

Национальный исследовательский университет ИТМО  
Факультет информационных технологий и программирования  
Прикладная математика и информатика

# Методы оптимизации

Отчет по лабораторной работе №2

⟨Собрано 3 июня 2023 г.⟩

**Работу выполнили:**

Бактурин Савелий Филиппович М32331

Вереня Андрей Тарасович М32331

Сотников Максим Владимирович М32331

**Преподаватель:**

Ким Станислав Евгеньевич

# Стохастический градиентный спуск

## Исследование с разными размерами батча

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).

## Стохастический градиентный спуск

*Стохастический градиентный спуск* — модификация к основному методу итерационного поиска минимума через антиградиент дифференцируемой функции в рассматриваемой плоскости  $\mathbb{R}^n$ . Идея: пусть есть множество  $M$  – какой-то полный набор данных вычисленных оценок при спуске к минимуму, тогда в рассматриваемой версии мы будем брать случайное значение из выбранного  $M' \subset M$ . Как правило, такой подход преуменьшает вычислительные ресурсы, в особенности, следует помнить, что `float` считается крайне медленно, и ускоряет итерацию по количествам эпохам, но при этом мы теряем точность сходимости.

Пусть  $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$  – координата в  $\mathbb{R}^n$  и задана функция  $f(x_i) : \mathbb{R}^n \rightarrow \mathbb{R}$ . Мы хотим нашу задачу свести к исследованию на некоторых специальных образцах заданной функции, для каждой точки из которых мы будем минимизировать ошибку для дальнейшего нахождения приближенного минимума рассматриваемой функции  $f(x_i)$ . Мы хотим найти линейную регрессию, представляющая из себя полином 1-ой степени от  $n$  переменных. Для начала мы найдем функцию ошибки  $S$  по следующей формуле:

$$S(f) = (X'^T \times X')^{-1} \times X'^T \times Y \times \vec{x},$$

где  $Y$  – матрица значений при множестве  $X$  (определение),  $X'$  – это матрица  $X$ , но в 1-ой колонке забытый единицами. По другому мы можем записать данную формулу следующим образом:

$$S(f) = \sum_{i=1}^N (y_i - x_i \cdot w_i)^2,$$

где  $y_i \in D(f(x_i))$  (образ функции),  $w_i \in W$  – сгенерированные веса, коэффициенты при линейной функции.

Рассмотрим идею алгоритма. При итерации, пока мы не превысили максимальное количество шагов или не сведем нашу функцию потери до некоторого  $\varepsilon$ , мы будем обобщать экспериментальные данные в виде случайных точек в некоторую многомерную линию. На каждом шаге мы будем изменять функцию потерь от измененной  $w$ .

Напишем идейный псевдокод. Скажем, что  $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$  – координата в  $n$ -мерном пространстве  $\mathbb{R}^n$ ,  $y_i = \{y_i^0, y_i^1, \dots, y_i^{n-1}\}$  – образ функции  $f(x_i)$ .

```
1 function S(x, y, w):
2     return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4 function stochastic_descent(x, y):
5     w  $\leftarrow$  [ $w_i \in \mathbb{R}$ ] * n
```

```

6   prev ← INIT, предыдущее значение функции потери
7   next ← S(x, y, w), текущее значение функции потери
8   α ← const
9   while |prev - next| > ε:
10      prev ← next
11      i ← x ∈ [0, |Y|]
12      T ← [0] * n
13      ∀j ∈ |w| do
14          Tj ← (yi - xi × w) · xij
15      w ← w + α · T
16      next ← S(x, y, w)
17   return w
18

```

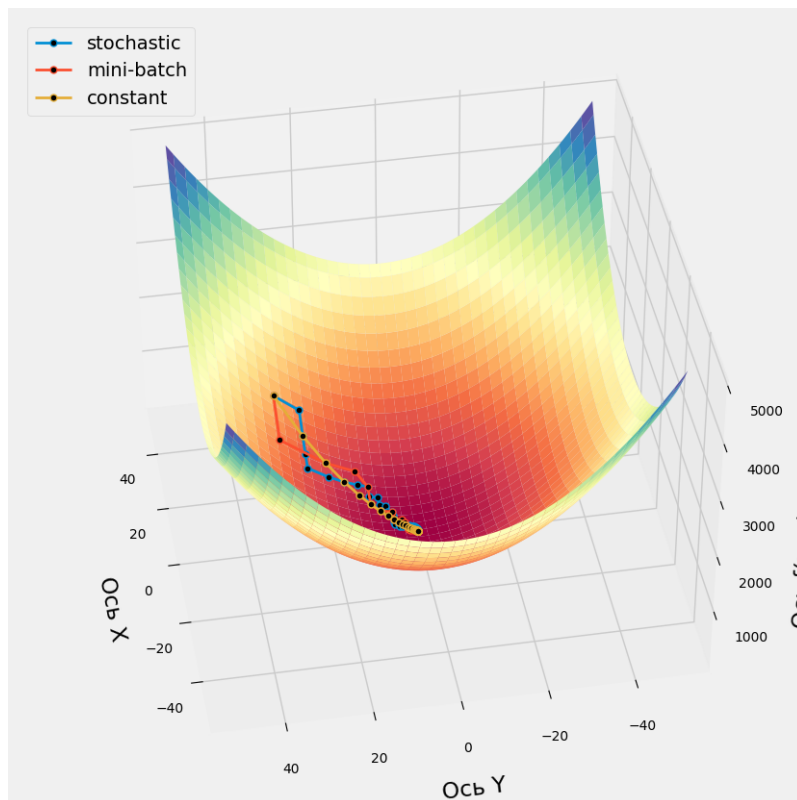
**Пример с SGD, Minibatch, GD.** Для понимания контекста, давайте увидим разницу между рассматриваемым градиентным спуском, Minibatch и общим, с которым мы уже давно знакомы. Возьмем в качестве функции  $f(x, y) = x^2 + y^2$ , тогда его градиентом будет  $\frac{\partial f}{\partial \vec{x}} = 2x + 2y$ , также для GD мы выставим следующие параметры:

▷ learning\_rate = 0.1

▷  $\varepsilon = 0.00001$

▷ max\_epoch = 1000 – одно из условий остановки спуска, вторым – проверка, что мы сошлись.

Запустим и посмотрим на график всех трех джентльменов.



SGD vs. Minibatch vs. GD

Полученные данные, после прохода алгоритмов, выглядят так:

- ▷ Для стохастического нашлась  $f(0.002658, 0.00836) \approx 0.000008$ .
- ▷ Для Minibatch –  $f(0.002658, 0.000219) \approx 0.000007$ .
- ▷ Для градиентного спуска (constant) –  $f(0.001701, 0.002042) \approx 0.000007$ .

Точками отмечены шаги алгоритмов. Заметим, что в отличие GD, стохастический спуск уже с первого шага начинает идти не туда и, как можно приглядеться в минимуме функции, [синяя](#) полоса пробирается то левее, то правее GD небольшими шагами. Minibatch также начинает свое [движение](#) не совсем по траектории GD, но в отличие от своего предшественника, быстрее начинает сходиться к настоящей траектории.

## Minibatch градиентный спуск

Модификация *Minibatch* обобщает вариант стохастического градиентного спуска, тем, что во время итерации мы будем брать не одну случайную точку из посчитанных на предыдущем шаге и изменять функцию потерь как бы относительно её, а теперь возьмем выборку  $M' \subset M$ , причем, обязательно, чтобы  $|M'| > 1$  и  $|M'| < |M|$ .

Тогда псевдокод от предыдущего рассмотренного варианта почти ничем не отличается. Скажем также, что  $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$  – координата в  $n$ -мерном пространстве  $\mathbb{R}^n$ ,  $y_i = \{y_i^0, y_i^1, \dots, y_i^{n-1}\}$  – образ функции  $f(x_i)$ ; значение  $m$  – выступает в роли мощности подмножества  $M'$ .

```
1 function S(x, y, w):
2     return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4 function stochastic_descent(x, y, m):
5     w  $\leftarrow$  [ $w_i \in \mathbb{R}$ ] * n
6     prev  $\leftarrow$  INIT, предыдущее значение функции потерь
7     next  $\leftarrow$  S(x, y, w), текущее значение функции потерь
8      $\alpha \leftarrow$  const
9     while |prev - next| >  $\varepsilon$ :
10         prev  $\leftarrow$  next
11          $\forall i \in [0, m]$  do
12             T  $\leftarrow$  [0] * n
13              $\forall j \in |w|$  do
14                  $T_j \leftarrow (y_i - x_i \times w) \cdot x_i^j$ 
15             w  $\leftarrow$  w +  $\alpha \cdot T$ 
16         next  $\leftarrow$  S(x, y, w)
17     return w
18
```

## Градиентный спуск

Наконец, самый общий случай и являющийся самым быстроходным среди двух рассмотренных ранее, благодаря тому, что мы учитываем все точки  $M' \equiv M$ . Данный

метод работает крайне медленно, но является одним из самых быстрых в сходимости к приближенной точке минимума.

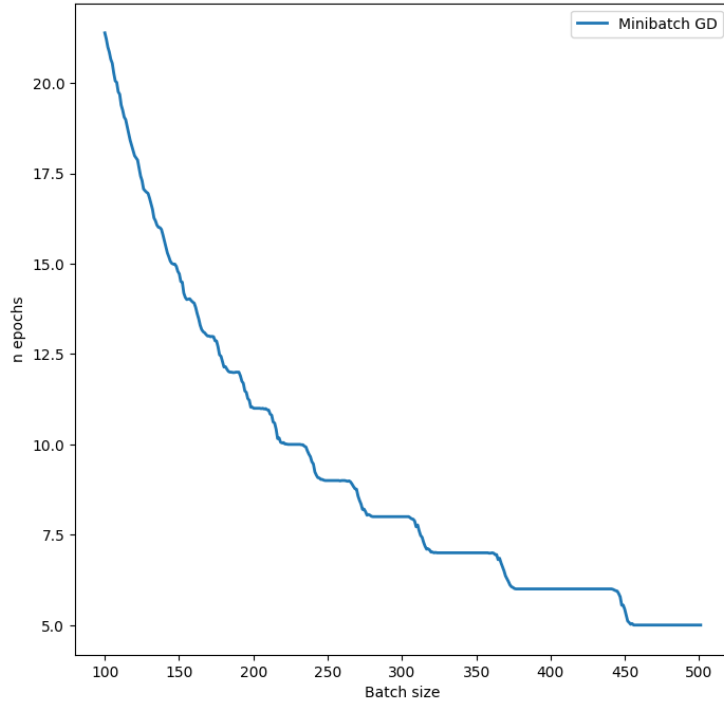
Пусть  $m$  – есть мощность множества  $M$ , тогда идейным псевдокодом-решением задачи будет являться тот же код, что и для предыдущего варианта, то есть

```

1 function S(x, y, w):
2     return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4 function stochastic_descent(x, y, m):
5     w  $\leftarrow$  [wi ∈ ℝ] * n
6     prev  $\leftarrow$  INIT, предыдущее значение функции потери
7     next  $\leftarrow$  S(x, y, w), текущее значение функции потери
8     α  $\leftarrow$  const
9     while |prev - next| > ε:
10         prev  $\leftarrow$  next
11         ∀i ∈ [0, m] do
12             T  $\leftarrow$  [0] * n
13             ∀j ∈ |w| do
14                 Tj  $\leftarrow$  (yi - xi × w) · xij
15             w  $\leftarrow$  w + α · T
16     next  $\leftarrow$  S(x, y, w)
17     return w
18

```

**Исследование Stochastic vs. Minibatch vs. GD.** В качестве рассматриваемого пространства  $\mathbb{R}^2$ , где  $2 = \text{количество\_весов} + \text{«результатирующий\_вес»}$ , для  $X \subset \mathbb{N}$  мы возьмем множество  $\{x\}_{i=1}^n$ , где  $n \leftarrow 500$  и  $x_i < x_{i+1}$ . Возьмем в качестве генерируемой прямой в пространстве одного из наиболее известного представителя фауны функции потерь, называемый *MSE*. Построим график исследования эпохи(batch\_size), где под *эпохами* мы определяем количество шагов до сходимости к значению функции потерь на реальных данных с  $\pm$  точности, порядка, стремящихся к нулю. Построим график.



SGD vs. Minibatch vs. GD

Итак проанализируем. Мы уже знаем, что стохастический градиентный спуск и Minibatch в внезапный момент начинают идти куда-либо, только не по направлению анти-градиента. Замечаем также, особенность в Minibatch: существуют некоторые  $n_i : \exists n'_i : n_i < n'_i, \forall n'' \in [n_i, n'_i] \text{ epochs}(n'') = \text{const}$ . Другими словами: существуют некоторые периоды  $\pi_i$ , при которых количество шагов до сходимости не меняются. При этом, замечаем  $\forall i$  верно, что  $|\pi_i| < |\pi_{i+1}|$ , что, вероятнее всего, связано с размером батча и генерируемых весов точек, заданных уже имплементацией лабораторной работы. Почти полная таблица того, как именно изменяются количества эпох:

Batch size	Кол-во эпох $\times 10^2$
100	2229
101	2202
102	2186
103	2169
104	2136
105	2121
106	2099
107	2085
$\vdots$	$\vdots$
208	1102
209	1102
210	1100
211	1100
212	1100
213	1100
214	1099
215	1099
216	1097
$\vdots$	$\vdots$
468	512
469	501
470	500
471	501
472	500
473	500
$\vdots$	$\vdots$
494	500
495	500
496	500
497	500
498	500
499	500
500	500

