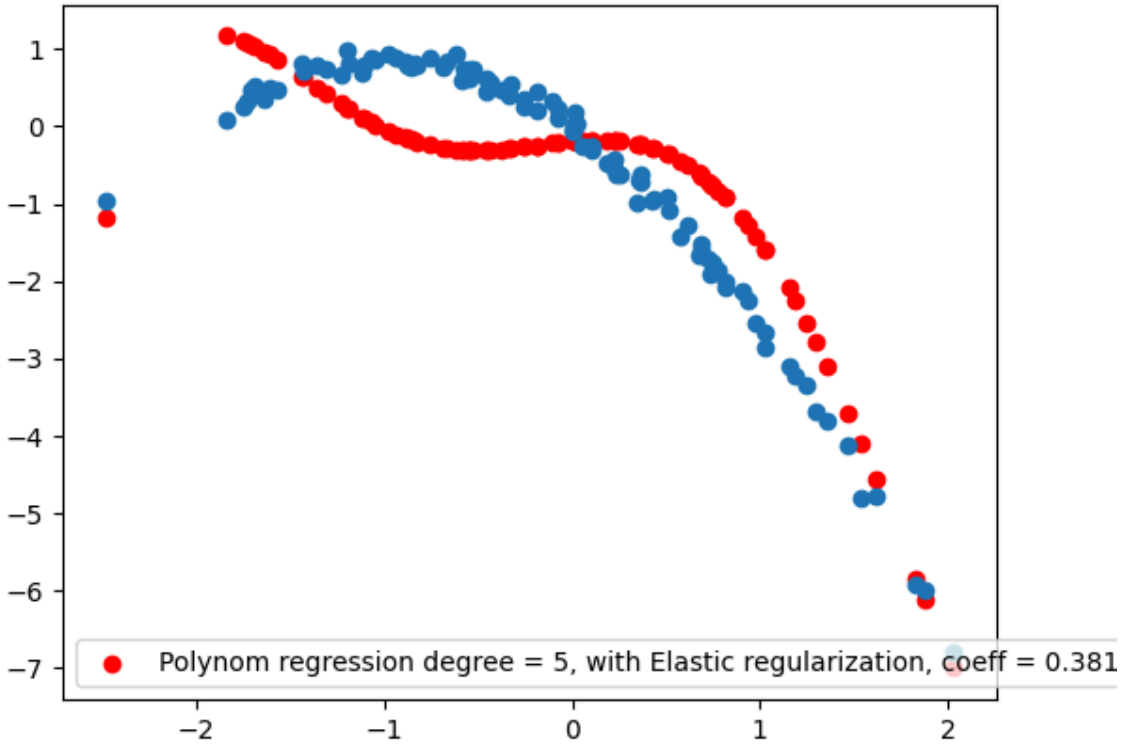


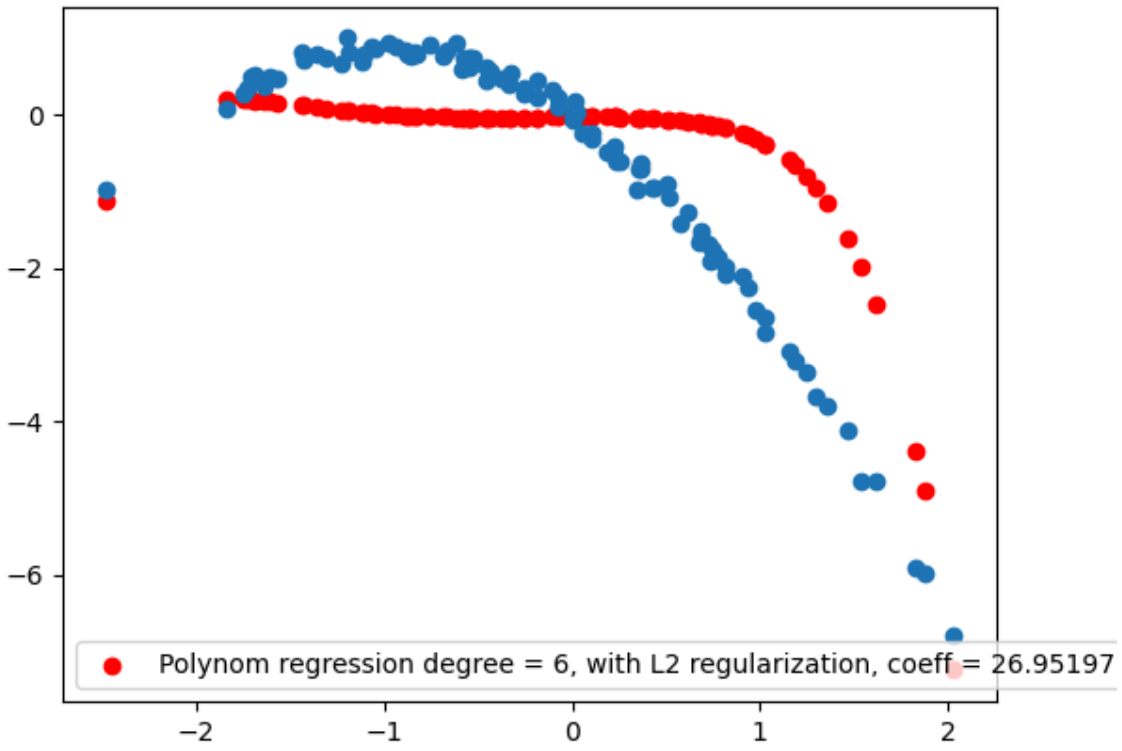
Вот как менялась погрешность от числа итераций в optune
Как и ожидалось библиотека выбрала значение m равное 3 и не выбрала никакой регуляризации, потому что кубическая функция лучше всего приближает наши точки.

Теперь попробуем ограничить m в диапазоне [5, 10]

Best hyperparameters: {'learning_rate': 0.0007792170490537096, 'regularization_coeff': 0.3817549006441767, 'batch_size': 85, 'power': 5, 'regularization': 'Elastic'}
The best value for the objective function is: 0.9114007538683053

После 200 итераций оптуны выбралась степень 5 и Elastic регуляризация.


1
0
−1
−2
−3
−4
−5
−6
−7
−2
−1
0
1
2
Polynom regression degree = 5, with Elastic regularization, coeff = 0.381

