Методы оптимизации 2

Назаров Владимир Целиков Владислав

19 июня 2023 г.

1 Реализуйте стохастический градиентный спуск для решения линейной регрессии.

1.1 Линейная регрессия

Регрессионная модель:

$y = f(x, b) + \varepsilon$

b - параметры модели

ε - случайная ошибка модели

$f(x, b) = b_0 + b_1x_1 + b_2x_2 + ... + b_kx_k$, Где b_j - параметры (коэффициенты) модели, x_j - регрессоры (факторы модели), k - кол-во факторов модели.

Матричное представление:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{pmatrix}$$

$$X = \begin{pmatrix} x_{11} & x_{12} & ... & x_{1k} \\ x_{21} & x_{22} & ... & x_{2k} \\ ... & & & \\ x_{n1} & x_{n2} & ... & x_{nk} \end{pmatrix}$$ - факторы модели.

$$\varepsilon = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ ... \\ \varepsilon_n \end{pmatrix}$$ - вектор ошибок.

$$b = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_k \end{pmatrix}$$ - параматры модели.

Тогда наша реграссионная модель будет иметь формулу: $y = Xb + \varepsilon$

Для решения задачи: $y = b_0 + b_1 * x$. Будем говорить, что у нас есть x_0.

$y = b_0 * x_0 + b_1 * x_1 = b_0 * 1 + b_1 * x_1$

1.2 Производная линейной регрессии

Наша задача состоит в том чтобы найти такой вектор b, чтобы $\varepsilon = (Xb - y)^2 \to min$

$(Xb - y)^2 = (Xb - y)^T(Xb - y) = (Xb)^TXb - y^TXb - (Xb)^Ty + y^Ty$

$y^TXb = (Xb)^Ty = b^TX^Ty$

$(Xb)^TXb = b^TX^TXb$

$\varepsilon = b^TX^TXb - 2b^TX^Ty + y^Ty$

$\frac{d(b^TX^TXb - 2b^TX^Ty + y^Ty)}{db}$

$(b^TX^TXb)' - (2b^TX^Ty)' + (y^Ty)' = 0$

$2X^TXb - 2X^Ty = 0$

$X^TXb = X^Ty$

Значит $b^{[i+1]} = b^{[i]} - \alpha\nabla F(b^{[i]}) = b^{[i]} - \alpha(X^TXb^{[i]} - X^Ty)$