## Learning rate scheduling

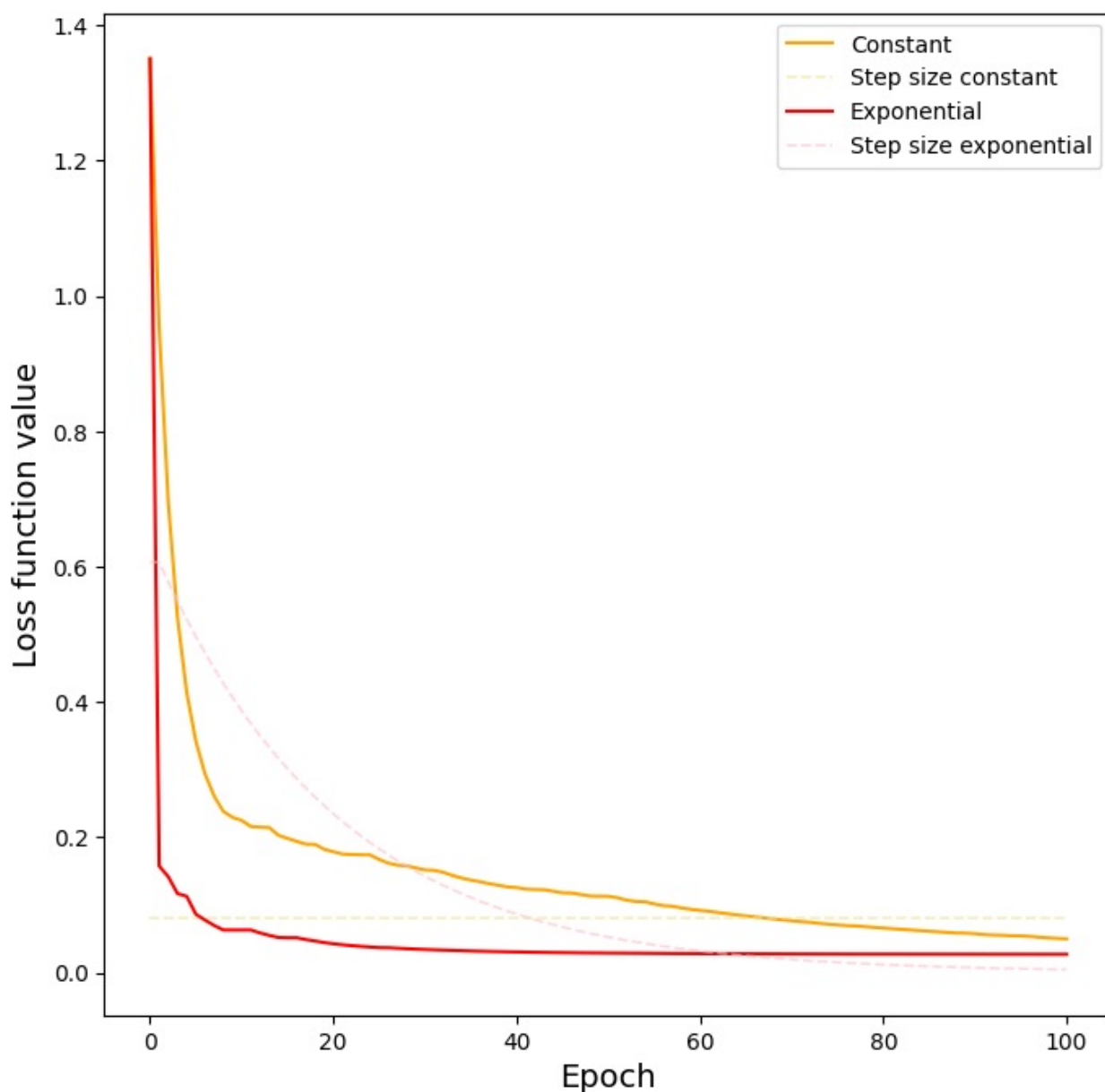
Задачей мы поставим подбор функции изменения шага, чтобы улучшить сходимость из предыдущего пункта. Тогда, мы использовали самый простой способ – *константный*, однако у него есть недостаток: иногда шаг в `const_learning_rate` может «перепрыгнуть» через минимум и/или начать «прыгать» через него бесконечно много раз, в таком случае мы хотим, чтобы шаг был уменьшен. Другой случай: когда такой шаг может привести к долгому ожиданию, пока алгоритм не дойдет до минимума. Вот тут и возникает задача подбора такой функции изменения шага, чтобы алгоритм за какое-то конечное  $k$  эпох добрался до минимума быстрее. В качестве такой мы возьмем *экспоненциальную функцию*. Экспоненциальная функция, в общем случае,

для изменения сходимости выглядит так:

$$\text{learning\_rate} = \text{start\_learning\_rate} \cdot e^{-k \cdot \text{epoch}},$$

где  $\text{learning\_rate}$  – текущий размер шага,  $\text{start\_learning\_rate}$  – стартовая длина,  $k$  – некий параметр, который может зависеть от размера выбранного батча, и, наконец,  $\text{epoch}$  – текущий номер эпохи.

**Пример с  $f(x) = 2x + 0$ .** Для линейной регрессии рассмотрим в качестве функции  $f(x) = 2x + 0$ , где  $a = 2$  и  $b = 0$ . Сравним разницу между константным и экспоненциальной функциями потерь, где в качестве значения  $k \leftarrow 0.05$  и  $\text{epoch}$  будем считать следующим образом:  $(\text{current\_epoch} + 10)$ . Рассмотрим следующую зависимость: значение\_функции\_потерь(количество\_эпох).



Constant lr vs. Exponential lr

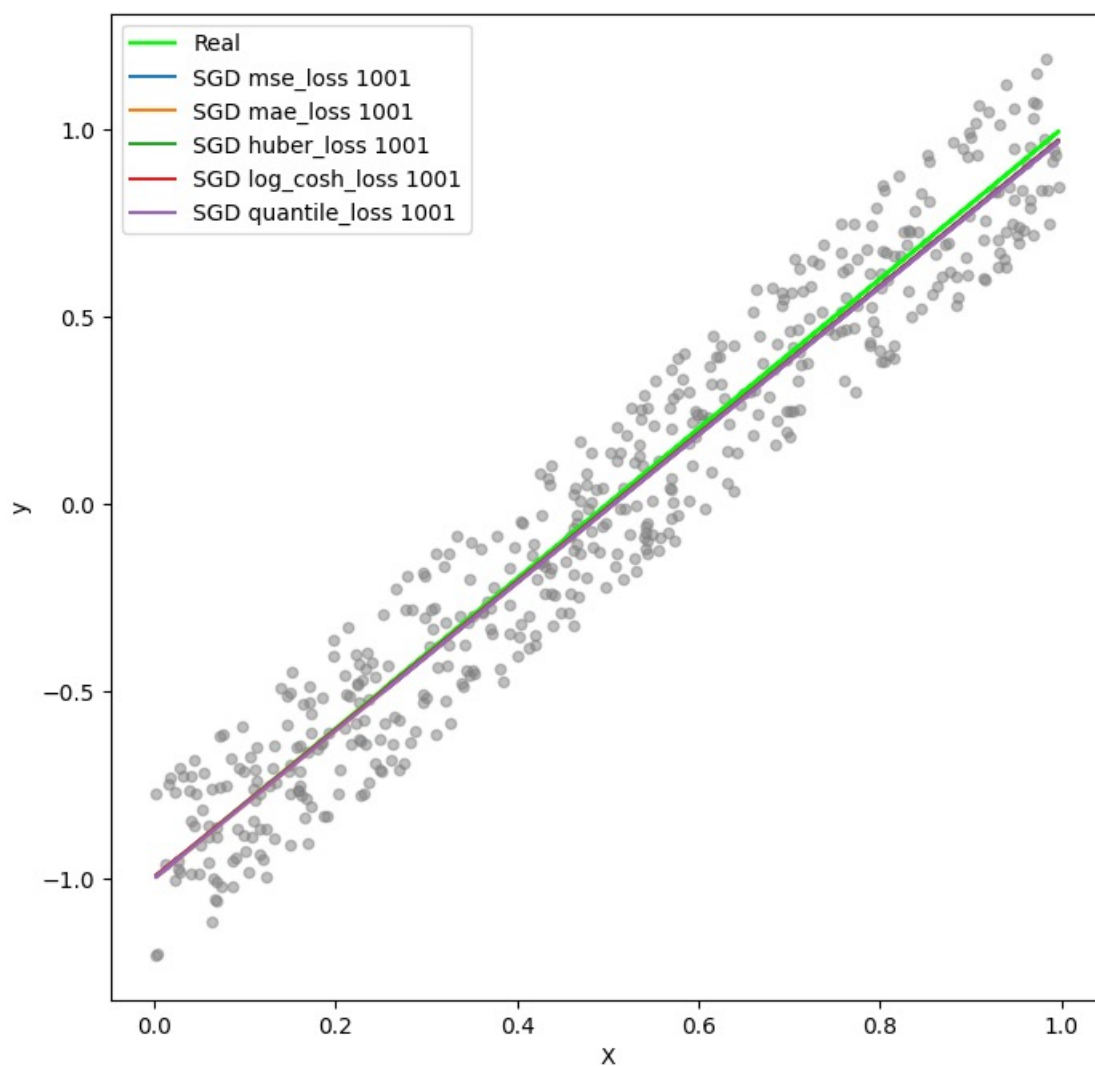


Как можно заметить, константный метод по очевидным причинам хуже сходится (параллельная линия оси ординат), экспоненциальная же понемногу (на самом деле, по многу) начинает быстрее сходиться из-за присутствия «минуса» в степени и, соответственно, меньшим  $\lg$ .

**SGD и разные функции потерь.** Как и в предыдущей задаче, мы рассмотрим линейную регрессию, но в качестве исследования мы возьмем других не менее известных представителей функций потерь:

$$\begin{aligned} \text{MAE} &= \frac{\sum_{i=1}^n |y_i - x_i|}{n} \\ L_\delta(a) &= \begin{cases} \frac{1}{2} \cdot a^2, & |a| \leq \delta, \\ \delta \cdot \left( |a| - \frac{1}{2} \cdot \delta \right) & , \text{ где } a = y - f(x) \end{cases} \\ \text{logcosh} &= \log \left( \frac{e^{y_{\text{pred}} - y_{\text{real}}} + e^{-y_{\text{pred}} + y_{\text{real}}}}{2} \right) \\ \text{quantile} &= \begin{cases} \alpha \cdot (y_{\text{real}} - y_{\text{pred}}), & y_{\text{real}} - y_{\text{pred}} \geq 0 \\ (\alpha - 1) \cdot (y_{\text{real}} - y_{\text{pred}}) \end{cases} \end{aligned}$$

На сей раз, мы рассмотрим линейную функцию  $f(x) = 2x - 1$ . Для каждого из функций потерь построим pred-функцию и сравним их эффективность в плане восстановления до исходной прямой.



Real vs. mae vs. mse vs. huber vs. logcosh vs. quantile

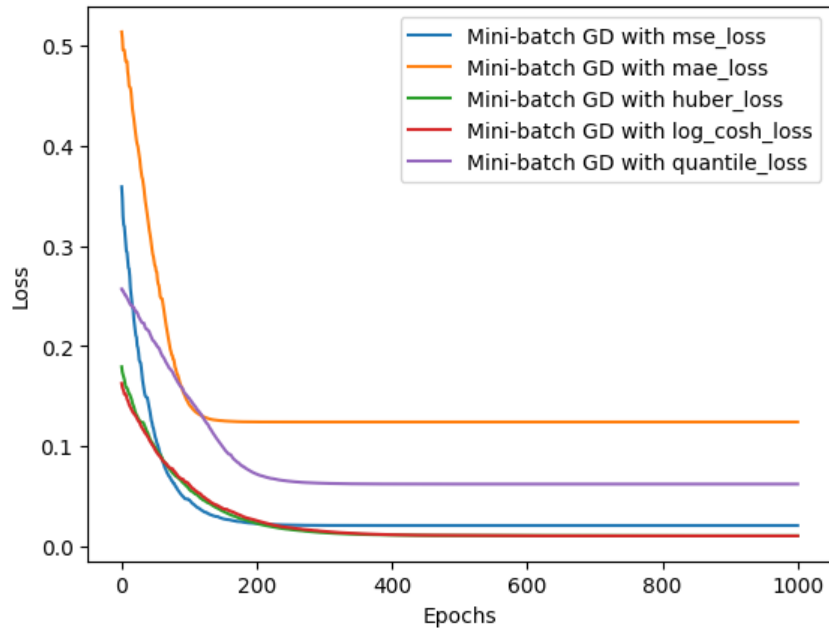
Для нее мы задали следующие настройки алгоритма:

- ▷ Количество генерируемых весов – 5000.
- ▷  $\varepsilon \leftarrow 0.0085$
- ▷ `learning_rate`  $\leftarrow 0.1$
- ▷ Максимальное число до сходимости – 1000.

Заметим, что почти все методы отработали почти **одинаково** и совпали с настоящей прямой. Полные результаты коэффициентов уравнений прямой такие:

$$\begin{aligned} \text{MSE} &\rightarrow y = 2.002 \cdot x - 1.000 \\ \text{MAE} &\rightarrow y = 2.005 \cdot x - 1.002 \\ \text{HUBER} &\rightarrow y = 1.999 \cdot x - 0.998 \\ \text{LOGCOSH} &\rightarrow y = 1.999 \cdot x - 0.998 \\ \text{QUANTILE} &\rightarrow y = 2.006 \cdot x - 1.002 \end{aligned}$$

**Minibatch и разные функции потерь.** Как и в прошлом примере, мы возьмем функцию  $f(x) = 2x - 1$  и те же разные местные представители фауны из предыдущего пункта. Однако, теперь мы возьмем Minibatch GD и составим график на сей раз `значение_функции_потери(эпохи)`. Настройки для алгоритма такие же, как и в прошлом пункте. Запустим и проанализируем:



Minibatch vs. loss functions

И мы получаем две категории победы у разных функций:

- ▷ По скорости (то есть, по количеству эпох) сходимости (то есть, когда мы получаем параллельную прямую над осью ординат) – MAE, при этом эта функция проигрывает следующей категории.
- ▷ По аргументу (то есть, по значению функции потерь) – получаем вровень идущих  $L_\delta(a)$  и  $\text{logcosh}$ , приближающееся к нулю.

Полные результаты значений функций потерь:

$$\begin{aligned}
\text{MSE} &\rightarrow \begin{cases} \text{REAL} = 0.02061322422787911 \\ \text{MSE} = 0.020611644279003895 \\ \text{DIFF} = -1.579948875216064e - 06 \end{cases} \\
\text{MAE} &\rightarrow \begin{cases} \text{REAL} = 0.12404634033734865 \\ \text{MAE} = 0.12403644707794269 \\ \text{DIFF} = -9.89325940596586e - 06 \end{cases} \\
\text{HUBER} &\rightarrow \begin{cases} \text{REAL} = 0.010306612113939555 \\ \text{HUBER} = 0.010306151224572697 \\ \text{DIFF} = -4.6088936685867443e - 07 \end{cases} \\
\text{LOGCOSH} &\rightarrow \begin{cases} \text{REAL} = 0.010242649302131148 \\ \text{LOGCOSH} = 0.01024226833627978 \\ \text{DIFF} = -3.809658513671127e - 07 \end{cases} \\
\text{QUANTILE} &\rightarrow \begin{cases} \text{REAL} = 0.06202317016867433 \\ \text{QUANTILE} = 0.062018214022529945 \\ \text{DIFF} = -4.956146144381723e - 06 \end{cases}
\end{aligned}$$

## Исследование различных модификаций

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

### Momentum

Пусть нам дана функция  $f(x_i)$ , где  $x_i = \{x_i^1, x_i^2, \dots, x_i^n\}$  – координата точки  $x_i$  в  $n$ -мерном пространстве – и пусть у данной  $f$  есть множество локальных точек минимума, образующийся «впадиной» функции, и «горы» – максимумов. Рассмотрим некоторый объект  $\mathfrak{D}$ , который обладает свойством «скольжения» по поверхности нашей функции  $f$ , определяемый следующим образом:

- если он «сходит» с горы, то он не останавливается при «схождении» с локального максимума;
- если он «перескакивает» локальный минимум, то он либо остановится на идеально гладкой части поверхности функции  $f$ , либо он «сойдет» к локальному минимуму.

Идея модификации градиентного спуска *Momentum* в создании алгоритма, обладающий свойством  $\mathfrak{D}$  в имени свойства импульса.

Рассмотрим последовательность  $\{u_i\}$ , которую мы назовем «скоростью», скажем, что  $\{g_i\} \equiv \{-\nabla f(x_i)\}$ , тогда определим скорость следующим образом:

$$v_{i+1} = \lambda \cdot v_i + (1 - \lambda) \cdot g_i, \quad \lambda \in (0, 1)$$

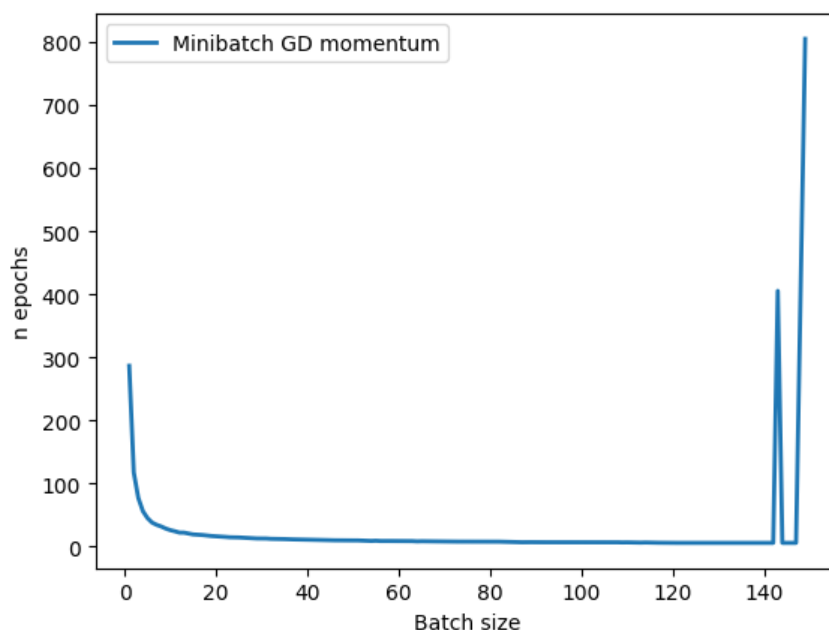
и зададим изменение на  $i + 1$ -ой итерации наших весов точек:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1}, \quad \alpha = \text{const}$$

Здесь и во всех ниже рассматриваемых модификациях стохастического градиентного спуска мы будем рассматривать в качестве функции потерь функцию  $MSE$ . Настроим наше исследование и запустим с следующими параметрами:

- ▷ Максимальное количество эпох – 4000.
- ▷ Начальным `learning_rate` – 0.001.
- ▷ Количество генерируемых точек – 500.

