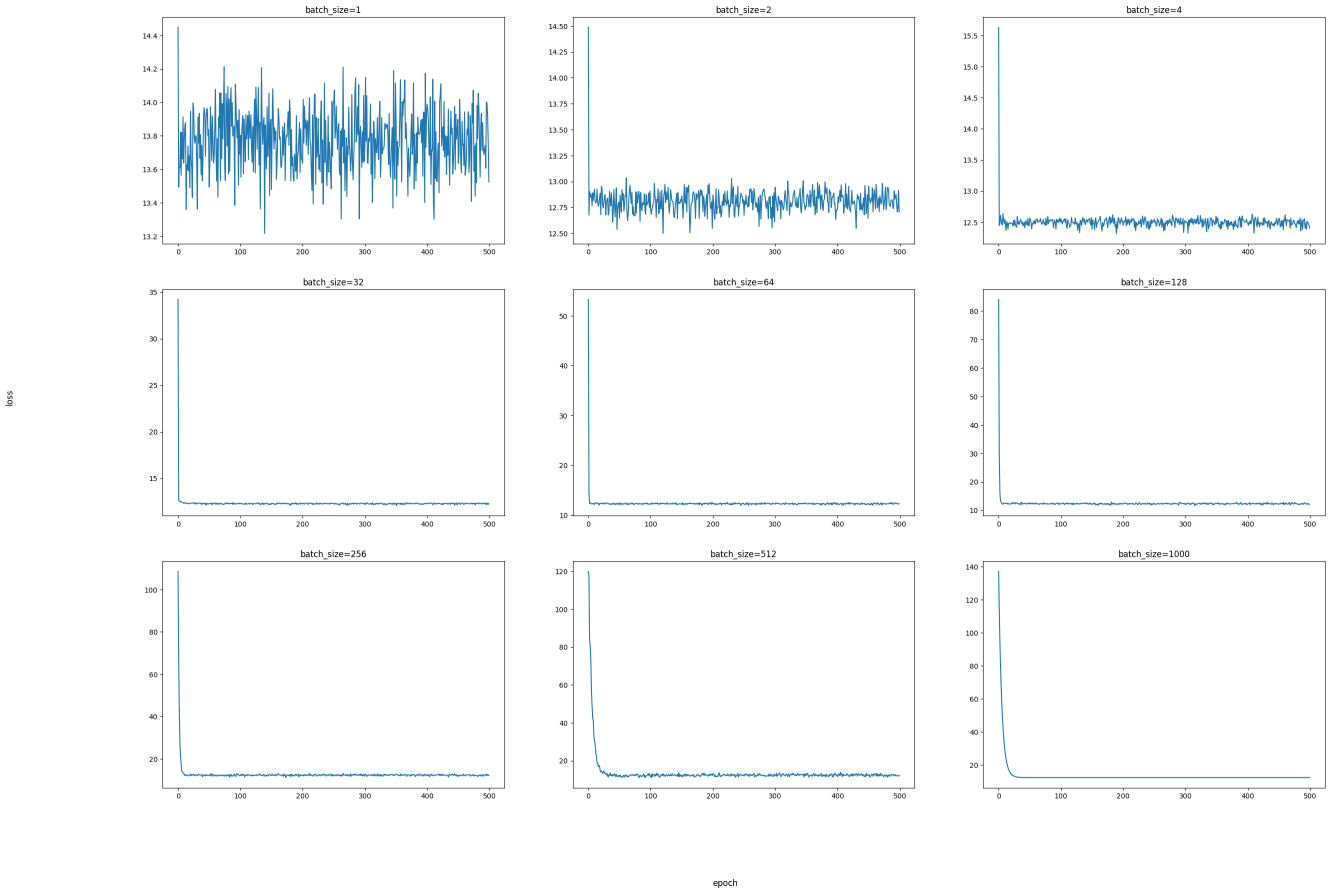
1. Реализуйте стохастический градиентный спуск для решения линейной регрессии. Исследуйте сходимость с разным размером батча (1 - SGD, 2, .., n - 1 - Minibatch GD, n - GD из предыдущей работы).