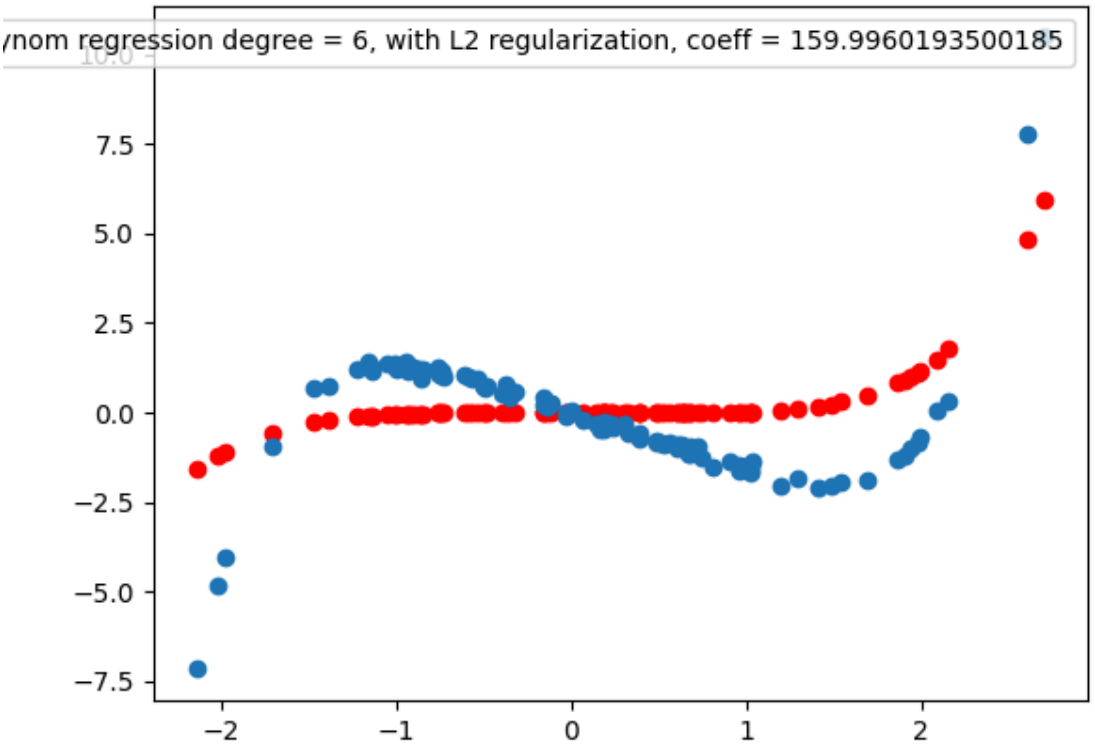


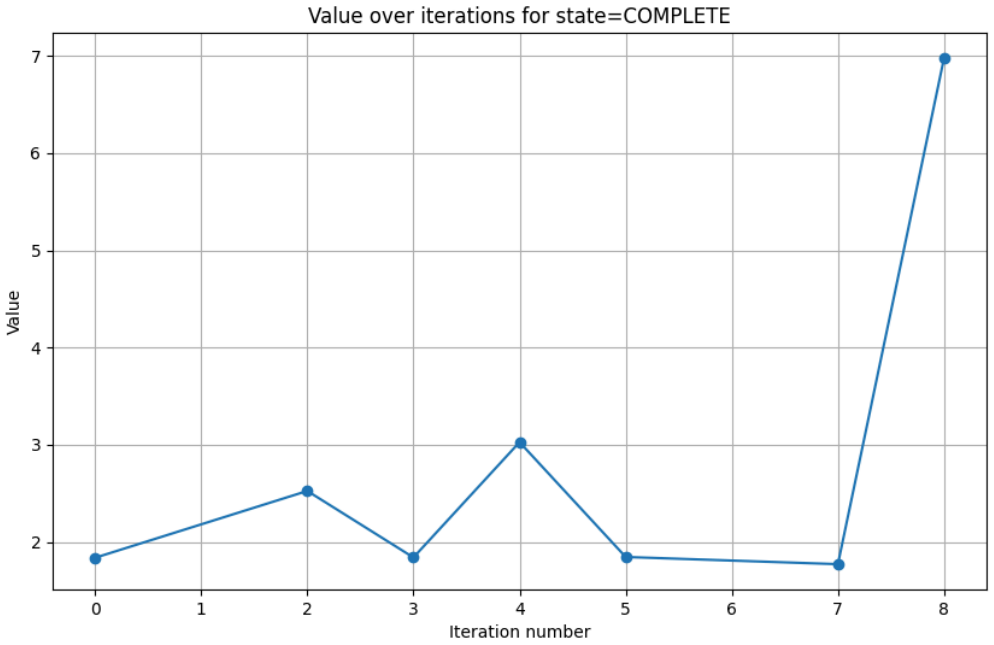
Вот такой результат если ограничить степень полинома от [6, 10]
Best hyperparameters: {'learning_rate': 6.175812748845313e-05, 'regularization_coeff': 26.951973464692315, 'batch_size': 53, 'power': 7, 'regularization': 'L2'}
The best value for the objective function is: 1.4602103945404847