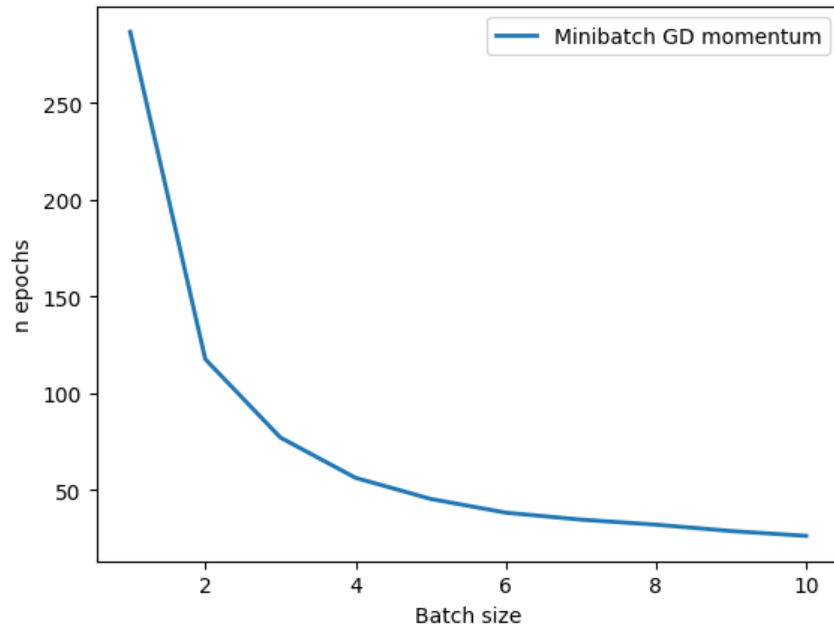
Важное замечание: здесь и далее мы будем ограничивать по количеству шагов алгоритма. Если оно превысило число генерируемых точек, то алгоритм останавливается и передается новое значение батча. Рассмотрим же график для *Momentum*.



Momentum in general

Заметим, что изначально, как и ожидается, алгоритм быстро сходится при больших числах размерах батча. Однако, в какой-то момент, число шагов резко возрастает и, можно заметить, что в среднем зависимость будет перепадать на сам батч, чем на веса. Это происходит из-за многих проблем: одна из которых заключается в том, что наши шаги становятся слишком маленькими настолько, что максимальное число эпох не покрывает в пределе.

Более детальное изображение Momentum при первых десяти batch размерах:



Momentum in  $[1, 10]$

## Nesterov

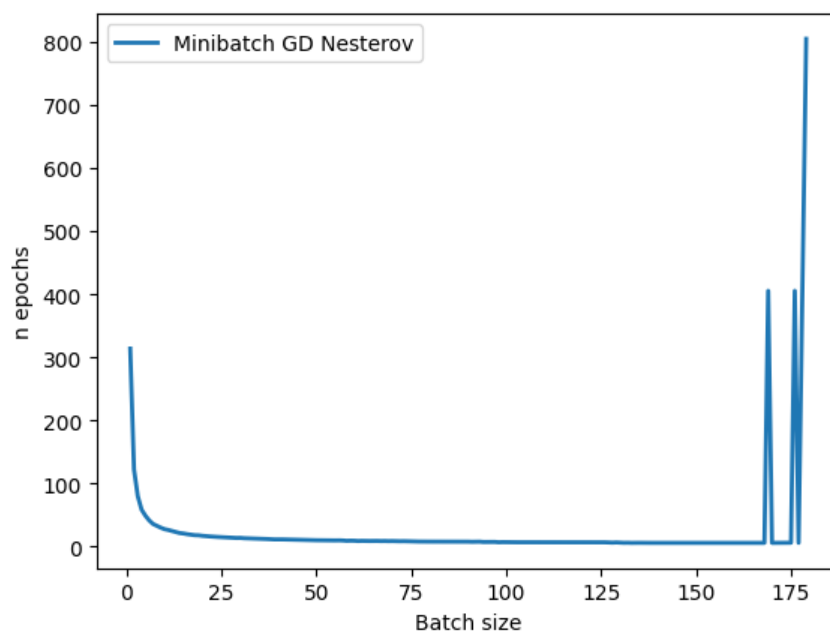
Модификация *Nesterov* по своей сути почти ничем не отличается ранее рассмотренного Momentum: единственное, что отделяет братьев по крови, так это то, что Nesterov считает градиент не в текущей точке, на которой мы стоим, а той, куда мы бы могли пойти, следуя импульсу. Тогда, наши основные формулы немного меняются, для скорости:

$$v_{i+1} = \lambda \cdot v_i + (1 - \lambda) \cdot g(w_i - \alpha \cdot \lambda \cdot v_i), \quad \alpha = \text{const}$$

И для весов:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1}$$

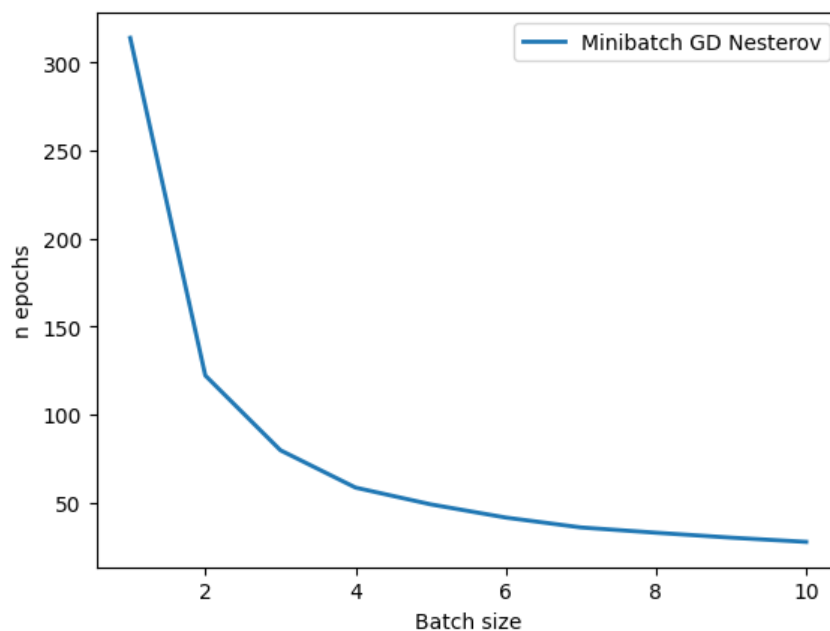
Для исследования алгоритма мы выставим ровно те же настройки, что были у Momentum. Рассмотрим же график для Nesterov.



Nesterov in general

Замечаем, что количество шагов не сильно (иначе говоря, не хуже), чем в предыдущем рассмотренном пациенте. Однако, проблема с шагами остается, чему является подтверждение на последних итерациях batch размеров.

Более детальное изображение Nesterov при первых десяти batch размерах:



Nesterov in  $[1, 10]$

## AdaGrad

Рассмотрим некий `start_learning_rate` и стандартный метод стохастического спуска. Как мы уже видели, одним из недостатков такого метода является появления слишком больших или, наоборот, маленьких компонент относительно друг

друга. Для исправления мы введём матрицу  $\Omega$ , которую определим следующим образом:

$$\Omega = \sum_{j=1}^i g_j \times g_j^T,$$

где  $g_j = \nabla f(x_j)$ . Теперь рассмотрим основную формулу и поделим часть, где идет «шаг» алгоритма, на  $\Omega_{i,i}$ :

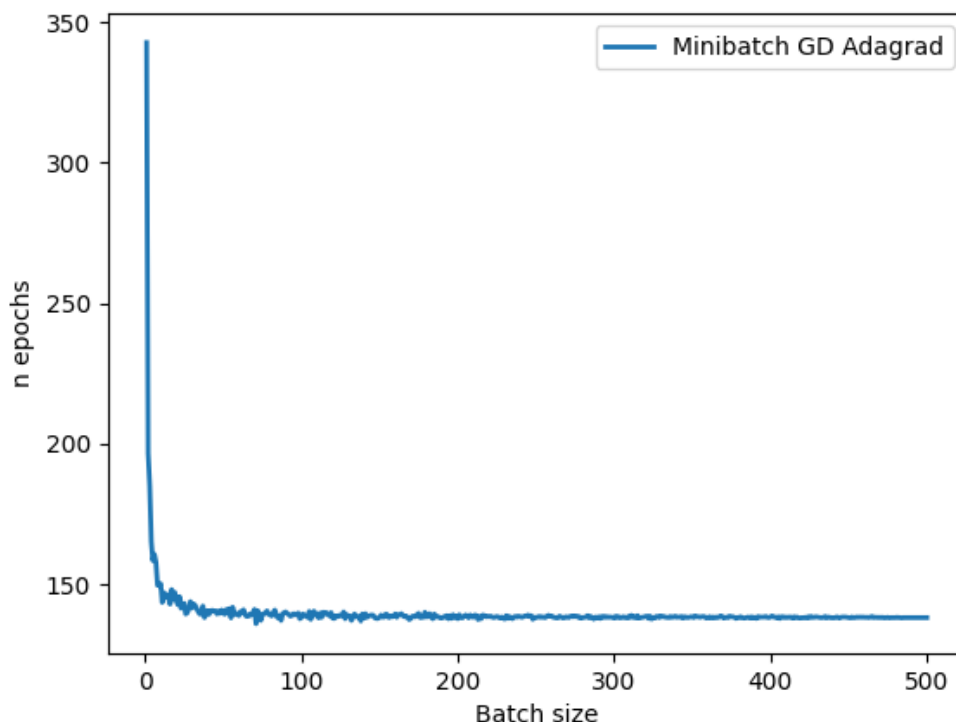
$$w_{i+1} = w_i - \frac{\alpha \cdot g_j}{\sqrt{\Omega_{i,i}}}$$

Теперь же, там, где мы потенциально делаем большие шаги, мы делим на большое  $\Omega_{i,i}$  и таким образом уменьшаем количество больших скачков. Тут мы получаем глобальную проблему в виде потенциально слишком высокой скорости уменьшения шага.

Настроим наше исследование и запустим с следующими параметрами:

- ▷ Максимальное количество эпох – 4000.
- ▷ Начальным `learning_rate` – 0.1.
- ▷ Количество генерируемых точек – 500.
- ▷ В качестве константы положим  $\alpha = 1e - 5$ .

Рассмотрим же график для AdaGrad.



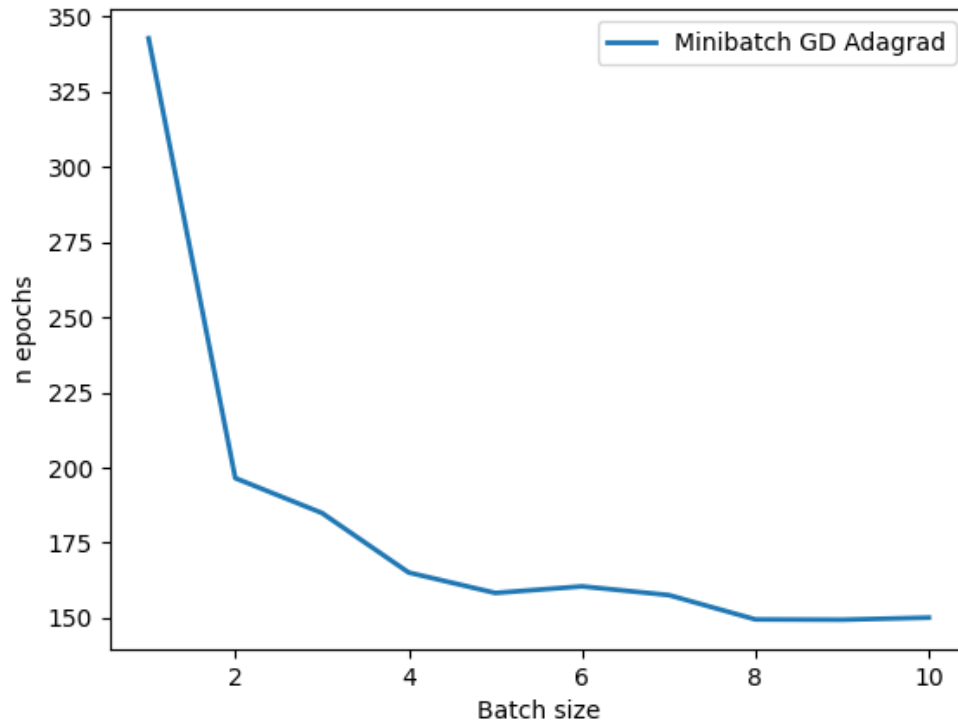
AdaGrad in general

Количество шагов, из-за все ещё нерешенной глобальной проблемы, стало в среднем немного хуже, чем в предыдущих модификациях, однако здесь мы не видим



каких-либо резких возвышений и перепадов. Также заметим, что график функции выглядит не как линия, а как некий «шум», резко перебегающей функцией то вверх, то вниз.