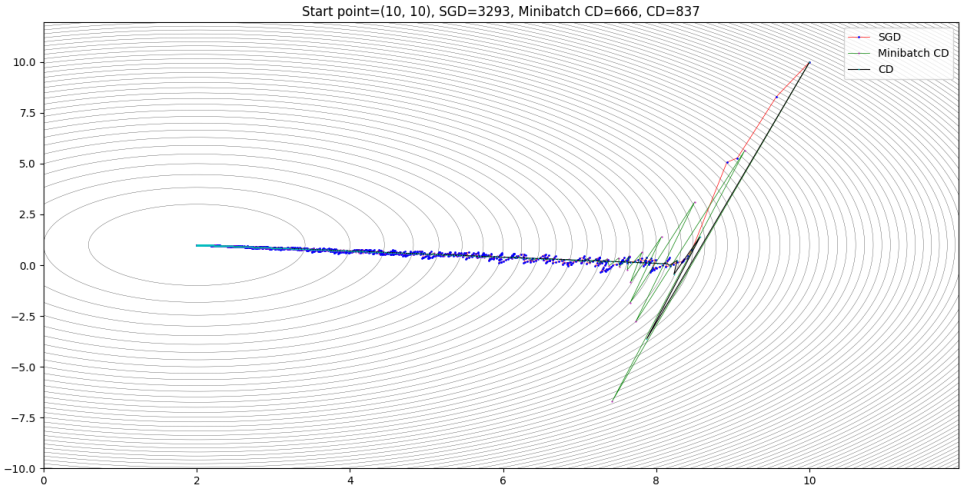
1.2.1 Исследования

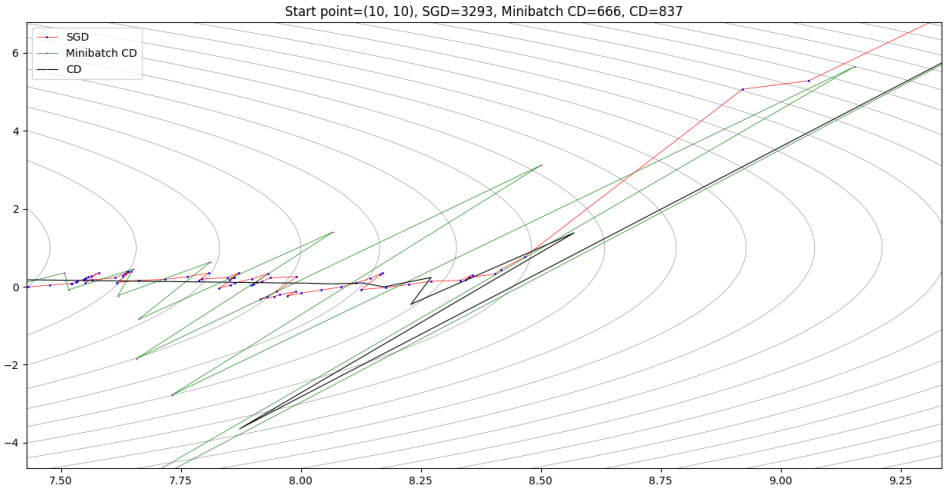
Для иследования возьмем 100 точек. И прямую $y = x + 2$. Не будем добавлять шума для обьективности исследований. Стартовая точка для всех *batch* будет $(10, 10)$

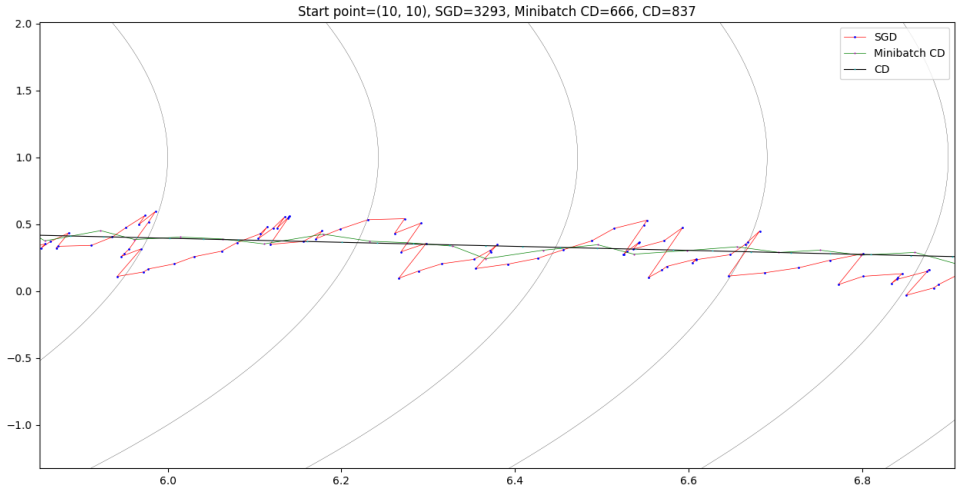
Для SGD $alpha = 0.01$

Для $Minibatch\ GD$ $alpha = 0.001$

Для GD возьмем наилучшую из экспериментов $alpha = 0.0004$







Можно заметить, что SGD метод довольно рандомный. он ведет себя очень не предсказуемо. Идет то вверх то вниз. Так же требует довольно много итераций. Но в отличие от остальных размров *batch*. Одна итерация стоит довольно дёшево.

$Minibatch\ CD$ уже менее заметна эта проблема. Хотя рандомность присутствует, но все равно график лежит плотнее к CD, и кол-во итераций требуется наименьшее.

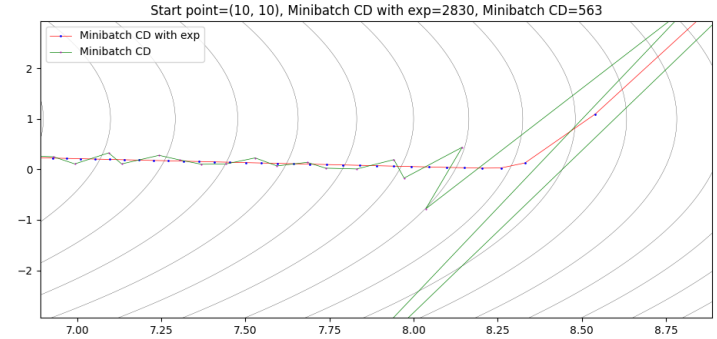
CD идет равнее всех. Но один его шаг стоит дороже всех. Хоть он и является самым точным, но так же он является не самым эффективным.