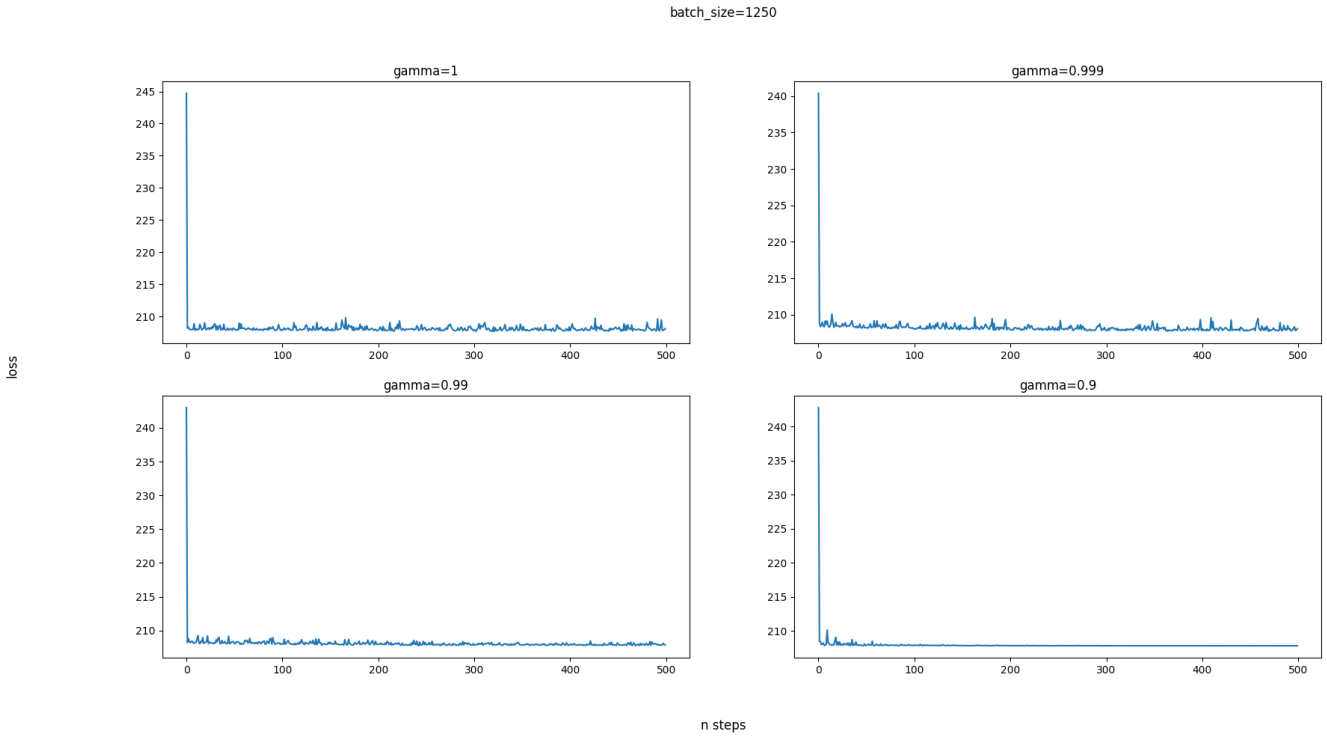
```
model = SGD(data_x, data_y, learning_rate=init_lr / batch_size,
batch_size=batch_size, features=features, N = N)
```

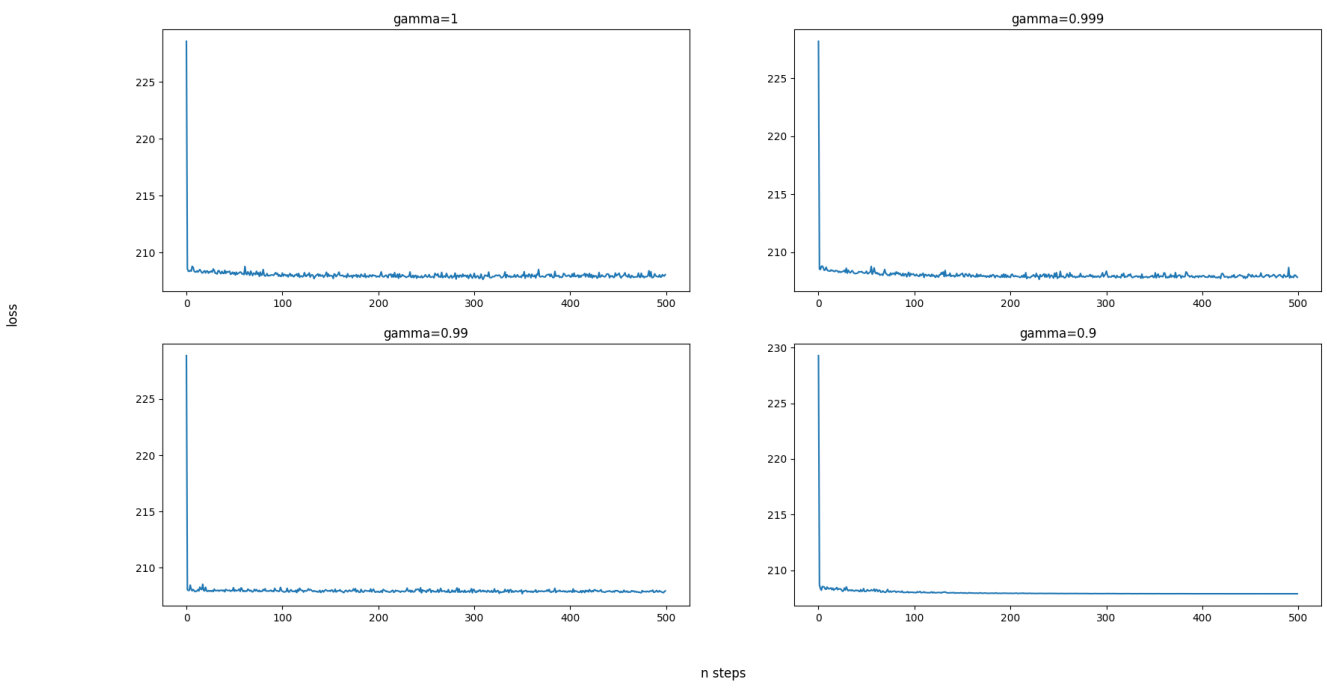


Существует оптимальный диапазон размеров батчей (слишком большие ⇒ долгое обучение, слишком маленькие ⇒ долгое обучение)

2. Подберите функцию изменения шага (learning rate scheduling), чтобы улучшить сходимость, например экспоненциальную или ступенчатую.

N = 5000; features = 1; epoch = 500;
init_lr = 0.001; gammas = [1, 0.999, 0.99, 0.9] (Exponential LR scheduler)





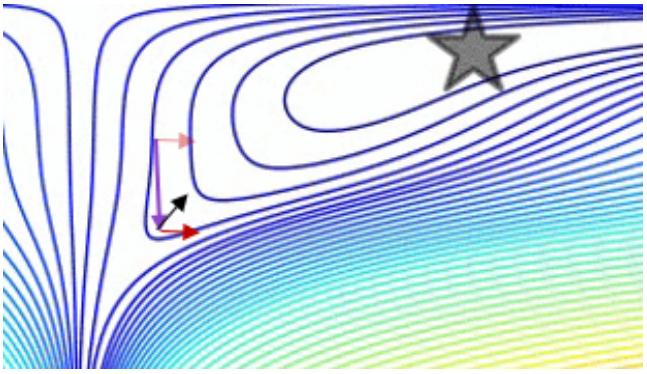
3. Исследуйте модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).