Более детальное изображение AdaGrad при первых десяти batch размерах:



RMSProp in [1, 10]

## RMSProp

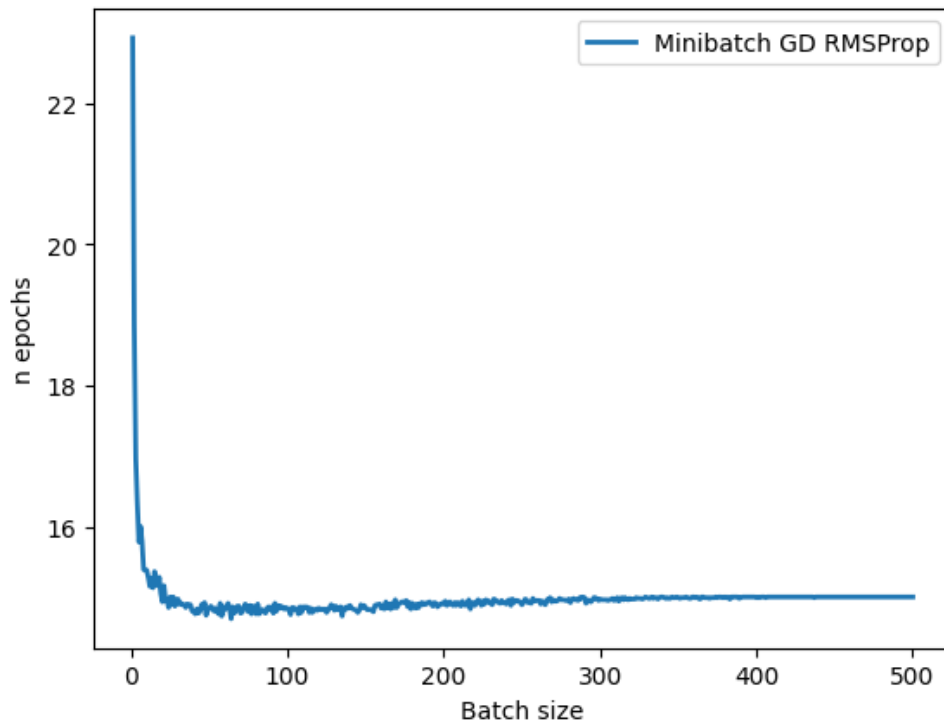
Модификация *RMSProp* решает возникшую проблему с AdaGrad, в качестве решения было предложено усреднять значения. Пусть  $(g_i)^2$  – покомпонентное возведение в квадрат,  $\delta$  – некое число порядка  $[10^{-10}, 10^{-7}]$ , дабы по формуле не было деления на ноль. Тогда, в качестве скорости возьмем:

$$s_{i+1} = \lambda \cdot s_i + (1 - \lambda) \cdot (g_i)^2, \quad \lambda \in (0, 1)$$

Наконец, изменение весов:

$$w_{i+1} = w_i - \alpha \cdot \frac{g_i}{\sqrt{s_{i+1} + \delta}}, \quad \alpha = \text{const}$$

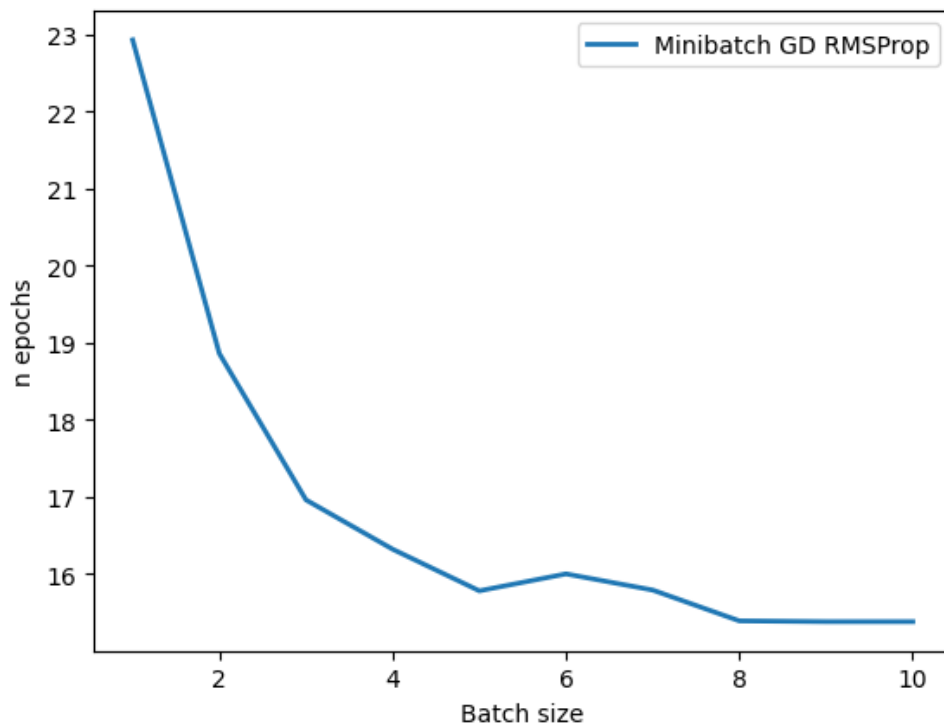
Для исследования алгоритма мы выставим ровно те же настройки, что были у AdaGrad. Рассмотрим же график для RMSProp.



RMSProp in general

Даже анализа не нужно, чтобы понять, количество необходимых шагов до сходимости резко возрастает в связи с более равномерным изменением весов, одновременно решая глобальную проблему, тянувшейся еще с Momentum.

Более детальное изображение RMSProp при первых десяти batch размерах:



RMSProp in  $[1, 10]$

## Adam

Начинаем объединение веселья предыдущих. В отличие от всех его предшественников модификация *Adam* является наиболее распространенным и используемым.

$$\begin{aligned}v_{i+1} &= \lambda_1 \cdot v_i + (1 - \lambda_1) \cdot g_i \\s_{i+1} &= \lambda_2 \cdot s_i + (1 - \lambda_2) \cdot (g_i)^2 \\ \hat{v}_{i+1} &= \frac{v_{i+1}}{1 - \lambda_1^{i+1}} \\ \hat{s}_{i+1} &= \frac{s_{i+1}}{1 - \lambda_2^{i+1}}\end{aligned}$$

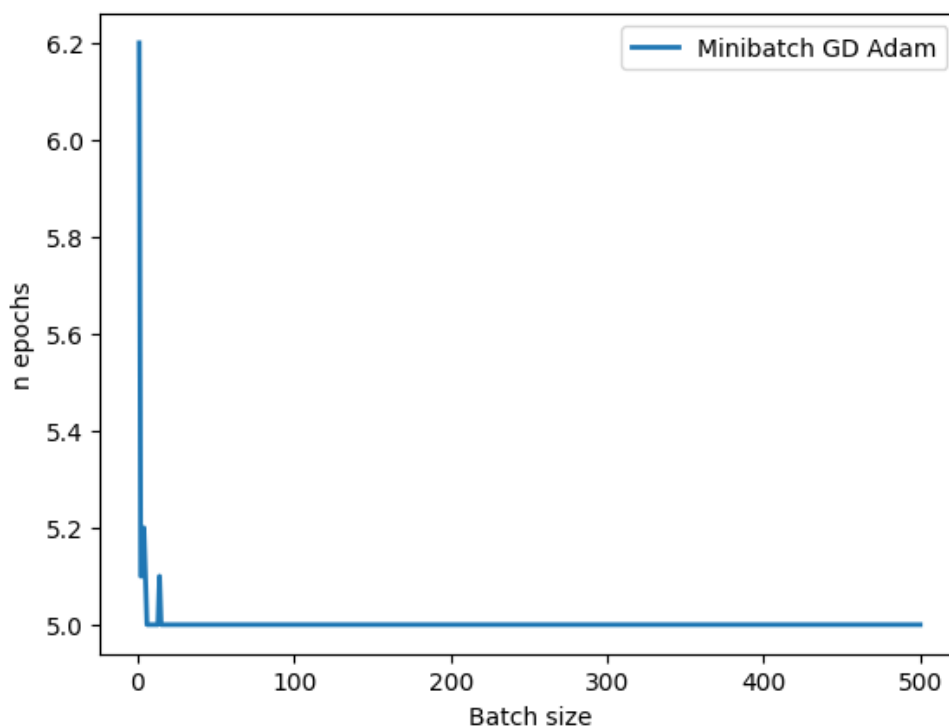
Наконец, изменение весов будем рассчитывать по следующей формуле:

$$w_{i+1} = w_i - \lambda \cdot \frac{\hat{v}_{i+1}}{\sqrt{\hat{s}_{i+1} + \delta}},$$

где  $\lambda_1 = 0.9$ ,  $\lambda_2 = 0.999$  и  $\delta = 10^{-8}$ . Настроим наше исследование и запустим с следующими параметрами:

- ▷ Максимальное количество эпох – 4000.
- ▷ Начальным `learning_rate` –  $5e - 1$ .
- ▷ Количество генерируемых точек – 500.

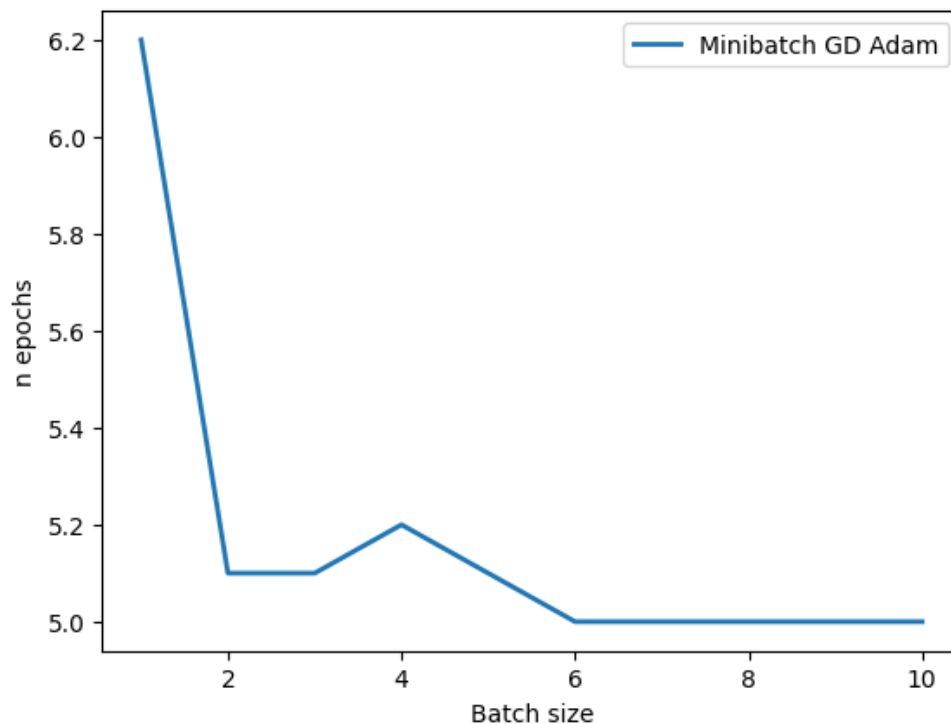
Рассмотрим же график для Adam.



Adam in general

Даже анализа не нужно, чтобы понять, количество необходимых шагов до сходимости резко возрастает в связи с более равномерным изменением весов, одновременно решая глобальную проблему, тянувшейся еще с Momentum.

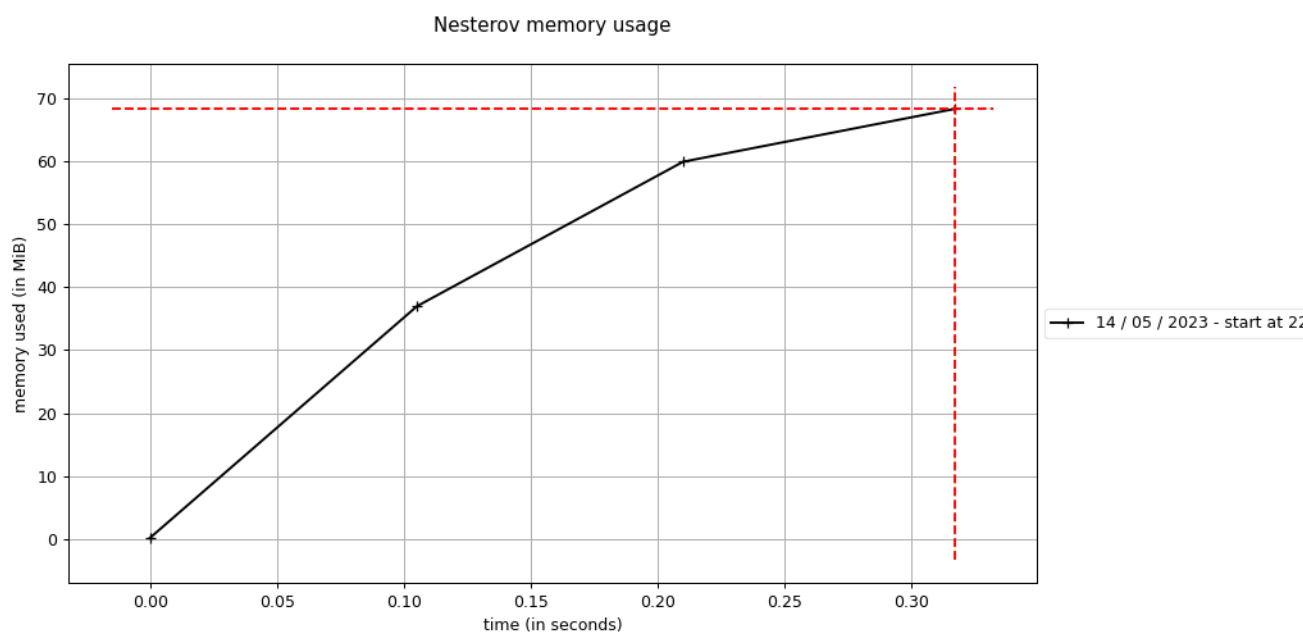
Более детальное изображение Adam при первых десяти batch размерах:



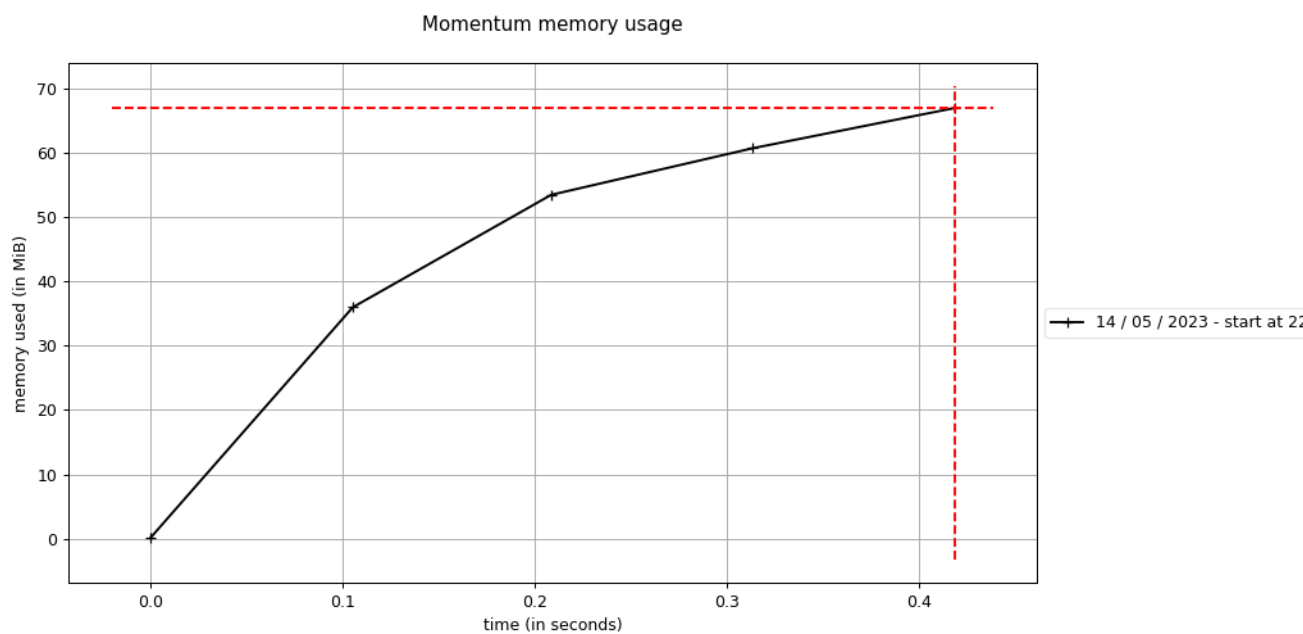
Adam in  $[1, 10]$

### Сравнение модификаций

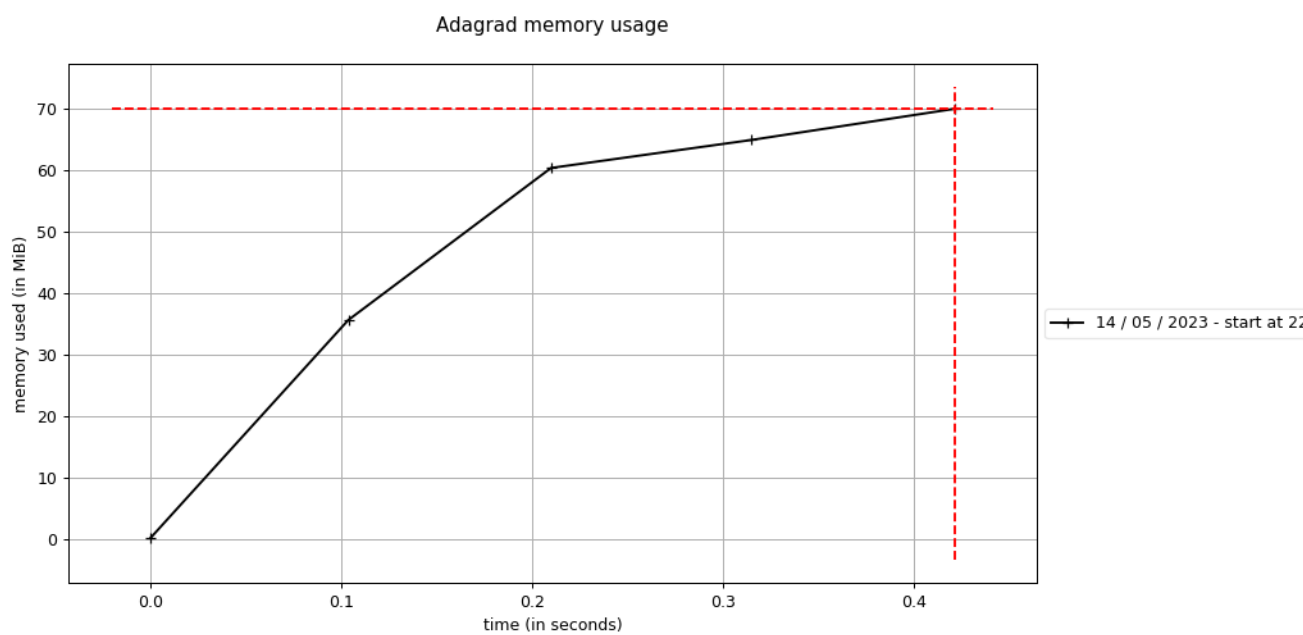
**Сходимость.** Ниже представлены графики зависимости времени и памяти на все созданные выше модификации с подписанными названиями.