2 Экспоненциальная функция изменения шага

Сейчас наш градиентный шаг равен $b^{[i+1]} = b^{[i]} - \alpha\nabla F(b^{[i]})$ Давайте попробуем заменить константную α на экспоненциальную $\alpha^{[i+1]} = \alpha^{[i]} * e^c$. c - константа, $c < 0$

Рассмотрим применение экспоненциальной $alpha$ с $Minibatch\ CD$. Возмём все те же значения из предыдущего исследования.

$exp\_alpha = 0.0003, c = -0.001$



Метод стал вести себя более стабильно. Но для этого пришлось подобрать подходящие $alpha, c$, что не очень легко. Так же потребовалось больше итераций для нахождения нужной точки.

3 Варианции

Базовый градиентный спуск:

$$b^{[i+1]} = b^{[i]} - \alpha\nabla F(b^{[i]})$$

3.1 Nesterov

$$u^{[i+1]} = \gamma u^{[i]} + \alpha \nabla F(b^{[i]} - \gamma u^{[i]})$$
$$b^{[i+1]} = b^{[i]} - u^{[i+1]}$$

пример фрема который исследуется

points	alpha	iters	result point
10	0.000100	1001	[5.3, 0.5]
10	0.000250	1001	[3.2, 0.8]
10	0.000300	1001	[2.9, 0.9]
10	0.000400	1001	[2.4, 0.9]
10	0.000500	1001	[2.2, 1.0]
10	0.000700	1001	[2.1, 1.0]
10	0.001000	1001	[2.0, 1.0]

выбраннные α для анализа

$$\alpha = [0.0001, 0.00025, 0.0003, 0.0004, 0.0005, 0.0007, 0.001]$$

то сколько раз метод не сошёлся на для каждой α всего тестов **28**

alpha	result err
0.000100	16
0.000250	8
0.000300	8
0.000400	8
0.000500	8
0.000700	12
0.001000	20

data with runs

count	alpha	run
40	0.000500	926.750000
	0.000700	658.000000
	0.001000	452.250000
70	0.000250	725.750000
	0.000300	602.000000
	0.000400	442.000000
	0.000500	348.750000
	0.000700	240.250000
	0.001000	154.250000
100	0.000250	598.750000
	0.000300	493.250000
	0.000400	363.500000
	0.000500	283.250000
	0.000700	192.750000
130	0.000100	965.500000
	0.000250	366.250000
	0.000300	300.250000
	0.000400	216.000000
	0.000500	164.000000
	0.000700	177.500000
160	0.000100	782.250000
	0.000250	292.750000
	0.000300	238.500000
	0.000400	168.500000
	0.000500	128.750000
190	0.000100	676.500000
	0.000250	250.000000
	0.000300	200.500000
	0.000400	138.250000

3.2 Momentum

$$u^{[i+1]} = \gamma u^{[i]} + \alpha \nabla F(b^{[i]})$$
$$b^{[i+1]} = b^{[i]} - u^{[i+1]}$$

взятые α

$$\alpha = [0.0001, 0.0005, 0.00075, 0.001, 0.00125, 0.0025, 0.005, 0.02]$$

пример фрейма

points	alpha	iters	result point	fail
10	0.000100	1001	[6.2, 0.3]	20/28
10	0.000500	1001	[2.8, 0.9]	4/28
10	0.000750	1001	[2.3, 1.0]	4/28
10	0.001000	1001	[2.1, 1.0]	4/28
10	0.001250	1001	[2.0, 1.0]	4/28
10	0.002500	623	[2.0, 1.0]	4/28
10	0.005000	305	[2.0, 1.0]	4/28
10	0.020000	1001	[inf, inf]	4/28

alpha	fail
0.000100	20
0.000500	4
0.000750	4
0.001000	4
0.001250	8
0.002500	16
0.005000	24
0.020000	28

количество итераций для каждого количества точек и α

count	alpha	run
10	0.002500	667.8
	0.005000	322.8
40	0.000500	726.5
	0.000750	476.2
	0.001000	343.5
	0.001250	273.0
	0.002500	103.2
70	0.000500	355.8
	0.000750	219.8
	0.001000	150.8
	0.001250	105.0
	0.002500	101.0
100	0.000500	208.2
	0.000750	120.2
	0.001000	94.5
	0.001250	92.8
130	0.000500	196.2
	0.000750	111.5
	0.001000	100.0
	0.001250	103.0
160	0.000100	788.0
	0.000500	115.5
	0.000750	92.0
	0.001000	90.2
	0.001250	122.2
190	0.000100	694.0
	0.000500	89.8
	0.000750	90.2
	0.001000	114.5