Код написан, следующие пункты – исследование. На практике Nesterov и Momentum отличаются не очень сильно (в случае несложных достаточно выпуклых функций).

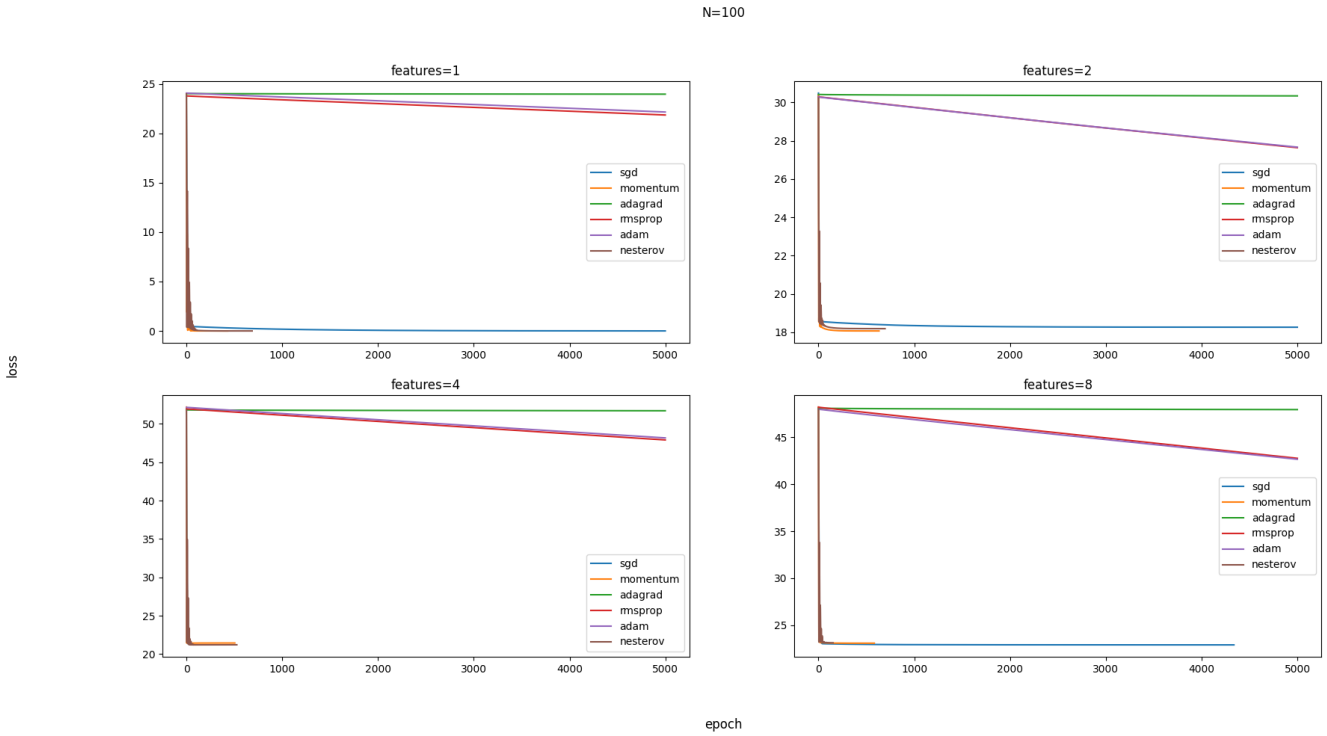
Nesterov придает больший вес члену lr · grad и меньший вес члену v.

В Momentum сначала корректируется v, а затем делается шаг в соответствии с v.

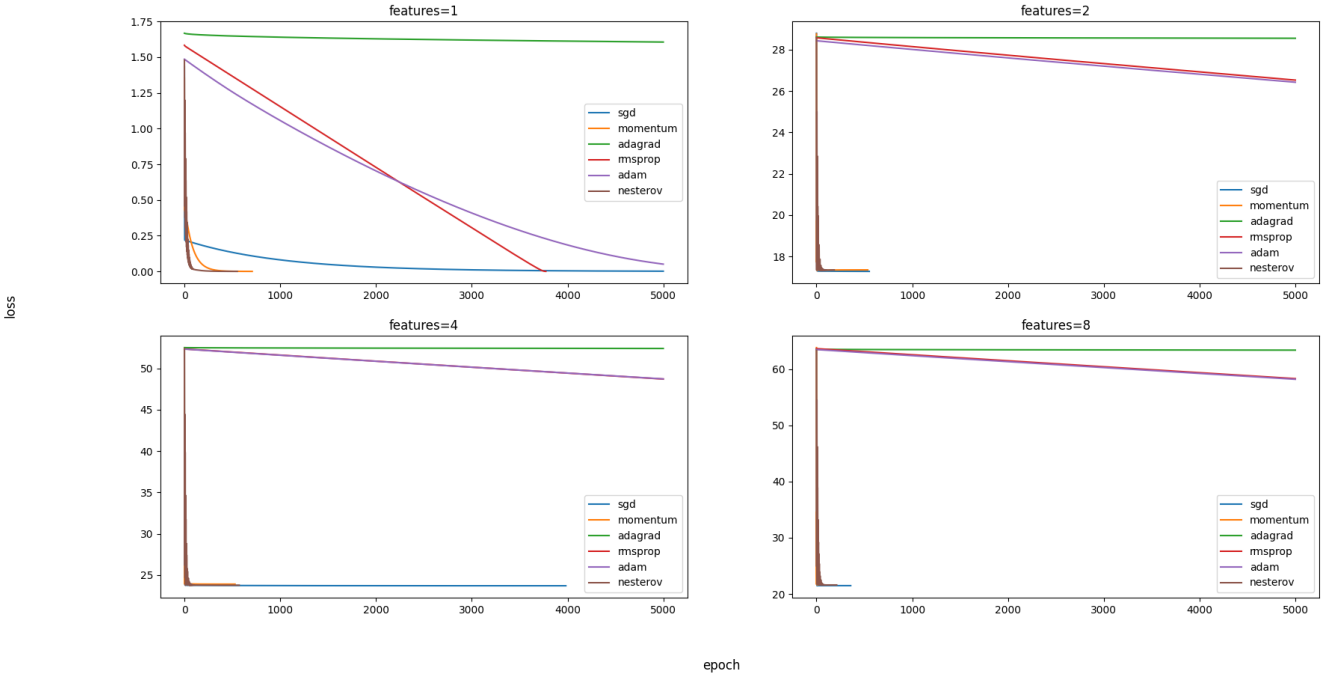
В Nesterov сначала шаг в направлении v, а затем вносится коррекция в вектор v на основе нового места (grad).