Теперь возьмём точки в виде полинома пятой степени

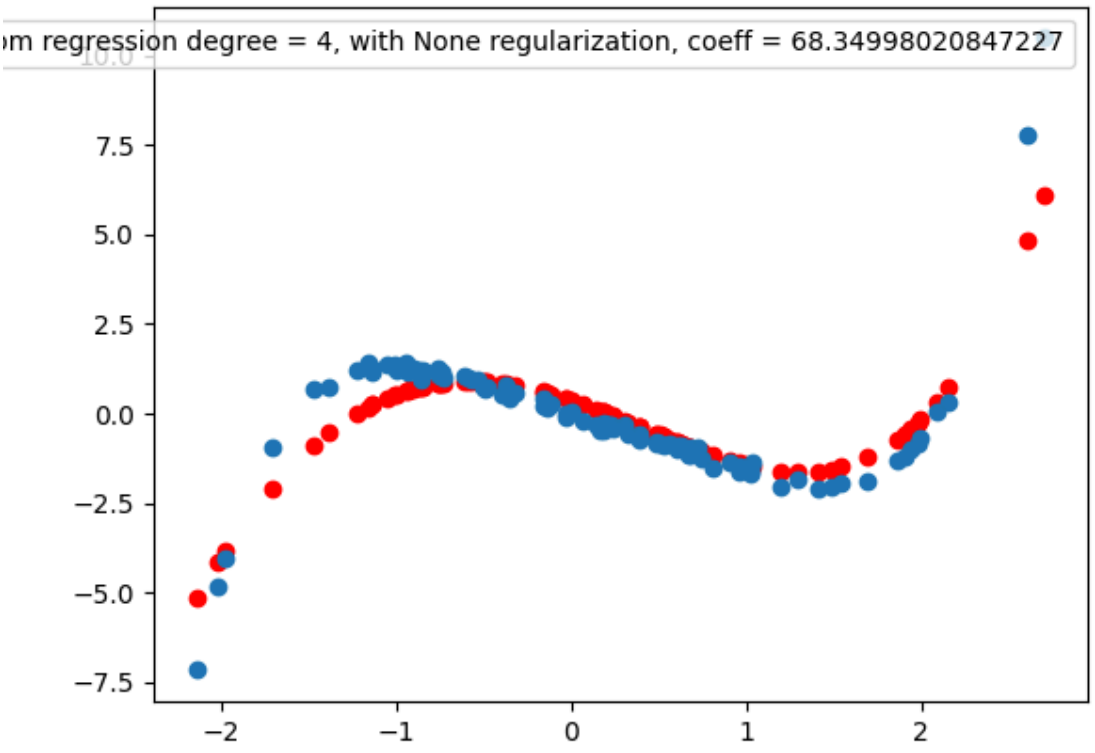
0.1*(x^5 - x^4 - 10x)

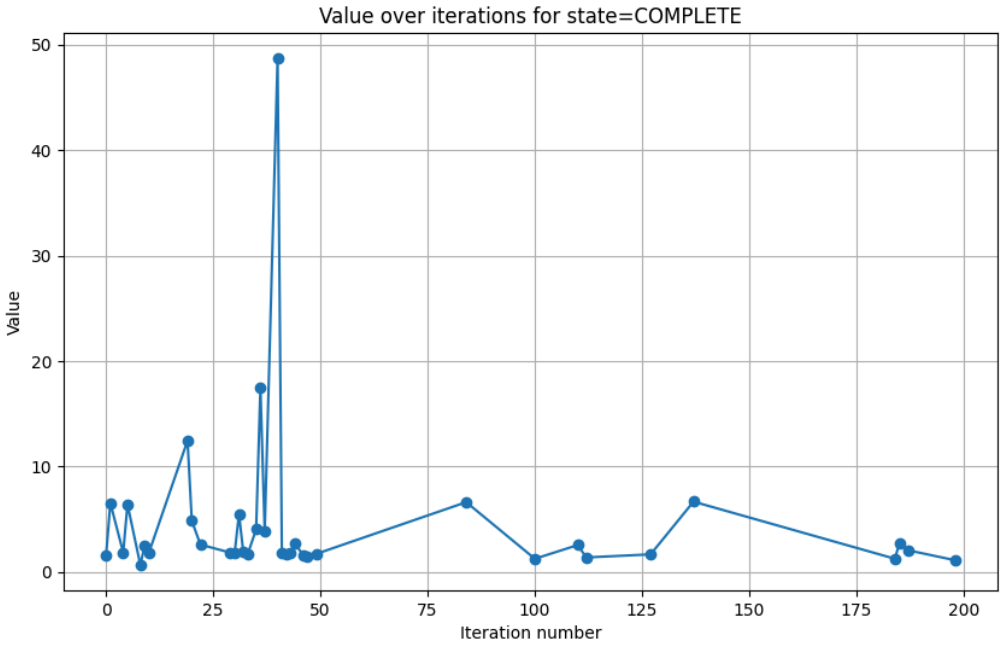
Best hyperparameters: {'learning_rate': 0.0001674610402704859, 'regularization_coeff': 159.9960193500185, 'batch_size': 52, 'power': 6, 'regularization': 'L2'}
The best value for the objective function is: 1.7717095443264408