3.3 Adagrad

$$\nabla F(b^{[i]}) = (g_0^{[i]}, ..., g_k^{[i]})$$

$$G_j^{[i+1]} = G_j^{[i]} + (g_j^{[i]})^2$$

$$\alpha_j^{[i]} = \frac{\alpha}{\sqrt{G^{[i]} + \varepsilon}}$$

$$b^{[i+1]} = b^{[i]} - \alpha^{[i]} \otimes \nabla F(b^{[i]})$$

$$\varepsilon = 0.1^8$$

пример фрейма который исследуется

points	alpha	iters	result point	fail
10	1	1001	[1.6, 1.1]	28/28
10	2	1001	[2.0, 1.0]	25/28
10	50	154	[2.0, 1.0]	0/28
10	100	185	[2.0, 1.0]	0/28
10	1000000	135	[2.0, 1.0]	0/28

для испытания кода, бралось 4 стартовые точки, (± 10, ± 10), ниже представлена таблица для какихα метод не сошёлся

решено было взять такие α так в силу того что *adagrad* использует в себе корретирование α и при меньшем алгоритм не успевает сойтись

$$\alpha = [1, 2, 50, 100, 1000000]$$

далее удалим те методы которые не сошлись из статистики

среднее значение итераций, для каждого количества точек и выбрынных α

count points	alpha	iters
10	50	206.2
	100	213.5
	1000000	245.0
40	50	224.5
	100	247.5
	1000000	234.2
70	50	179.2
	100	193.0
	1000000	201.0
100	2	874.0
	50	165.2
	100	158.8
	1000000	170.2
130	2	834.0
	50	151.5
	100	170.8
	1000000	182.0
160	50	197.5
	100	207.8
	1000000	230.2
190	2	851.0
	50	156.8
	100	164.2
	1000000	170.8

3.4 RMSProp

$$\nabla F(b^{[i]}) = (g_0^{[i]}, ..., g_k^{[i]})$$
$$G_j^{[i+1]} = G_j^{[i]} + (g_j^{[i]})^2 - (g_j^{[i-W]})^2$$
$$\alpha_j^{[i]} = \frac{\alpha}{\sqrt{\frac{1}{W}G^{[i]} + \varepsilon}}$$
$$b^{[i+1]} = b^{[i]} - \alpha^{[i]} \otimes \nabla F(b^{[i]})$$

Отличие от Adagrad, в том что суммируем не все, атолько W последних

выбранные α

$$\alpha = [0.01, 0.02, 0.05, 0.1, 0.25, 0.5, 1, 5, 20]$$

пример фрейма

points	alpha	iters	result point	fail
10	0.010000	1001	[1.7, 1.1]	28/28
10	0.020000	595	[2.0, 1.0]	14/28
10	0.050000	249	[2.0, 1.0]	0/28
10	0.100000	147	[2.0, 1.0]	0/28
10	0.250000	103	[2.0, 1.0]	0/28
10	0.500000	112	[2.0, 1.0]	0/28
10	1.000000	389	[2.0, 1.0]	8/28
10	5.000000	1001	[1.7, 1.0]	18/28
10	20.000000	1001	[5.0, -1.0]	28/28

для таких α не сошёлся

alpha	fail
0.010000	28
0.020000	14
0.050000	0
0.100000	0
0.250000	0
0.500000	0
1.000000	8
5.000000	18
20.000000	28

среднее число для каждого числа точек и α

count	alpha	iters
10	0.020000	615.5
	0.050000	427.8
	0.100000	270.0
	0.250000	324.2
	0.500000	467.5
	1.000000	414.2
40	0.020000	598.5
	0.050000	419.8
	0.100000	213.8
	0.250000	224.8
	0.500000	334.8
	1.000000	400.5
70	0.020000	594.0
	0.050000	422.0
	0.100000	250.8
	0.250000	214.5
	0.500000	140.8
	1.000000	209.5
	5.000000	259.0
100	0.020000	611.5
	0.050000	426.2
	0.100000	244.0
	0.250000	144.8
	0.500000	347.5
	1.000000	198.0
	5.000000	238.0
130	0.020000	605.0
	0.050000	446.2
	0.100000	251.8
	0.250000	204.0
	0.500000	313.8
	1.000000	180.5
	5.000000	354.0
160	0.020000	622.0
	0.050000	434.0
	0.100000	269.0
	0.250000	183.2
	0.500000	310.5
	1.000000	431.3
	5.000000	855.0
190	0.020000	604.0
	0.050000	456.5
	0.100000	240.2
	0.250000	240.8
	0.500000	220.2
	1.000000	613.5
	5.000000	782.5