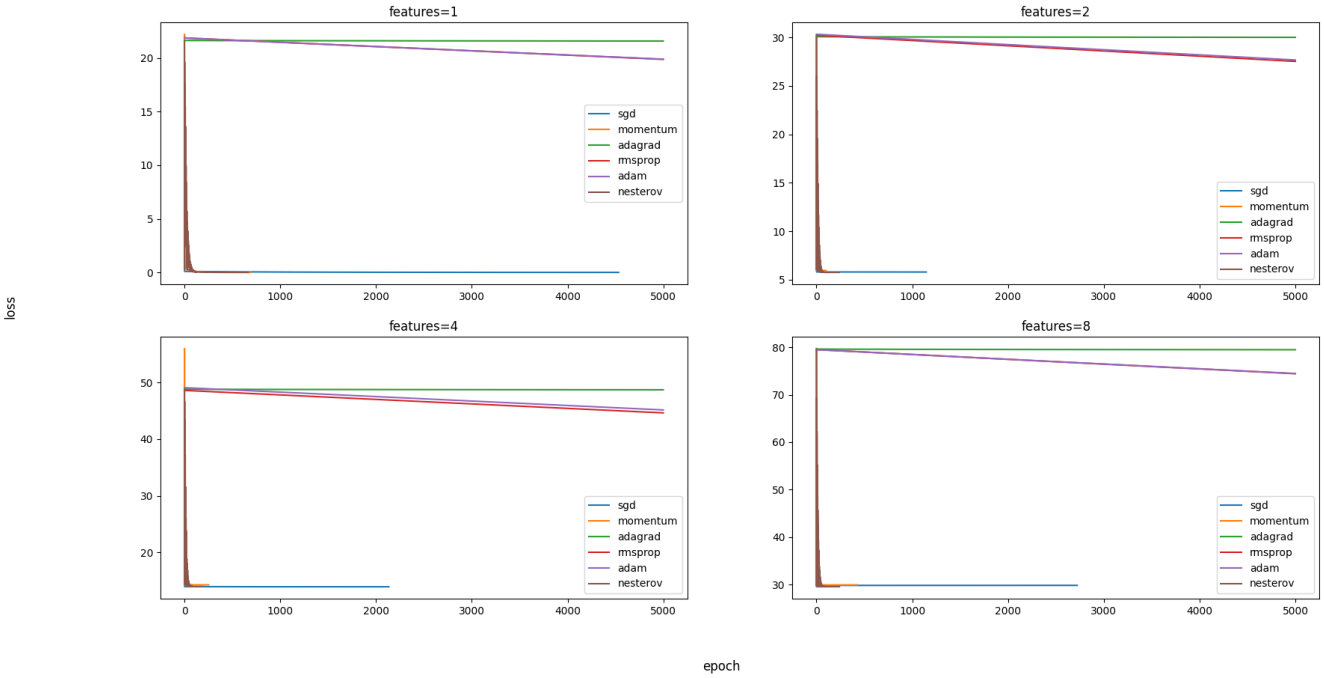
Фиолетовая - impulse.
Розовая – grad(начало фиолетовой).
Черная – grad(конец фиолетовой)
Momentum → красная
Nesterov → черная



N=500


features=1
features=2
features=4
features=8
loss
epoch
sgd
momentum
adagrad
rmsprop
adam
nesterov

N=1000


features=1
features=2
features=4
features=8
loss
epoch
sgd
momentum
adagrad
rmsprop
adam
nesterov

4. Исследуйте сходимость алгоритмов. Сравнить различные методы по скорости сходимости, надежности, требуемым машинным ресурсам (объем оперативной памяти, количеству арифметических операций, времени выполнения)

sgd	67.7 µs ± 892 ns per loop	peak memory: 239.84 MiB	165098
momentum	73.2 µs ± 1.28 µs per loop	peak memory: 239.41 MiB	162693
adagrad	168 ms ± 960 µs per loop	peak memory: 328.23 MiB	410000
rmsprop	31.4 ms ± 58.3 ms per loop	peak memory: 265.96 MiB	120143
adam	72.03 µs ± 1.28 µs per loop	peak memory: 215.28 MiB	61096
nesterov	68.6 µs ± 1.13 µs per loop	peak memory: 239.54 MiB	162668

sgd	1	4	5
momentum	4	2	4
adagrad	6	6	6
rmsprop	5	5	2
adam	3	1	1
nesterov	2	3	3

adam	5
nesterov	8
momentum	10
sgd	10
rmsprop	13
adagrad	18

1) adagrad нужно было ставить больший lr
2) Не зря default optimizer во всех frameworkах - Adam(1e-4)
3) Методика могла хромать + по хорошему не memory, а delta memory (более репрезентативно, т. к. Python мог много съесть без алгоритма)

5. Постройте траекторию спуска различных алгоритмов из одной и той же исходной точки с одинаковой точностью. В отчёте наложить эту траекторию на рисунок с линиями равного уровня заданной функции.