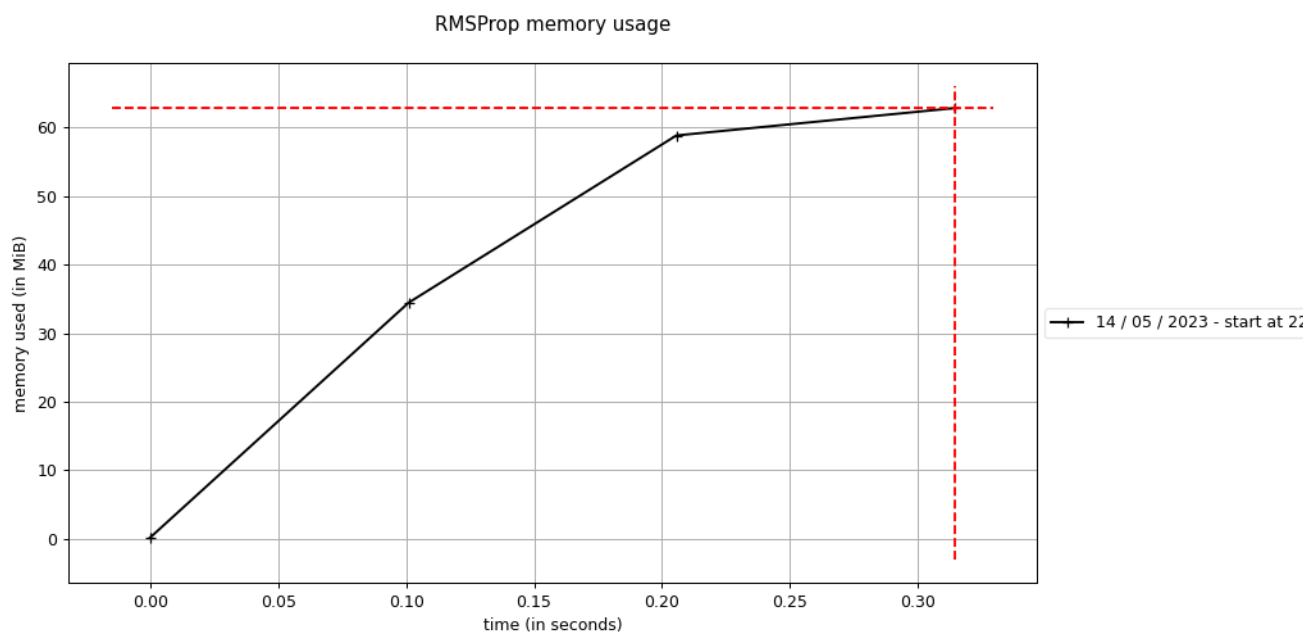
Nesterov/Time/Memory



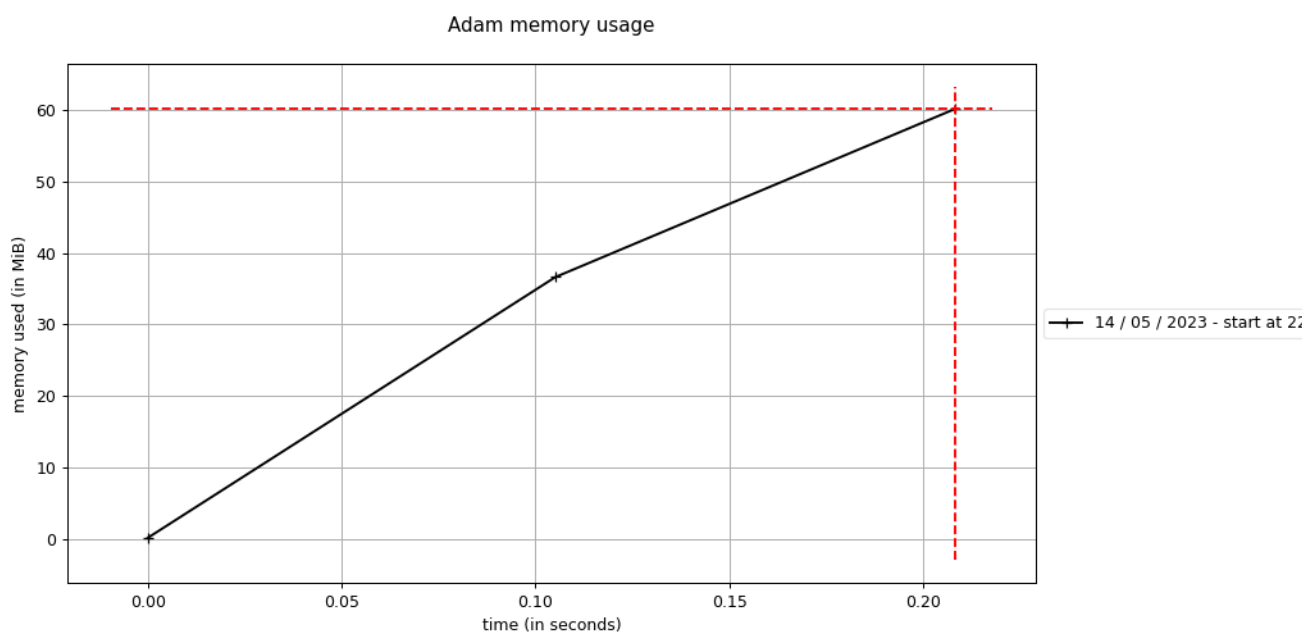
Momentum/Time/Memory



Adagrad/Time/Memory



RMSProp/Time/Memory

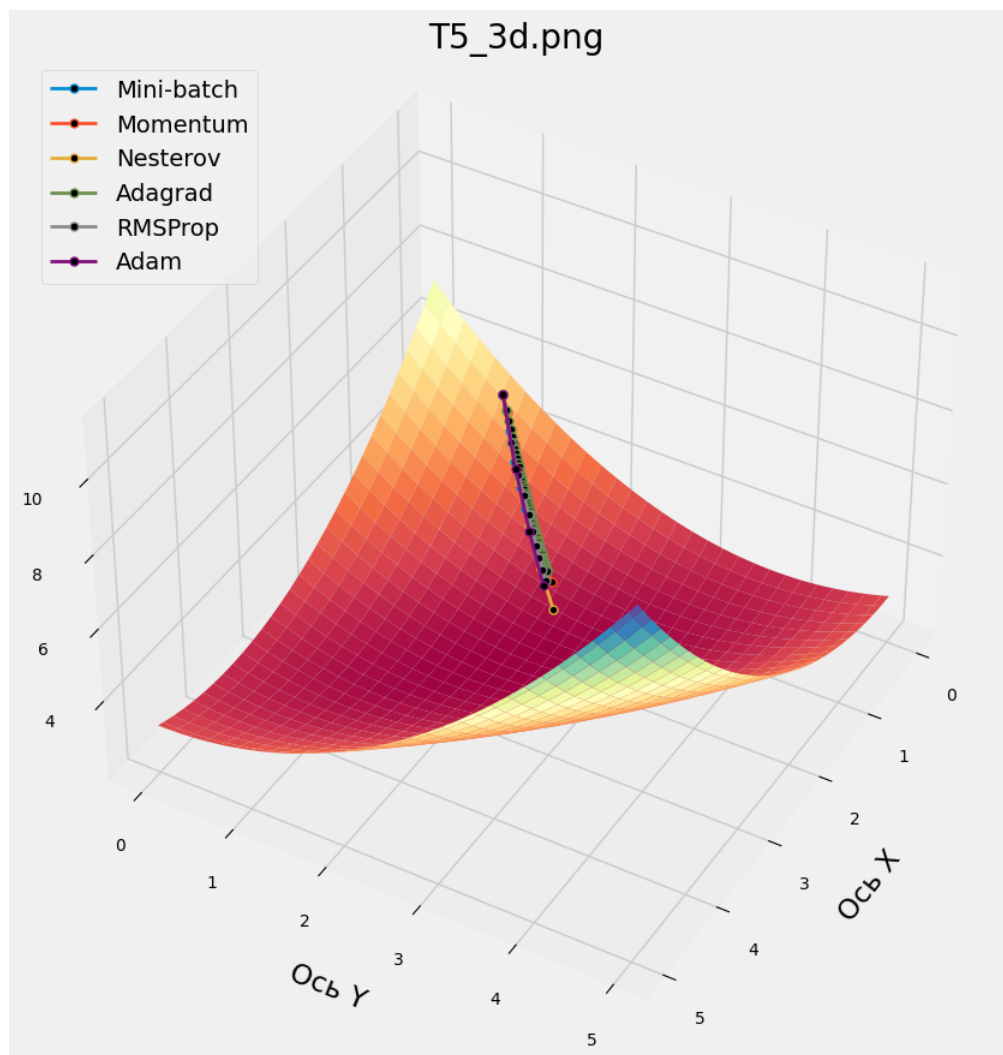


Adam/Time/Memory

Также представим таблицу использования арифметических операций в разных функциях, которые имплементированы в модификациях.

Функция из имплементации	Количество арифметических операций
<code>..._with_lr_scheduling_and_moment</code>	144539
<code>..._with_lr_scheduling_and_nesterov_moment</code>	147550
<code>..._with_lr_scheduling_and_adagrad</code>	511881
<code>..._with_lr_scheduling_and_RMSProp</code>	48187
<code>..._with_lr_scheduling_and_Adam</code>	12055

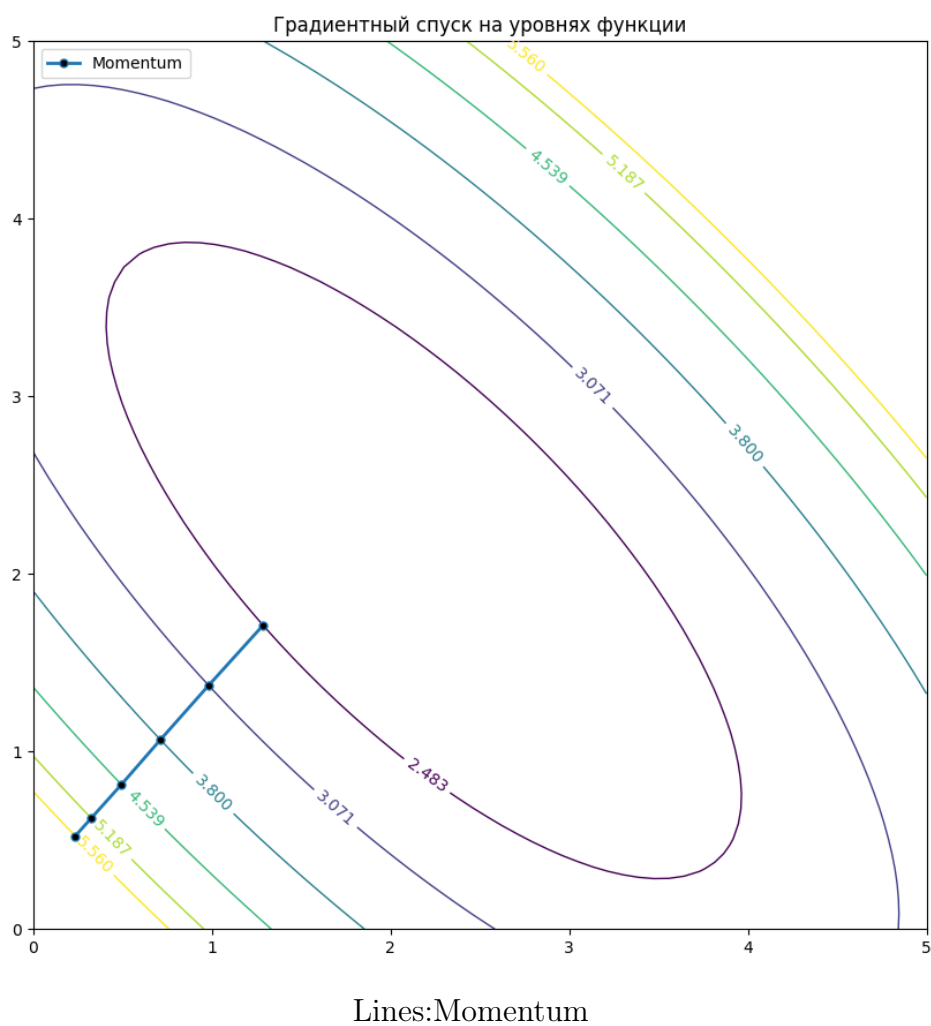
**Траектории.** Траектория спуска различных алгоритмов из одной и той же исходной точки с одинаковой точностью.