Best hyperparameters: {'learning_rate': 0.0018112659836807942, 'regularization_coeff': 68.34998020847227, 'batch_size': 27, 'power': 5, 'regularization': 'None'}
The best value for the objective function is: 0.6650448703203959

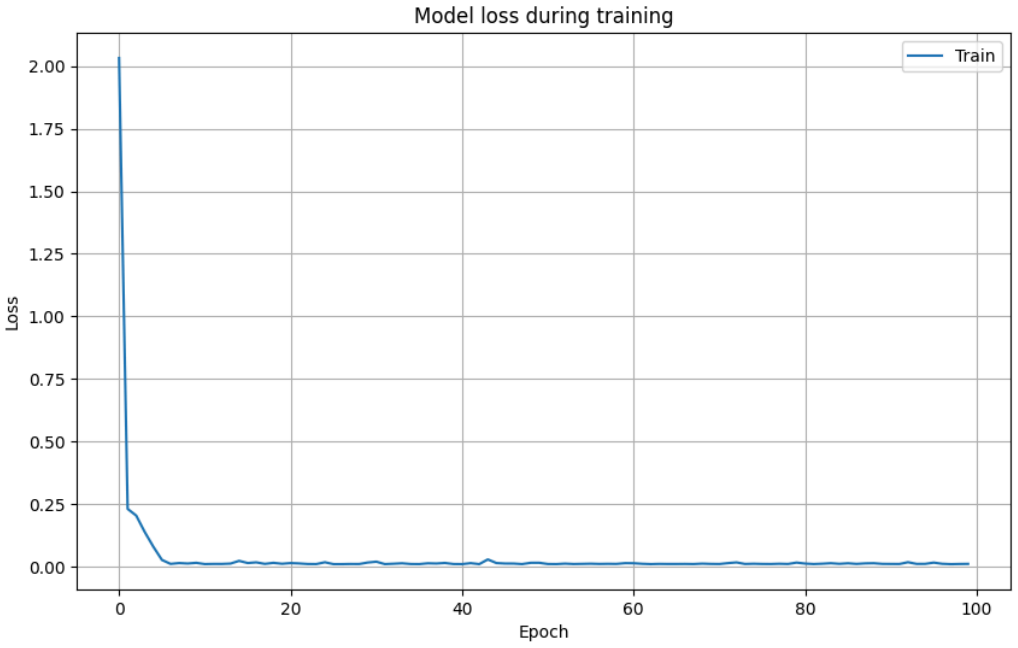




Основная сложность полиномиальной регрессии это подобрать правильную степень полинома и значение коэффициента регрессии. С помощью оптуны это удобно делать.

Так же попробовали применить к SGD with Nesterov, получили

Best parameters: {'learning_rate': 0.09925031616457114, 'momentum': 0.8159628291252279}
Best values: 0.010051322169601917



[Ссылка на гит-репо с кодом]