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2$$

Description:

Dimensions: 2

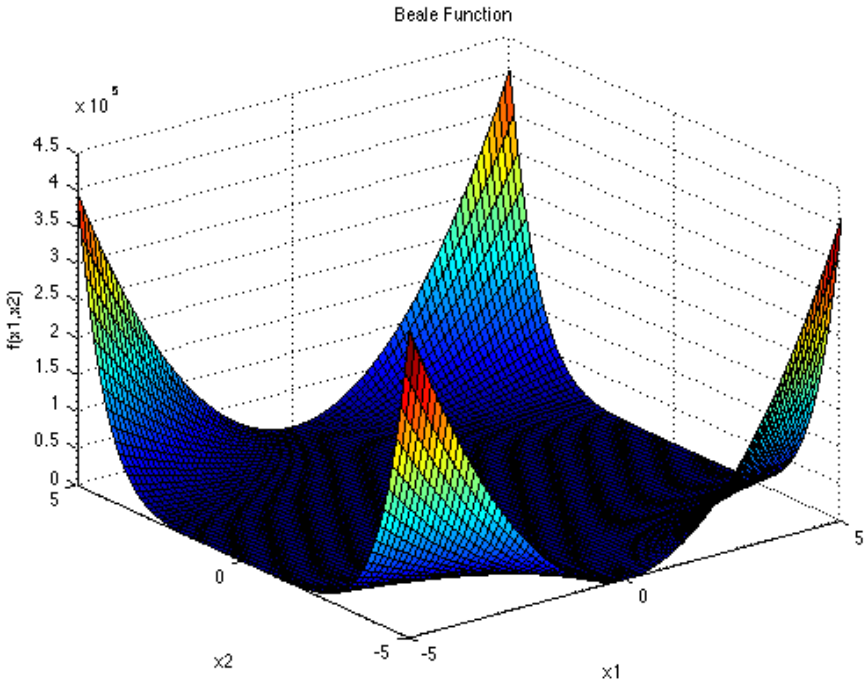
The Beale function is multimodal, with sharp peaks at the corners of the input domain.

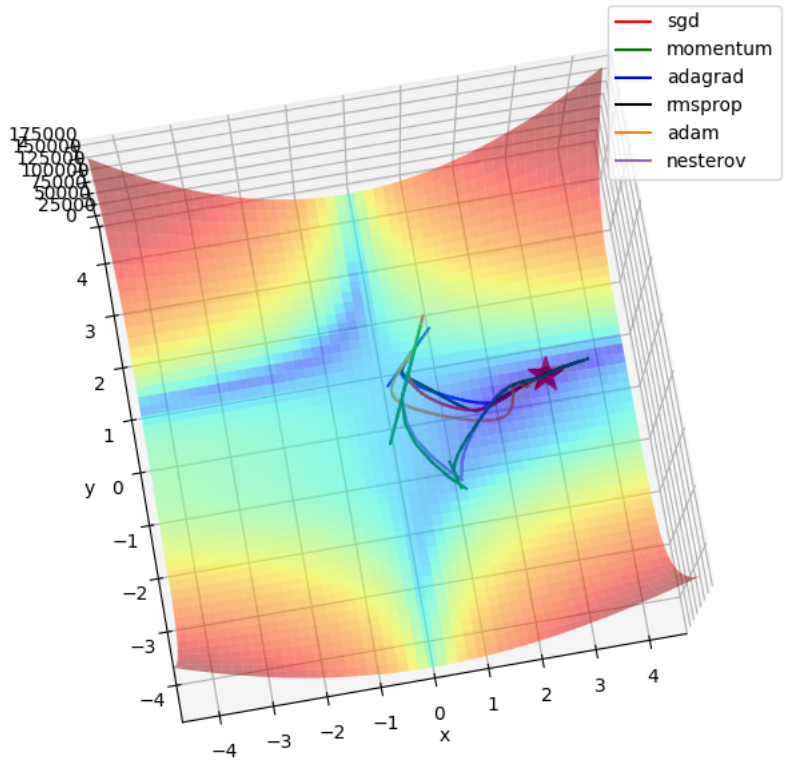
Input Domain:

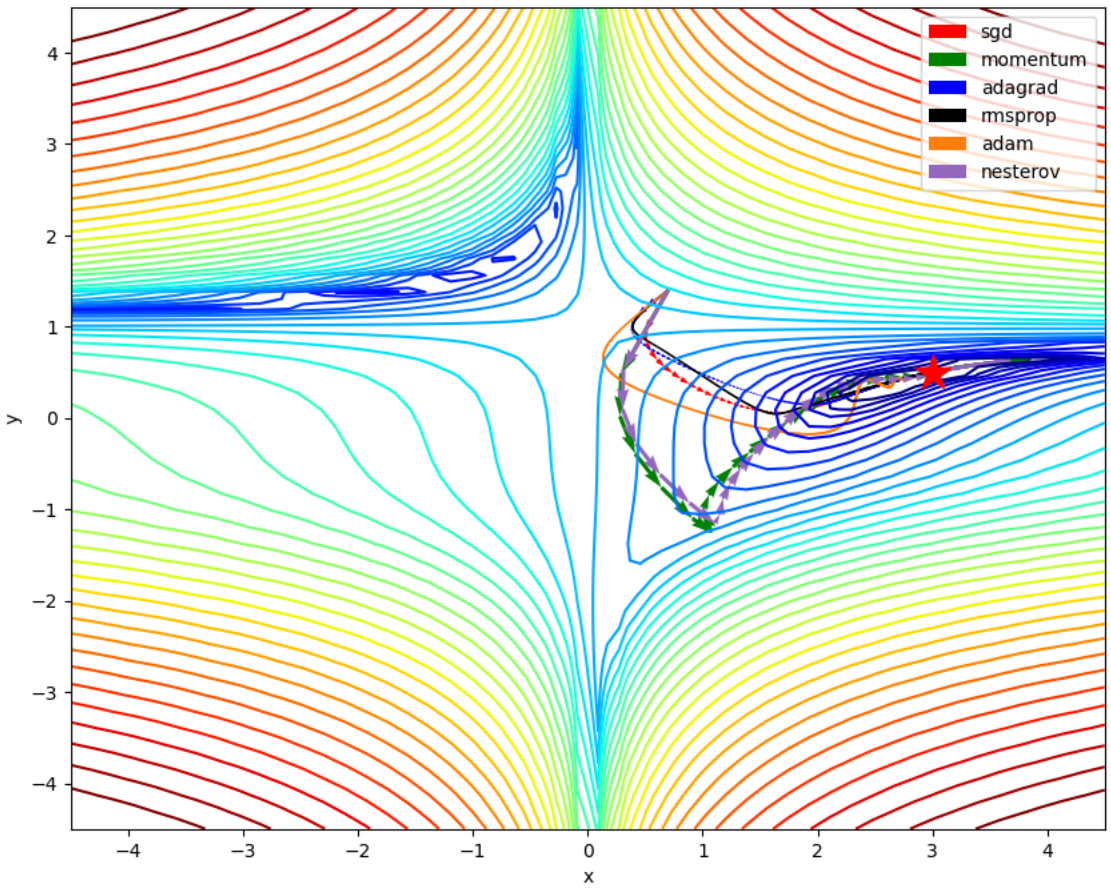
The function is usually evaluated on the square $x_i \in [-4.5, 4.5]$, for all i = 1, 2.

Global Minimum:

$f(\mathbf{x}^*) = 0$, at $\mathbf{x}^* = (3, 0.5)$




sgd
momentum
adagrad
rmsprop
adam
nesterov
175000
x
y


sgd
momentum
adagrad
rmsprop
adam
nesterov
x
y

Доп задание

1. Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.

```
x11 x12   x1m          w11 …  w1k            y11  …  y1k
x21 x22 … x2m
x31 …   …                …      …              …      …
 …
xn1 xn2   xnm          wm1 …  wmk            yn1  …  ynk
```

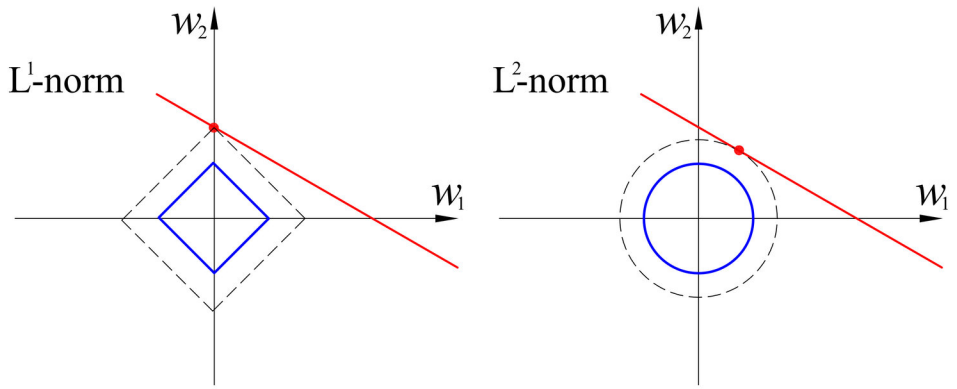
`b0 + x1 * w1 + x2 * w2 + … + xm * wm = y – Linear Regerssion`

`a0 + x^1 * a1 + x^2 * a2 + … + x^m * am = y – Polynomial Regression`

```
x1      1 x1 x1^2 … x1^m          Тогда задача Polynomial ⇒ Linear (останется
x2  ⇒                                    найти a0, a1, … am)
…
xn      1 xn xn^2 … xn^m
```

* Считаем что на вход дана максимальная степень полинома, иначе задача теряет смысл