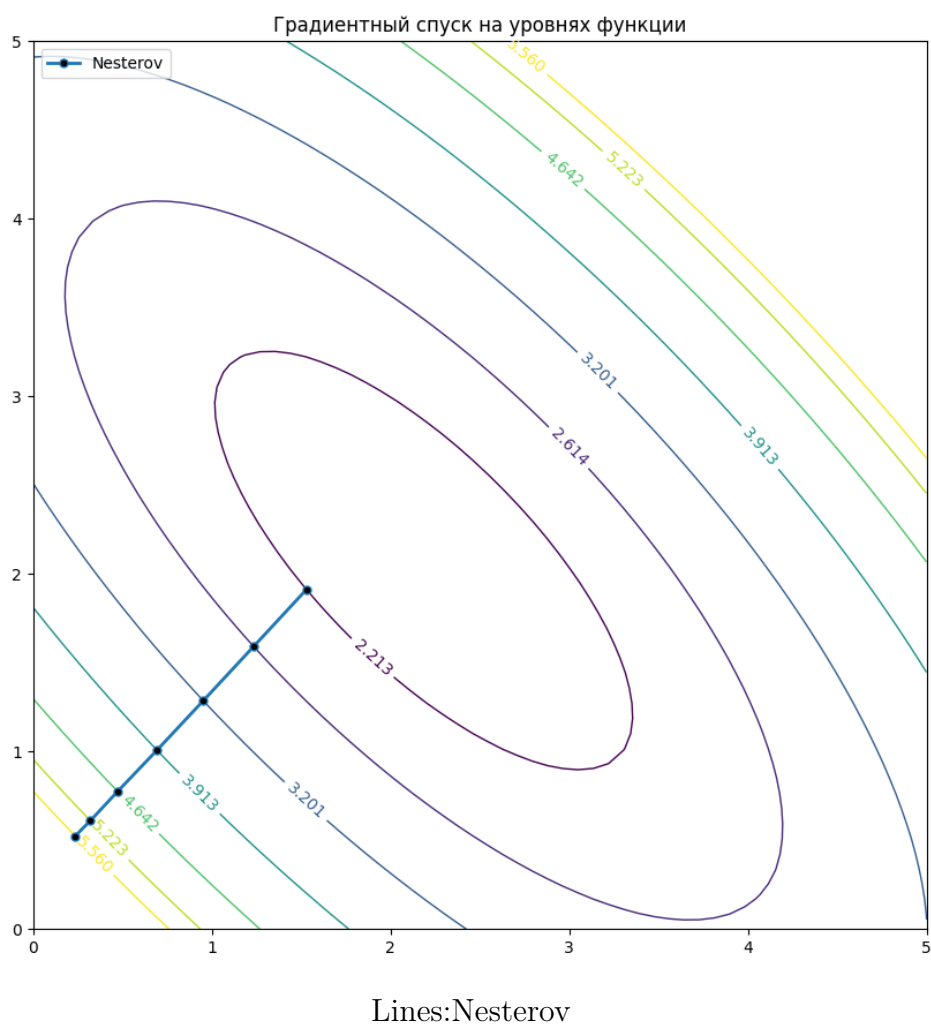
Directions



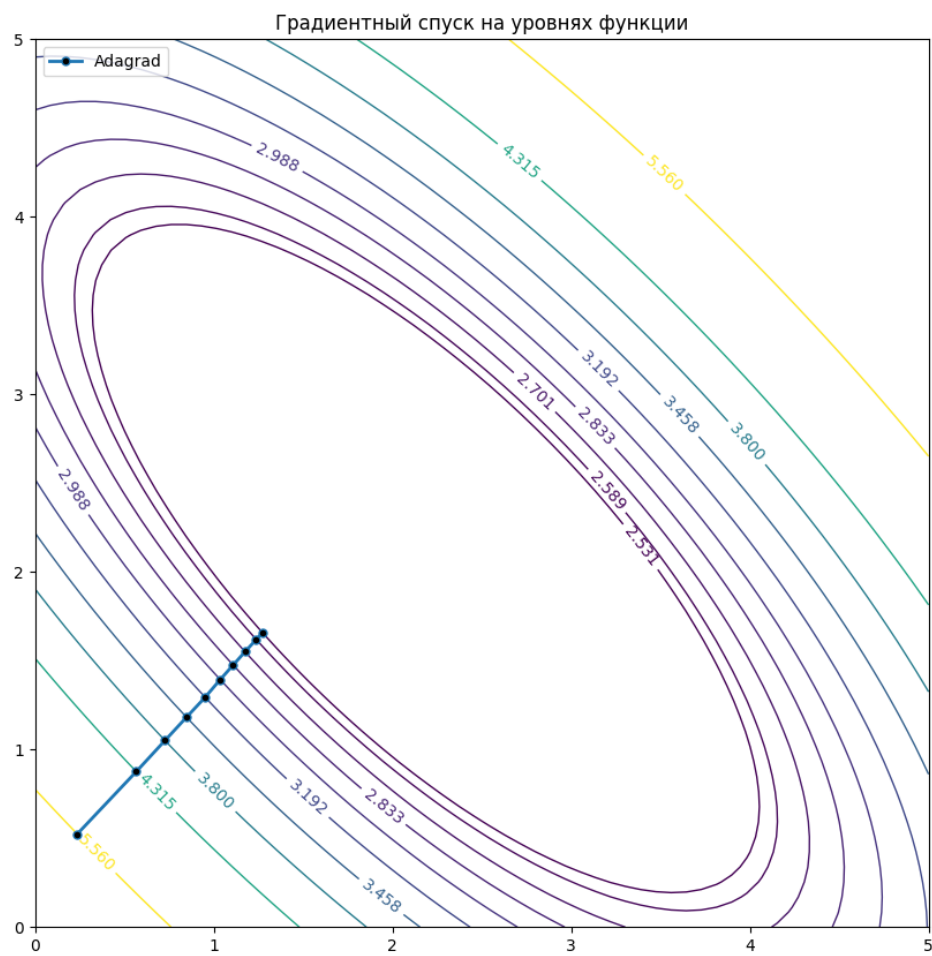
Рисунок с линиями равного уровня для Momentum.



Для Nesterov:

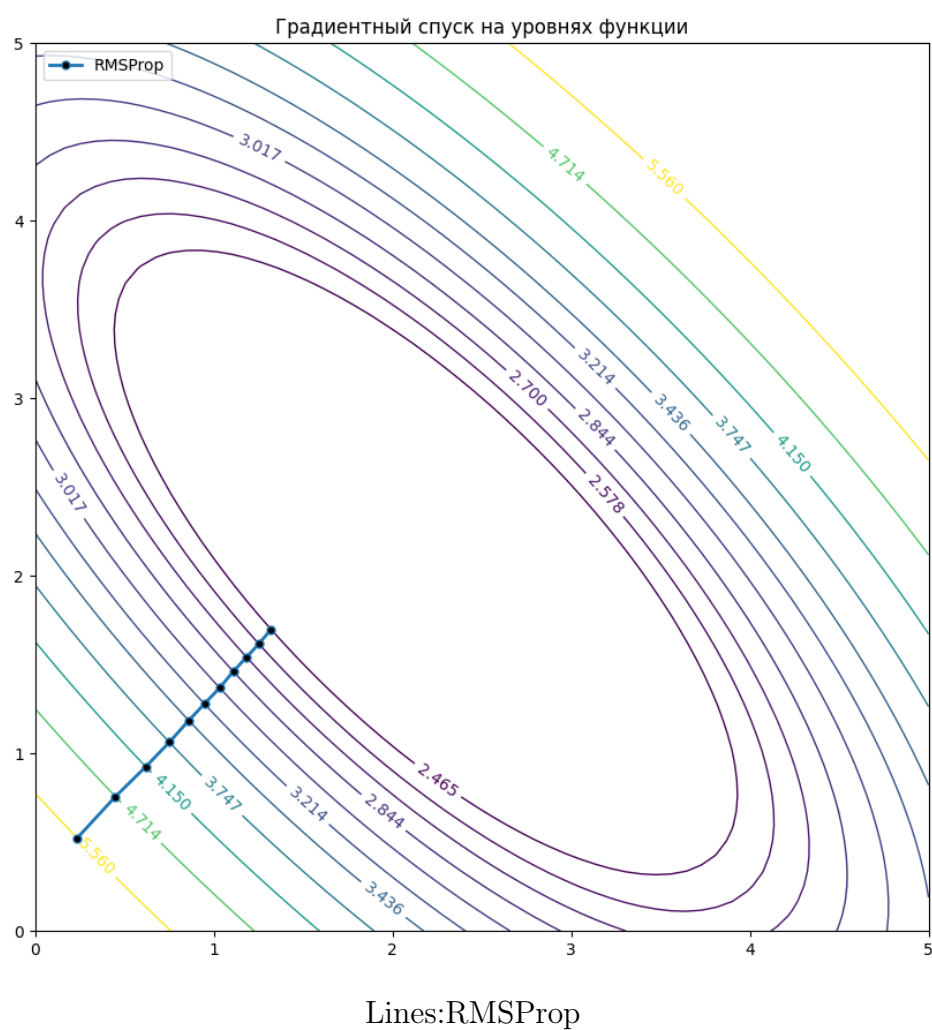


Для AdaGrad:

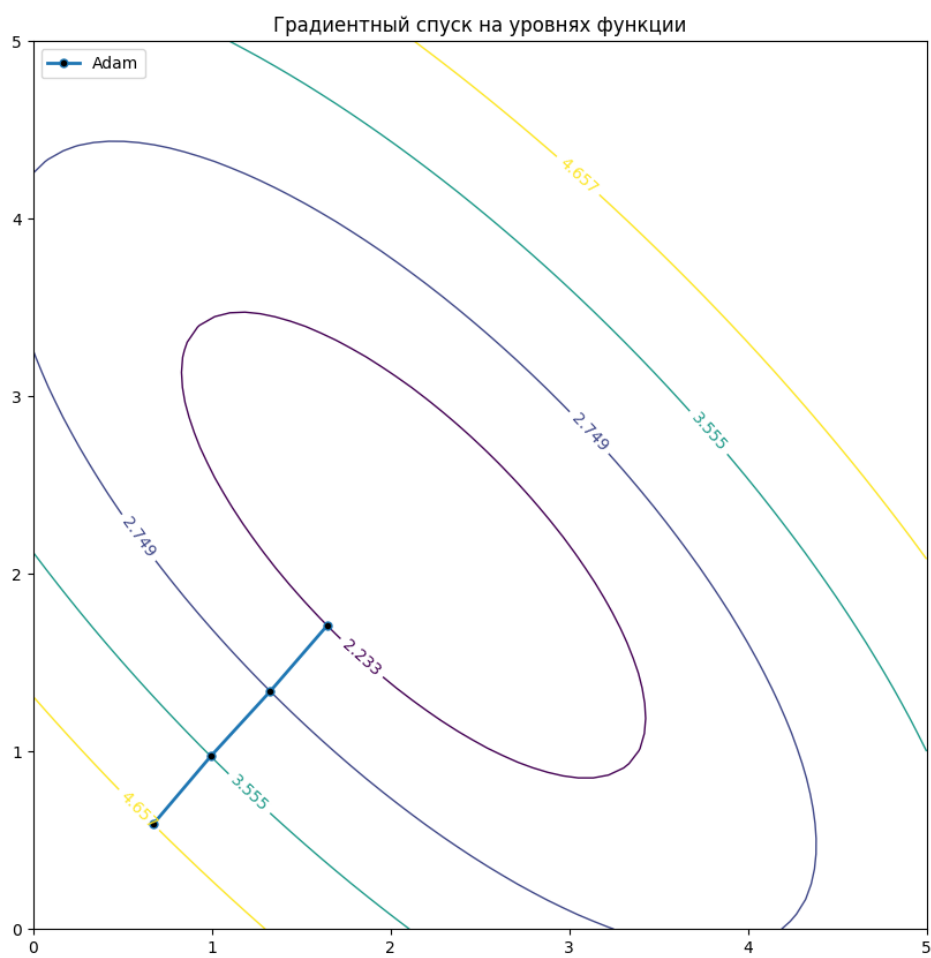


Lines:AdaGrad

Для RMSProp:



Наконец, для Adam:



Lines:Adam

# Полиномиальная регрессия

1. Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.
2. Модифицируйте полиномиальную регрессию добавлением регуляризации в модель (L1, L2, Elastic регуляризации).
3. Исследуйте влияние регуляризации на восстановление регрессии.

## Введение

Представим себе некую функцию  $f : \mathbb{R} \rightarrow \mathbb{R}$  причем такую, что она не зависит от каких-то побочных факторов (случайные значения и тому подобное), и при этом получаемые экспериментальные данные были как бы «разбросаны» относительно некой обобщающей кривой. Нашей задачей состоит в понимании, что это за кривая – по сути, это множество пар  $M = \{\langle x, y \rangle : x \in \mathbb{R}, y \in E(f(x))\}$ , которые при переходе от заданного  $x' \rightarrow x''$  по некоторому  $\delta$ , подает предугадываемое значение  $y''$  на основе полученных ранее начальных экспериментальных данных. Говоря простыми словами, с помощью данной прямой мы хотим получать прогноз.

Рассмотрим идею *полиномиальной регрессии*. Пусть даны  $x_i \in X$  и  $y_i \in Y$ . Тогда уравнением полинома будет иметь следующий вид в нашей задаче:

$$y_i = \sum_{j=0}^k a_j \cdot x_i^j,$$

где  $\{a\}$  – это коэффициенты полинома, где  $a_0$ , их мы ищем ровно тем же способом, что и в первой задаче – по методу наименьших квадратов. Наконец, по аналогии с задачей градиентного спуска, мы получаем следующее выражение:

$$L = \omega \cdot \left\| \sum_{i=1}^n (\mathbf{y}_i - y_i)^2 \right\| \rightarrow \min,$$

где  $\{\mathbf{y}\}$  – это значения полинома в точках  $\{x\}$ , тогда наше выражение преобразуется:

$$L = \omega \cdot \left\| \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right)^2 \right\| \rightarrow \min,$$

где  $\omega = \frac{1}{n}$ , для того чтобы усреднить квадратичные ошибки на всех обучающих примерах. Это позволяет получить более сбалансированную и интерпретируемую метрику ошибки, не зависящую от размера обучающей выборки. Для получения градиента исследуемой функции мы воспользуемся производной от  $L$  по некоторому коэффициенту  $a_i$ , тогда

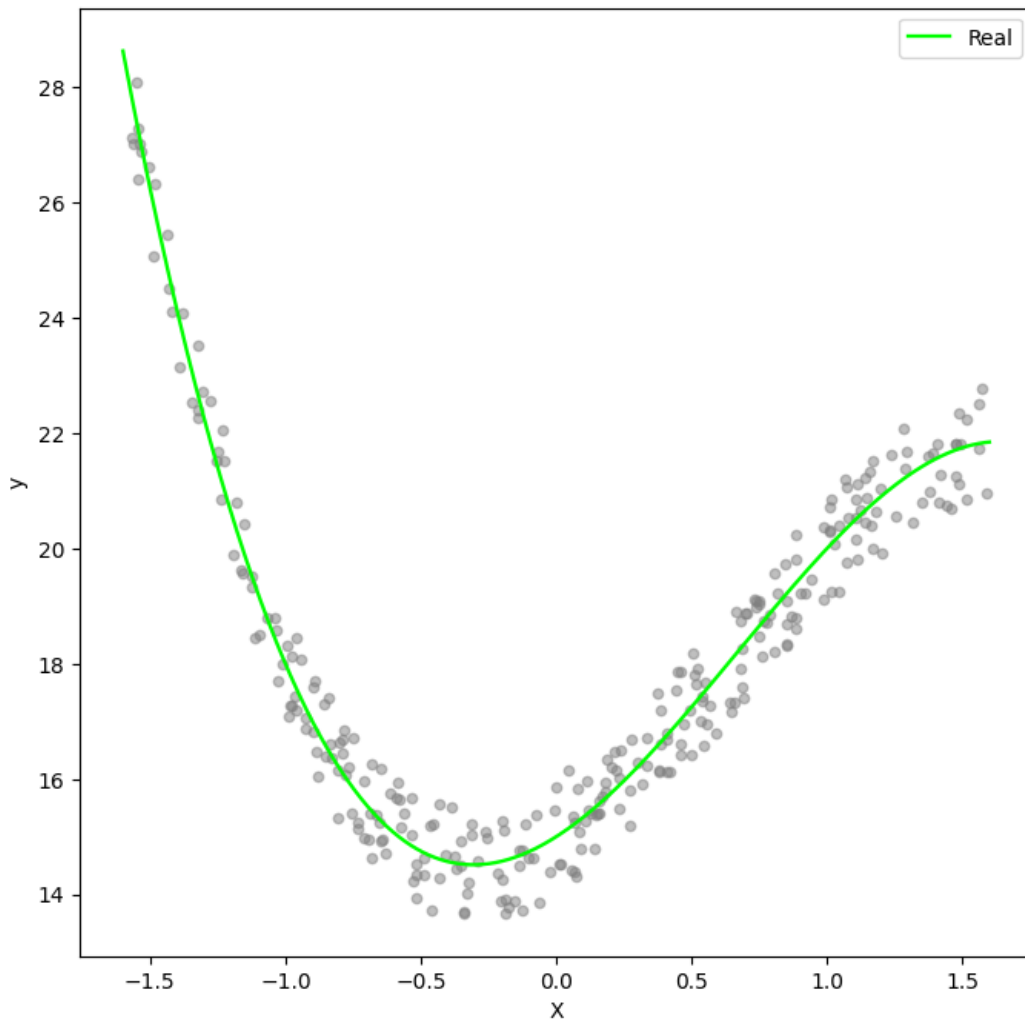
$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right) \cdot x_i^i$$

## Пример с простым полиномом

В качестве демонстрации алгоритма полилинейной регрессии мы возьмем «простой», в том смысле, что он однозначным образом задается, полином третьей степени:  $y = 15 + 3 \cdot x + 4 \cdot x^2 + (-2) \cdot x^3$ . Сгенерируем точки относительно прямой с такими параметрами:

```
dots_count = 800 (количество генерируемых точек)
variance = 1 (вариативность случайности, np.uniform(-1, 1) * variance)
```

На выходе получаем такой график, где серым отмечены сгенерированные данные для алгоритма, а зеленым – реальный график.