3.5 Adam

$$m^{[i+1]} = \beta_1 m^{[i]} + (1 - \beta_1) \cdot \nabla F(b^{[i]})$$

$$u^{[i+1]} = \beta_2 u^{[i]} + (1 - \beta_2) \cdot \nabla (F(b^{[i]}))^2$$

$$b^{[i+1]} = b^{[i]} - \frac{\alpha \cdot m^{[i]}}{\sqrt{u^{[i]}} + \varepsilon}$$

$$\beta_1 = 0.9, \beta_2 = 0.9$$

пример фрейма

points	alpha	iters	result point	fail
10	0.010000	1001	[1.4, 1.1]	28/28
10	0.025000	537	[2.0, 1.0]	13/28
10	0.050000	397	[2.0, 1.0]	0/28
10	0.050000	355	[2.0, 1.0]	0/28
10	0.075000	311	[2.0, 1.0]	0/28
10	0.100000	331	[2.0, 1.0]	4/28
10	0.200000	253	[2.0, 1.0]	19/28
10	0.500000	993	[2.0, 1.0]	24/28
10	1.000000	1001	[2.0, 1.0]	28/28

решено было взять такие α так в силу того что *adagrad* использует в себе корретирование α и при меньшем алгоритм не успевает сойтись

$$\alpha = [0.01, 0.025, 0.05, 0.075, 0.1, 0.2, 0.5, 1, 5]$$

для испытания кода, бралось 4 стартовые точки, ($\pm$10, $\pm$10), ниже представлена таблица для каких α метод не сошёлся

alpha	fail
0.010	28
0.025	13
0.050	0
0.075	0
0.100	0
0.200	4
0.500	19
1.000	24
5.000	28

среднее значение итераций, для каждого количества точек и выбрынных α

count	alpha	iters
10	0.025000	576.0
	0.050000	563.1
	0.075000	442.8
	0.100000	386.0
	0.200000	309.0
	0.500000	608.7
40	0.025000	692.3
	0.050000	486.1
	0.075000	397.8
	0.100000	439.5
	0.200000	352.7
	1.000000	896.5
70	0.025000	576.0
	0.050000	517.2
	0.075000	481.0
	0.100000	410.0
	0.200000	520.2
	0.500000	806.0
100	0.025000	593.5
	0.050000	527.1
	0.075000	438.2
	0.100000	452.5
	0.200000	352.7
	0.500000	428.5
130	0.025000	583.0
	0.050000	518.4
	0.075000	477.0
	0.100000	359.0
	0.200000	414.7
	0.500000	729.5
	1.000000	443.0
160	0.025000	595.5
	0.050000	531.2
	0.075000	432.0
	0.100000	435.8
	0.200000	505.8
	0.500000	808.0
190	0.025000	604.0
	0.050000	511.6
	0.075000	474.5
	0.100000	428.5
	0.200000	600.0
	1.000000	346.0

4 Исследования

math oper per iter

	name	math per oper
0	adagrad	12
1	adam	21
2	momentum	8
3	rms	15
4	nesterov	10

min iters per count

count	alpha	name	iters
10	0.005000	momentum	128.000000
10	50.000000	adagrad	206.250000
10	0.100000	rms	270.000000
10	0.200000	adam	309.000000
10	0.000100	nesterov	1001.000000
40	0.002500	momentum	99.750000
40	0.100000	rms	213.750000
40	50.000000	adagrad	224.500000
40	0.200000	adam	352.666667
40	0.001000	nesterov	452.250000
70	0.500000	rms	140.750000
70	0.001000	nesterov	154.250000
70	0.001250	momentum	162.500000
70	50.000000	adagrad	179.250000
70	0.100000	adam	410.000000
100	0.001000	momentum	93.250000
100	0.250000	rms	144.750000
100	100.000000	adagrad	158.750000
100	0.000700	nesterov	192.750000
100	0.200000	adam	352.666667
130	0.001000	momentum	92.250000
130	50.000000	adagrad	151.500000
130	0.000500	nesterov	164.000000
130	1.000000	rms	180.500000
130	0.100000	adam	359.000000
160	0.000750	momentum	93.250000
160	0.000500	nesterov	128.750000
160	0.250000	rms	183.250000
160	50.000000	adagrad	197.500000
160	0.075000	adam	432.000000
190	0.000750	momentum	97.750000
190	0.000400	nesterov	138.250000
190	50.000000	adagrad	156.750000
190	0.500000	rms	220.250000
190	1.000000	adam	346.000000