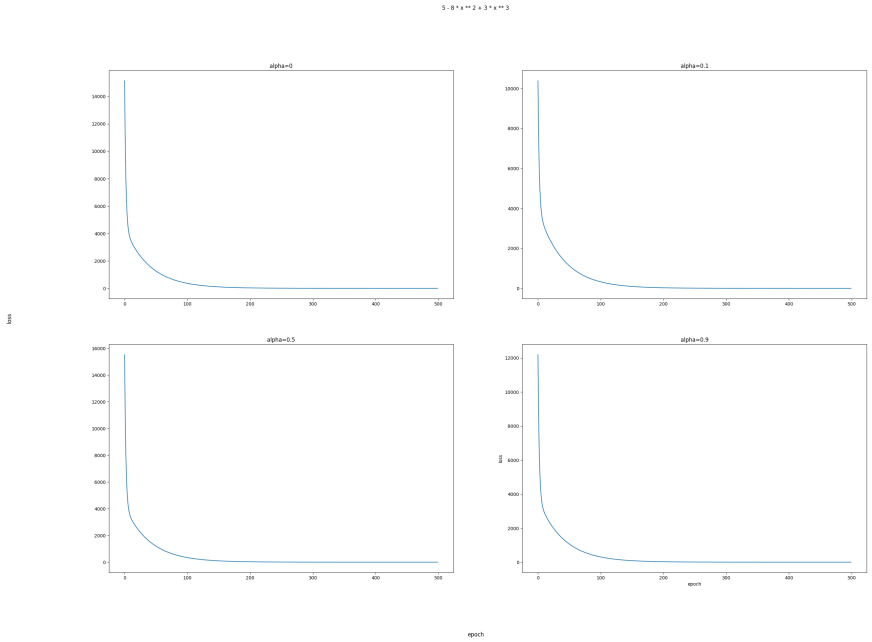
В общем L1 служит для отбора признаков (генерализация), L2 чаще показывает лучшие метрики.



L1 Reg (LASSO)

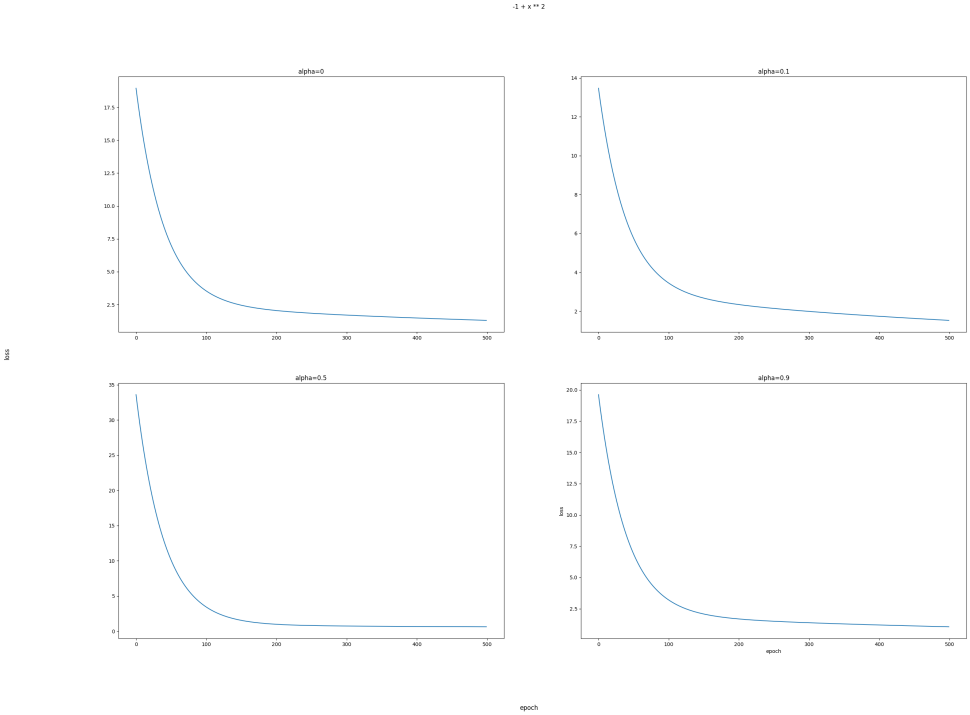
f(x) = 5 - 8 * x ** 2 + 3 * x ** 3

```
[0.52328107 0.65589212 0.81789068 0.12413685]
[0.99115964 0.31595049 0.30165725 0.82379466]
[0.99467257 0.62865374 0.57098384 0.03263719]
[0.46611058 0.55598197 0.12023518 0.45315332]
```



f(x) = -1 + x ** 2

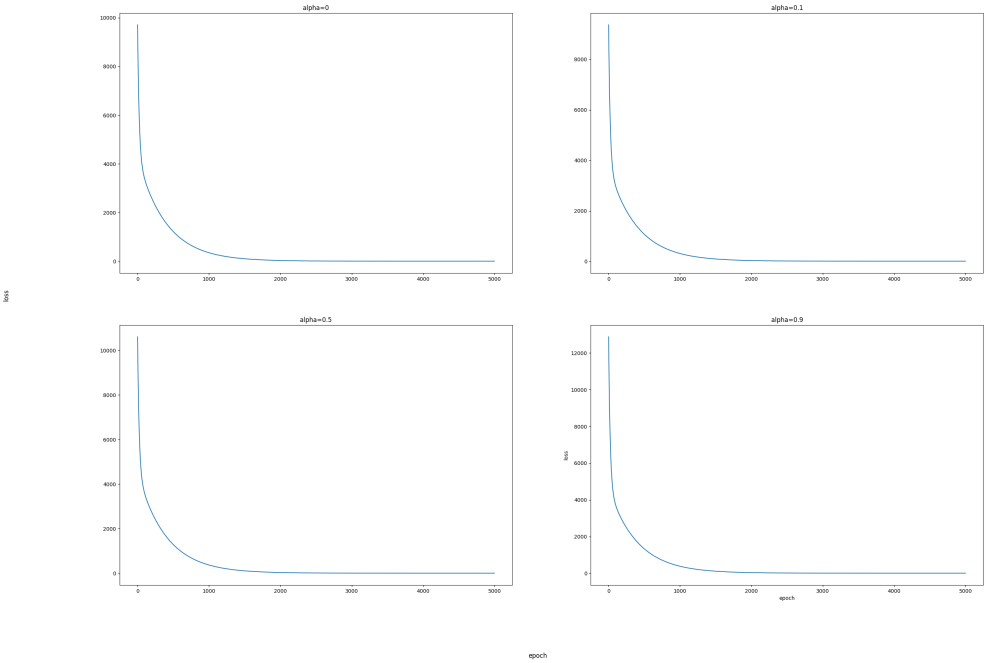
```
[0.20028949 0.7384526 0.42471256]
[0.32125794 0.79280728 0.52030422]
[0.32639975 0.34910702 0.19653969]
[0.09745812 0.66191699 0.4120264 ]
```