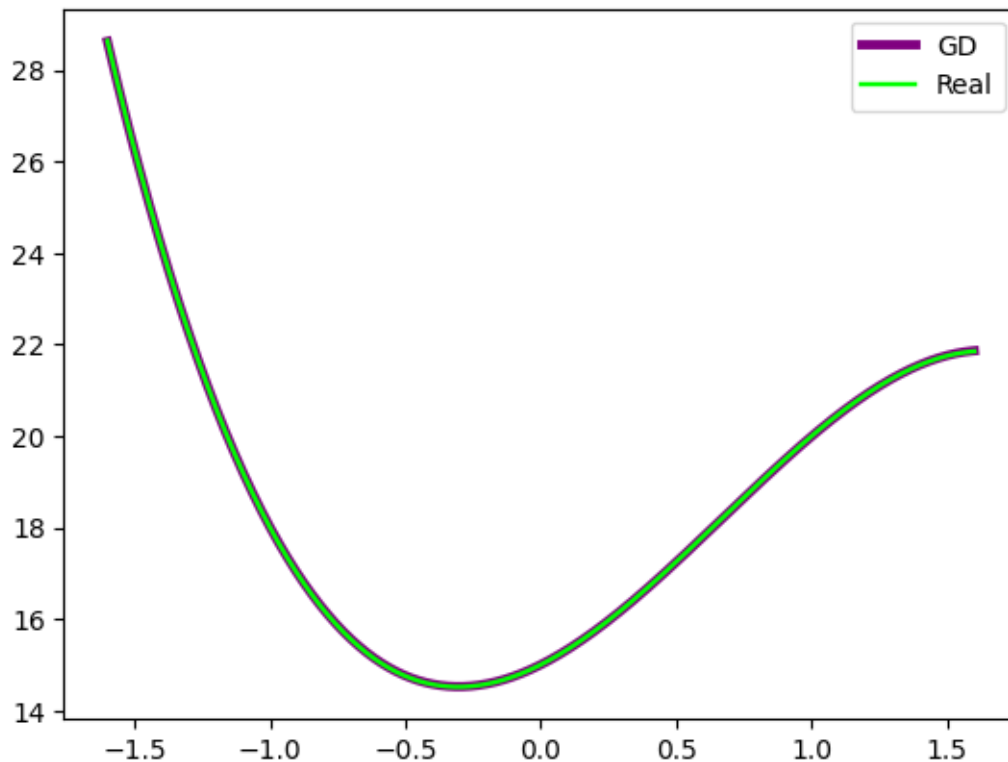


Poly 4th deg (dataset)

А теперь восстановим кривую по заданным точкам. Зададим ему следующие параметры на подбор функции по сгенерированным данным:

```
batch_size = 50 (количество точек или, по простому, число n)
max_epochs = 500 (максимальное число шагов до схождения)
learning_rate = 0.05 (шаг в схождении к минимуму)
```

Получаем следующий график. Здесь фиолетовым цветом отмечена восстановленная кривая, а зеленым – реальный график.



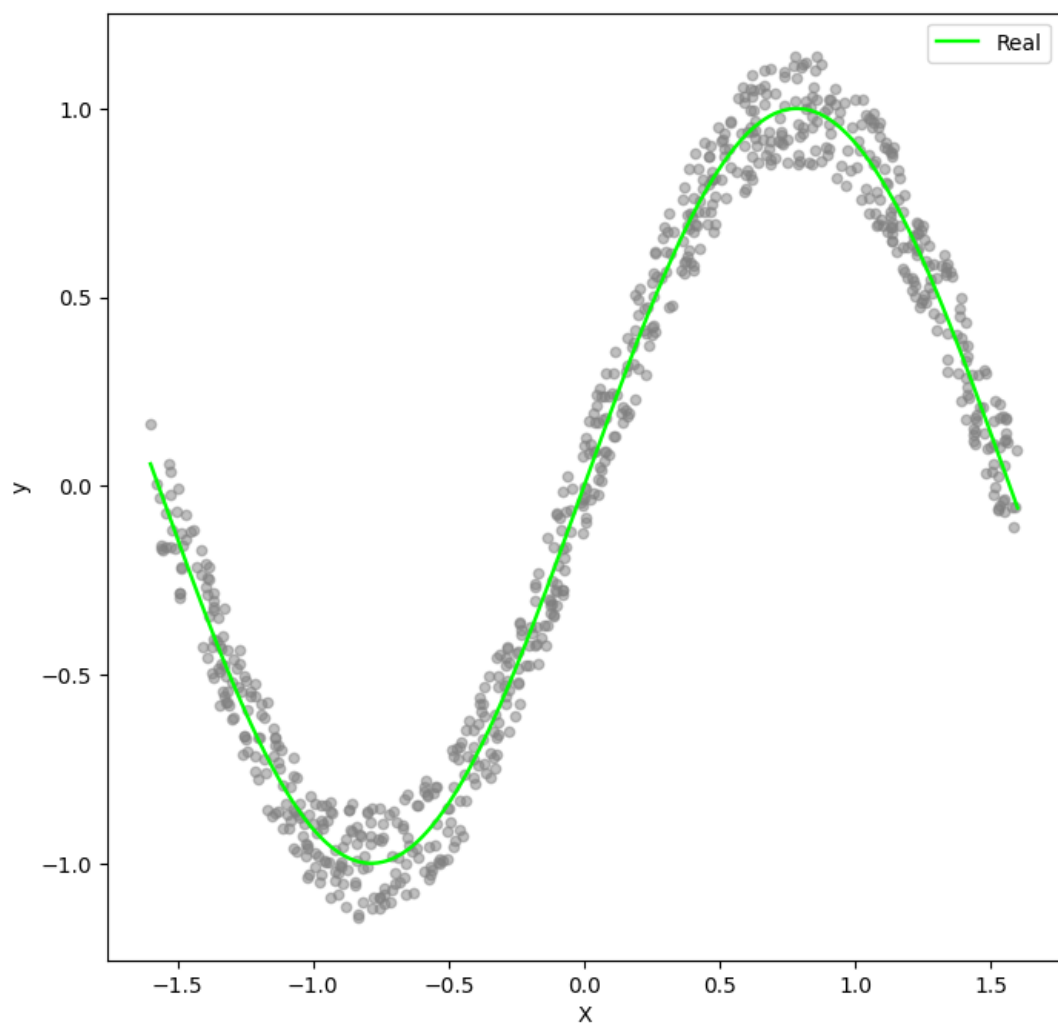
Poly 4th deg (real vs. recovered)

Заметим, что восстановленная кривая полностью совпала с реальным графиком.

### Пример с полиномом поинтереснее

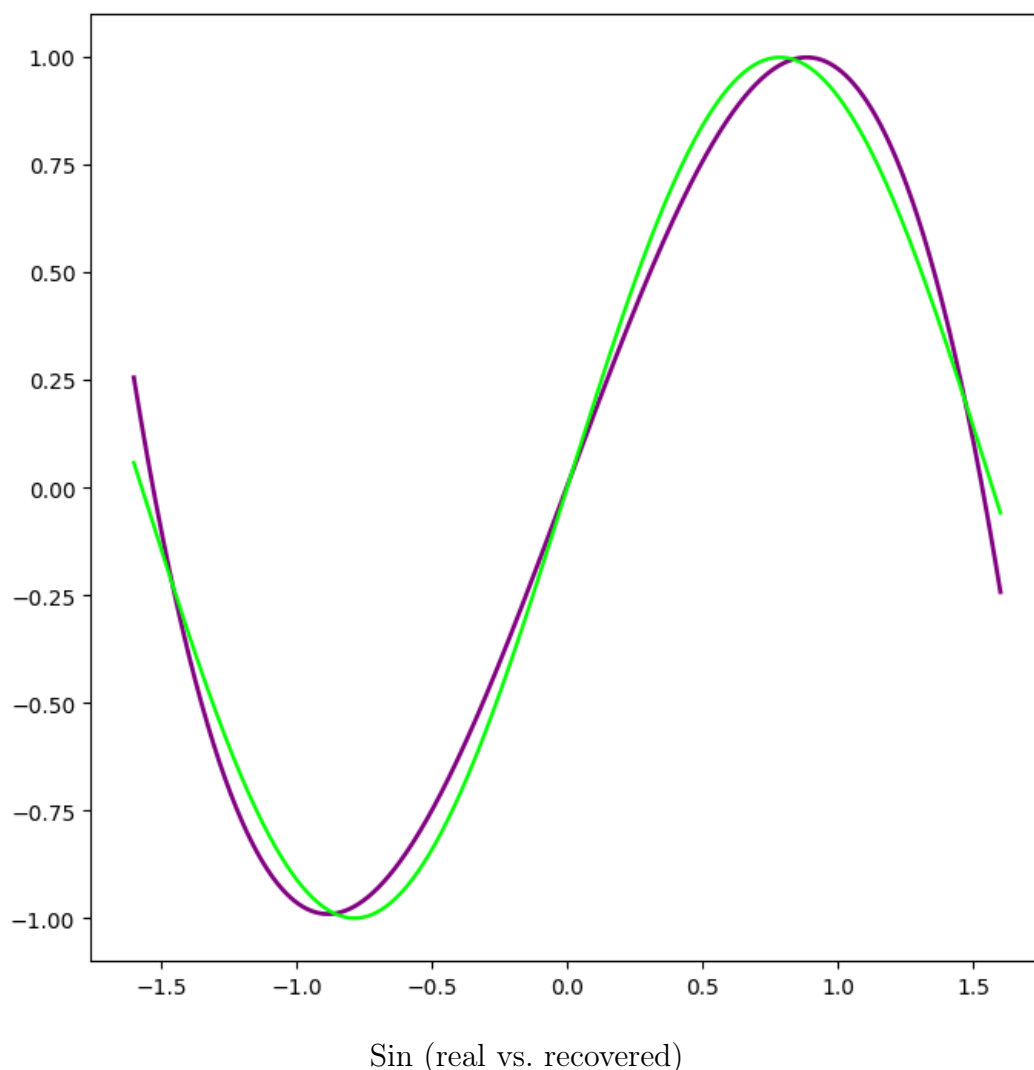
А теперь в качестве полинома мы возьмем такой, коэффициенты которого сгенерированы с помощью функции  $y = \sin x$ . На выходе получаем такой график, где серым отмечены сгенерированные данные для алгоритма, а зеленым – реальный график.





Sin (dataset)

Восстановим кривую по заданным точкам. Получаем следующий график. Здесь фиолетовым цветом отмечена восстановленная кривая, а зеленым – реальный график.



Заметим, что нынешняя восстановленная линия, теперь идеальным образом не совпадает. Во-первых, потому-что это не ограниченный полином, как раньше было, а нечто посложнее ввиду коэффициентов. Во вторых, на самом деле, и первый пример не был столь идеальным, как это могло бы показаться на первый взгляд, и здесь мы отчетливо видим смещение функции: это происходит из-за некоторых внезапных слишком больших или маленьких шагов. Данную проблему решают уже непосредственно модификации.

## Исследование различных модификация

Как и в задаче с градиентным спуском нам бы хотелось получать более реальные «гладкие» оценки предсказания значения, вместо имения «гор» и «впадин» кривой. Эта проблема может быть решена с помощью модификаций к методу, которые, по своей сути, представляют из себя отдельные слагаемые.

### L1

*Идея L1 регуляризации.* Давайте с каждой новой итерации мы будем неким способом «наказывать» нашу модель за высокие веса, чтобы в будущем увеличить обобщающую способность. Эта регуляризация основана на добавлении еще одного слагаемого

в основную сумму, равного абсолютному значению коэффициентов полинома.

$$L = \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right)^2 + \lambda \cdot \sum_{i=0}^k (|a_i|),$$

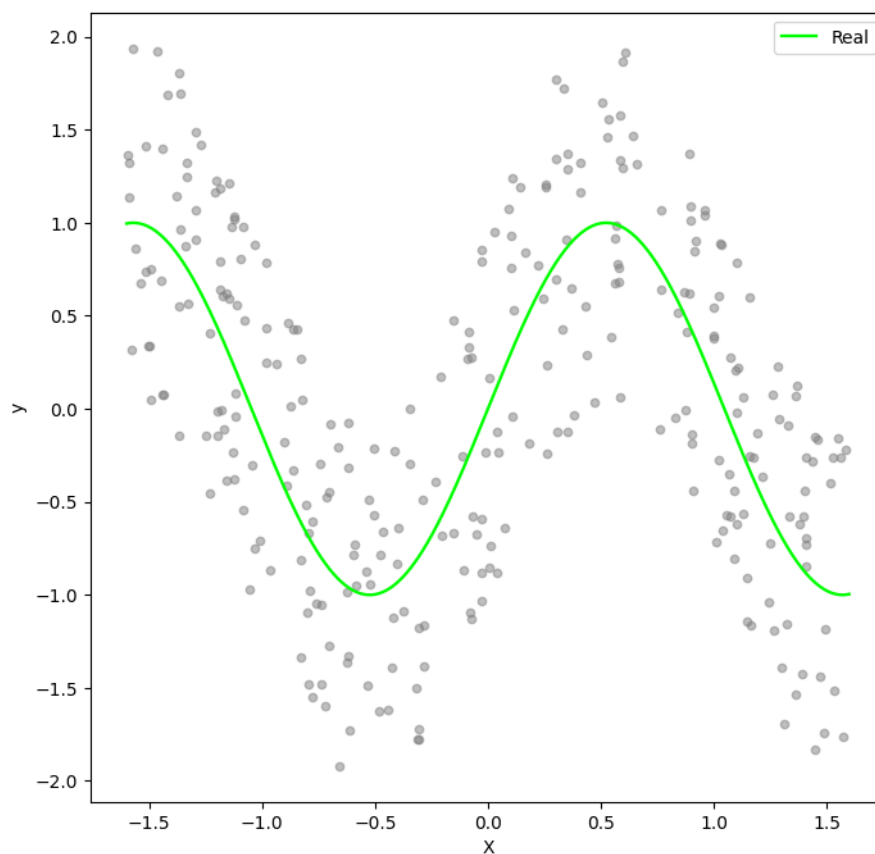
где  $\lambda$  – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right) \cdot x_i^i + \lambda$$