Momentum - модификация градиентного спуска, которая добавляет импульс, учитывающий предыдущие градиенты. Это позволяет более быстро сходиться к минимуму функции и преодолевать локальные минимумы. Однако, этот метод может привести к осцилляциям и увеличению ошибки, если шаг слишком большой.

Nesterov - это модификация Momentum, которая учитывает будущий градиент, что позволяет более точно определить направление движения. Это позволяет более быстро сходиться к минимуму и уменьшить количество осцилляций.

RMSprop - это метод, который адаптивно изменяет скорость обучения в зависимости от градиента. Это позволяет более быстро сходиться к минимуму и уменьшить количество осцилляций. Однако, этот метод может привести к проблеме "затухания градиента когда скорость обучения становится слишком маленькой.

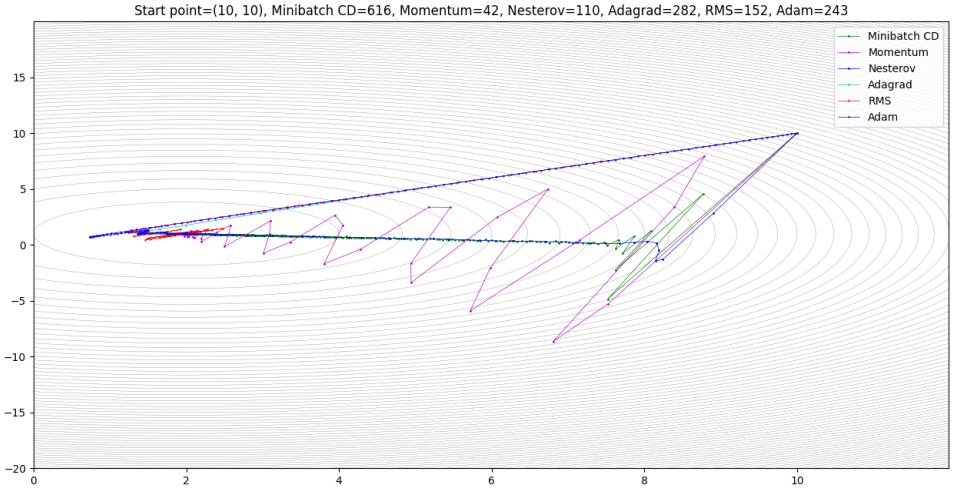
Adagrad - это метод, который адаптивно изменяет скорость обучения в зависимости от градиента и частоты появления каждого параметра. Это позволяет более эффективно использовать скорость обучения и уменьшить количество осцилляций. Однако, этот метод может привести к проблеме "затухания градиента когда скорость обучения становится слишком маленькой.

Adam - это метод, который комбинирует Momentum и RMSprop. Он адаптивно изменяет скорость обучения и добавляет импульс, учитывающий предыдущие градиенты. Это позволяет более быстро сходиться к минимуму и уменьшить количество осцилляций. Adam является одним из наиболее эффективных методов оптимизации в машинном обучении.