L2 Reg (Ridge)

alpha=0 gd.w=array([0.49595069, 0.03659527, 0.59881487, 0.91483353])
alpha=0.1 gd.w=array([0.16565044, 0.35368627, 0.09035679, 0.85197693])
alpha=0.5 gd.w=array([0.54260758, 0.38144387, 0.8255169 , 0.76940793])
alpha=0.9 gd.w=array([0.7079053 , 0.28303268, 0.95202729, 0.39922931])

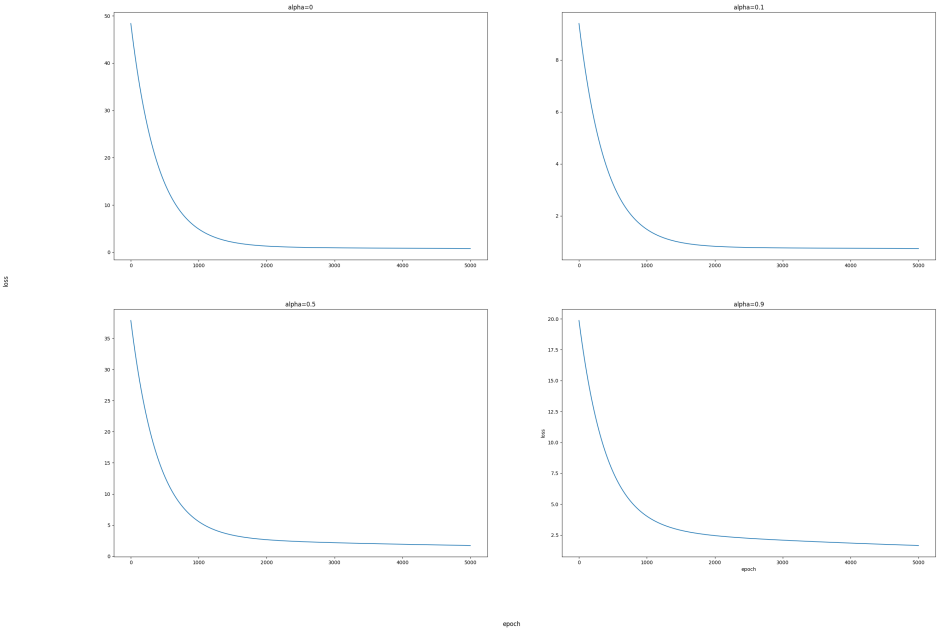
5 - 8 * x ** 2 + 3 * x ** 3



epoch

alpha=0 gd.w=array([0.4704534 , 0.4032398 , 0.03718274])
alpha=0.1 gd.w=array([0.82720197, 0.08418159, 0.50781699])
alpha=0.5 gd.w=array([0.74159742, 0.76564073, 0.14683678])
alpha=0.9 gd.w=array([0.66717868, 0.75694589, 0.37789715])

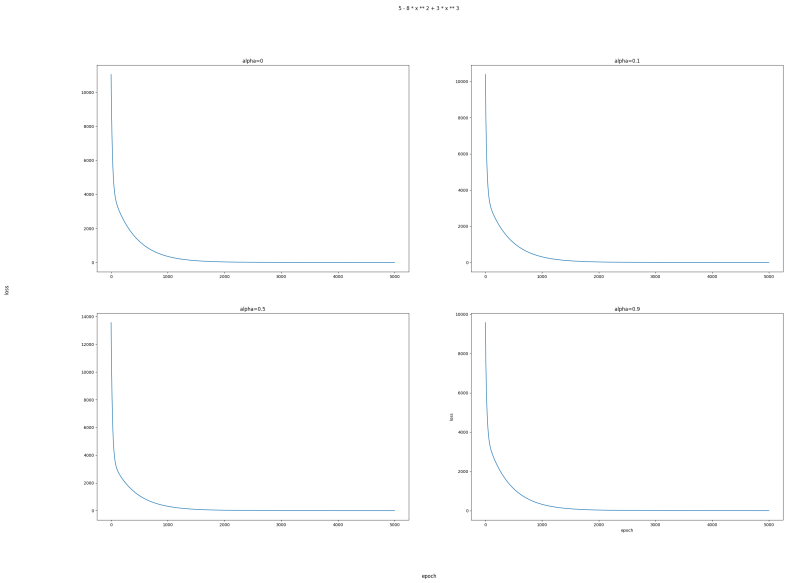
-1 + x ** 2



epoch

Elastic Reg

alpha=0 gd.w=array([0.90915838, 0.33354705, 0.63654371, 0.65456139])
alpha=0.1 gd.w=array([0.68433316, 0.83412455, 0.10336008, 0.63817414])
alpha=0.5 gd.w=array([0.75052565, 0.11435296, 0.09997972, 0.15019641])
alpha=0.9 gd.w=array([0.3291544 , 0.70659818, 0.24174252, 0.82460388])



alpha=0 gd.w=array([0.85238663, 0.87952042, 0.6353575])
alpha=0.1 gd.w=array([0.93359594, 0.59252719, 0.03090941])
alpha=0.5 gd.w=array([0.66763644, 0.67024798, 0.23193773])