**Пример** В качестве примера мы возьмем относительно небезобидную функцию  $f(x) = \sin 3 \cdot x$  и используем её как изначальный набор данных для восстановления кривой с помощью L1-регуляризации. Зададим некоторые параметры для исследования:

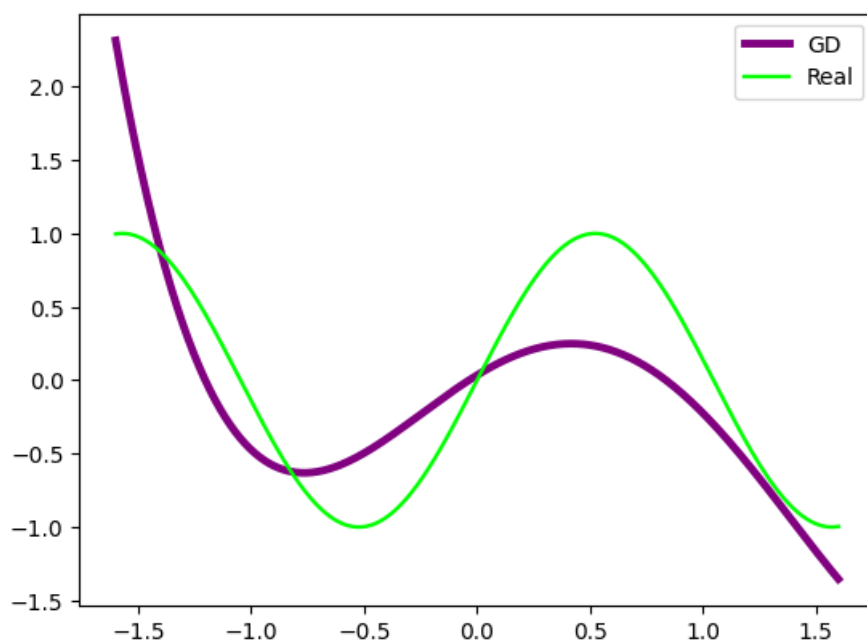
```
density = 8000 количество точек на плоскости
dots_count = 300 количество генерируемых точек
dist = 1.6 расстояния от и до генерации плоскости
```

Наконец, в качестве коэффициента  $\lambda = 0.3$ , так как предполагается, что штраф в виде суммы абсолютных значений будет слишком высокой, из-за чего рассматриваемая MSE-функция не сможет нормально приблизиться к реальным значениям. Кроме того,  $\sin 3 \cdot x$  обладает довольно-таки неприятельским свойством: она периодична (что в скором времени всплывет) и в пределе дает неопределенность. Итак, **зеленым** здесь обозначена реальная кривая, **серым** – сгенерированные точки для восстановления кривой.



Dataset of  $\sin 3 \cdot x$

Теперь, посмотрим на восстановленную кривую.



Recover of  $\sin 3 \cdot x$

Итак, наша кривая пытается быть похожей на своего соперника, но примерно

никак не подходит. Почему так? Как раз обращаемся к великому рандому точек и достаточно странной функции, которая не имеет какого-либо конечного полинома и все ошибки, связанные с почти правильным схождением восстановления кривой к реальным данным, уже относятся к коэффициентам кривой. В дальнейшем, а именно в L2 и Elastic мы также вернемся к этой функции.

## L2

*Идея L2 регуляризации.* Данная модификация является небольшим дополнением к предыдущему в виде большей гладкости производной и возможностью лучшей работой с градиентным спуском. Проблема L1 заключалась в том, что в некоторых местах, где производная не может быть установлена, имеет резкие «горы», «впадины» и вообще разрывы. Тогда, на помощь приходит немного иное штрафное значение функции – вместо суммы значений коэффициентов полинома мы возьмем квадраты. Отличает этот подход от L1 тем, что теперь мы штрафует больше, но при этом никогда не зануляемся, как это было с предыдущей модификацией.

$$L = \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right)^2 + \lambda \cdot \sum_{i=0}^k (a_i^2),$$

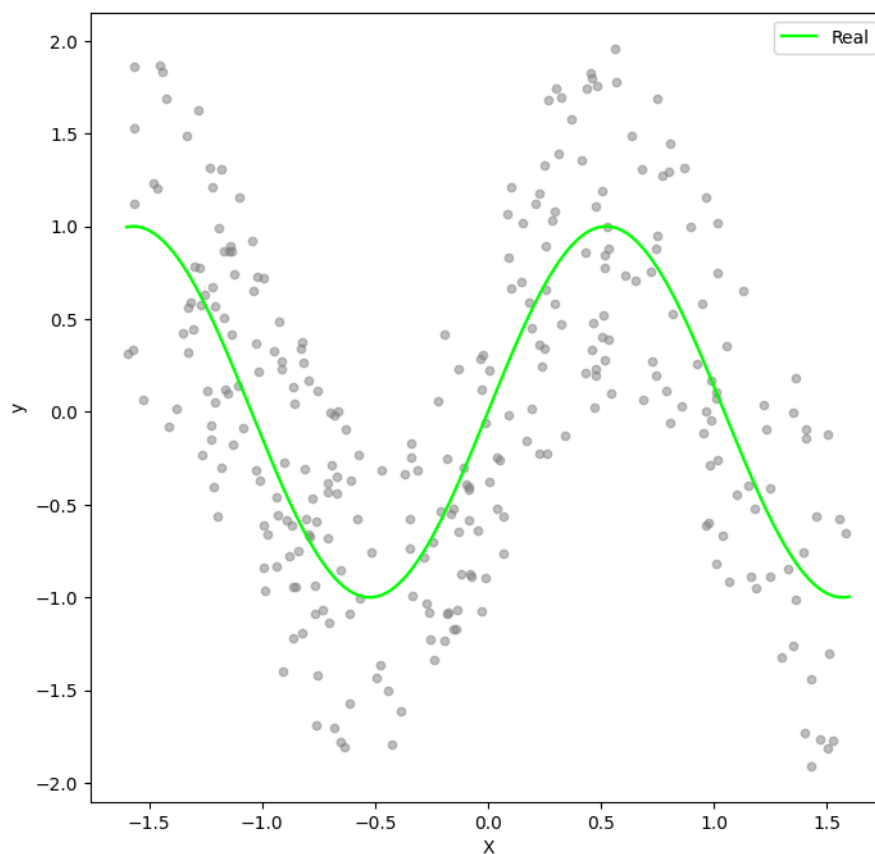
где  $\lambda$  – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right) \cdot x_i^i + \lambda \cdot a_i$$

**Пример** Опять же рассмотрим кривую, образованная функцией  $f(x) = \sin 3 \cdot x$ , как и в предыдущей регуляризации. Подставим те же настройки генерации точек, но в качестве основного коэффициента, который управляет наказанием, возьмем:

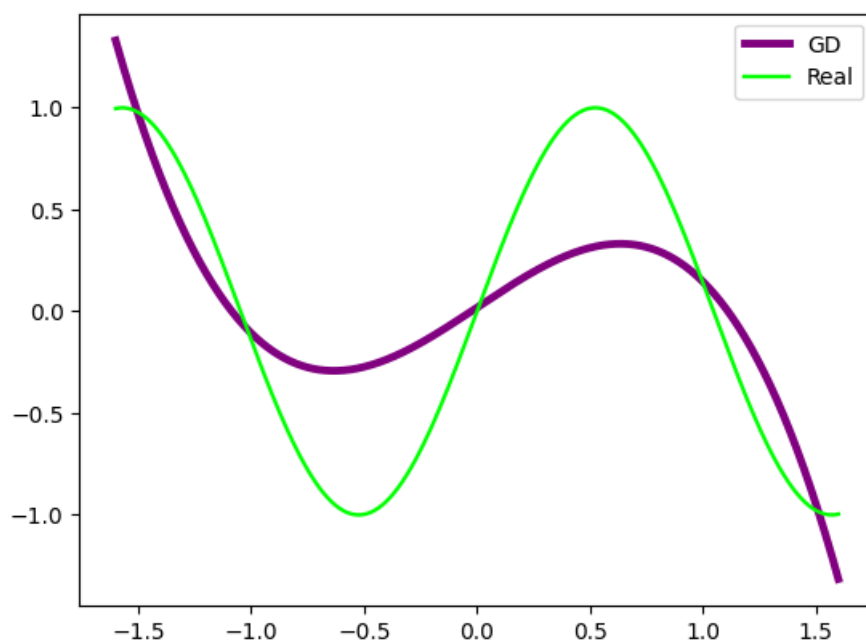
$$\lambda = 0.09,$$

объяснение выборности которого примерно такое же, что и в случае с L1: мы не хотим слишком сильно наказывать функцию, поэтому давайте попробуем убрать штраф из суммы квадратов значений. Утверждается, что мы получим значения все равно чуть больше, чем у L1. Итак, **зеленым** здесь обозначена реальная кривая, серым – сгенерированные точки для восстановления кривой.



Dataset of  $\sin 3 \cdot x$

Теперь, посмотрим на восстановленную кривую.



Recover of  $\sin 3 \cdot x$

Уже лучше, но все еще не хорошо. Проблема в том, что у нас как будто не

достаточно эвристически выбраны коэффициенты для наказания функции. Хочется как-то управлять сразу обоими методами регуляризации. . .

## Elastic регуляризация

Идея *Elastic регуляризация*. Объединение двух предыдущих идей, как и полезных особенностей, свойств обоих методов, при этом, корреляция переменных такая, что данная регуляризация не обнуляет некоторые из них, как в случае с L1.

$$L = \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right)^2 + \lambda_1 \cdot \sum_{i=0}^k (|a_i|) + \lambda_2 \cdot \sum_{i=0}^k (a_i^2),$$

где  $\lambda_1$  и  $\lambda_2$  – есть коэффициенты для регуляризации влияния штрафов L1 и L2 на модель соответственно. Градиентом для модели будет выступать тогда такая формула:

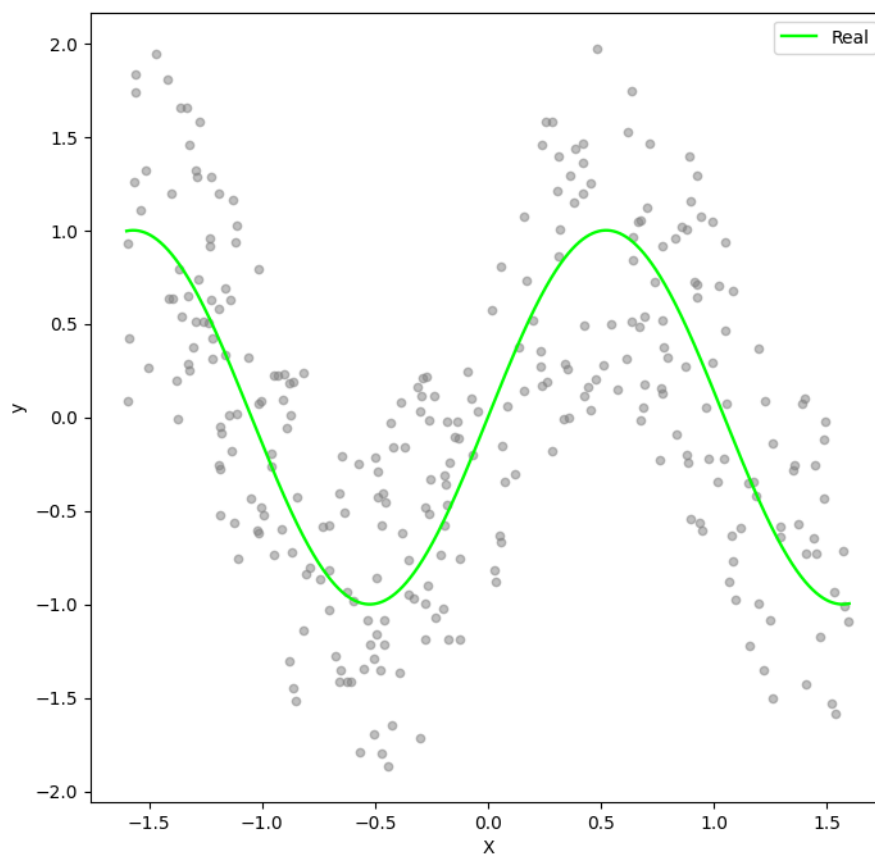
$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left( \sum_{j=0}^k (a_j \cdot x_i^j) - y_i \right) \cdot x_i^i + \lambda_1 + \lambda_2 \cdot a_i$$

**⟨экспериментальный⟩ Пример** Наконец, мы подошли к более грозному оружию, всё также пытаюсь побороться с  $f(x) = \sin 3 \cdot x$ . Подставим всё те же настройки генерации точек, но в качестве обоих коэффициентов, назову всем, выберем вот такие:

$$\lambda_1 = 0.15$$

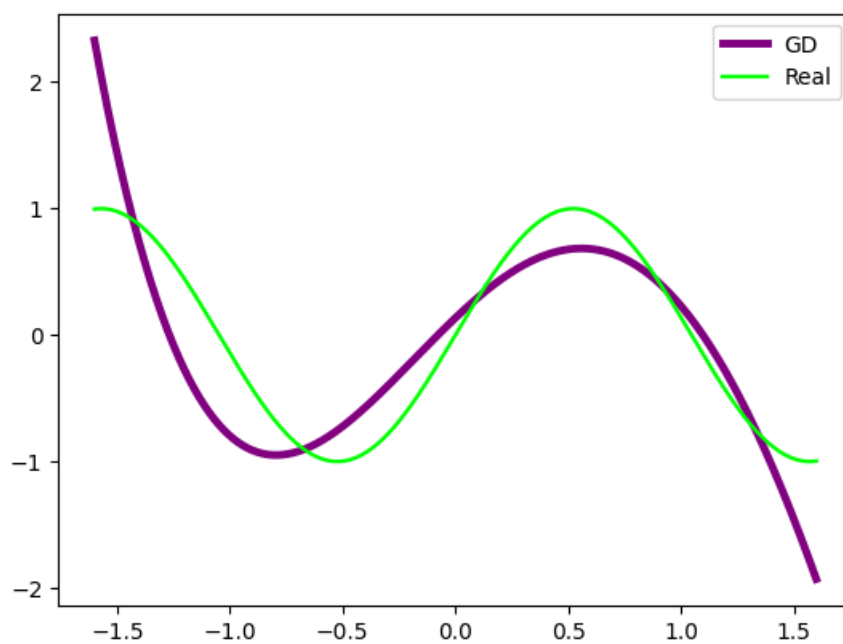
$$\lambda_2 = -0.03$$

Опустим пока момент с отрицательным коэффициентов, рассмотрим пока так и кое-что поймем далее. Итак, **зеленым** здесь обозначена реальная кривая, **серым** – сгенерированные точки для восстановления кривой.



Dataset of  $\sin 3 \cdot x$

Теперь, посмотрим на восстановленную кривую.



Recover of  $\sin 3 \cdot x$

Попытаемся объяснить, почему с отрицательным коэффициентом восстановлен-



ная кривая стала выглядеть на порядок лучше, чем ранее. Заметим, что наказание – это не поощряемое **большой** коэффициент, который говорит, что функция при следующем шаге не должна делать настолько большой шаг, как в текущем случае. Но здесь, поскольку один из коэффициентов уже отрицателен, то мы, идейно, поощряем ошибку, тем самым у полинома, который мы получаем в шагах алгоритма, коэффициенты становятся больше. Что мы замечаем при быстром взгляде «горки» справа: в L1 и L2 восстановленное нечто почти не двигалось в направлении реальной точки, а Elastic с поощрением ошибок внезапно стал сильно ближе к точке. Вероятнее всего, как это было сказано еще при рассмотрении L1, это связано с самой функцией, а именно её свойствах об отсутствии внятного предела (он не определен) и периодичностью.