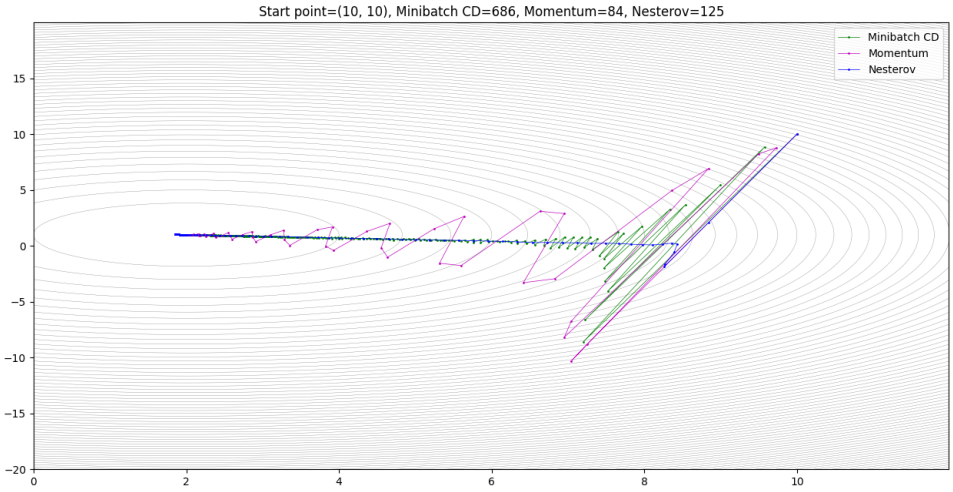
Таким образом, каждый из этих методов имеет свои достоинства и недостатки, и выбор конкретного метода зависит от конкретной задачи и данных.

5 Графики

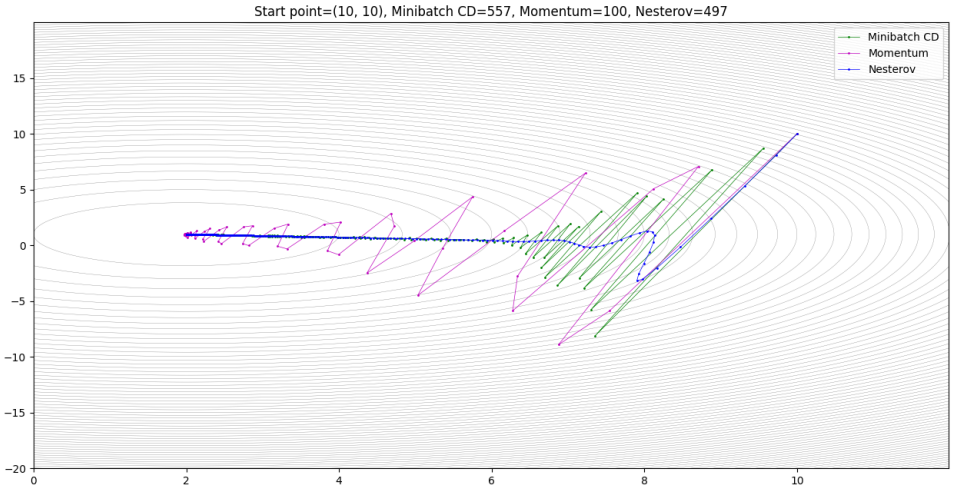
Давайте посмотрим, какие у методов траектории. Возьмем $y = x+2$, 100 точек и размер $batch = 50$



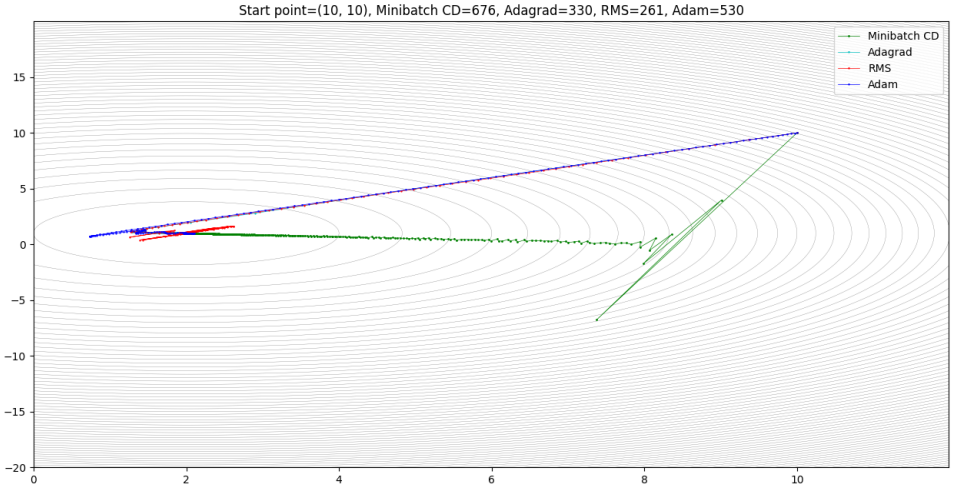
Можно выделить два вида графиков. Первый: $Minibatch\ CD$, $Momentum$, $Nesterov$



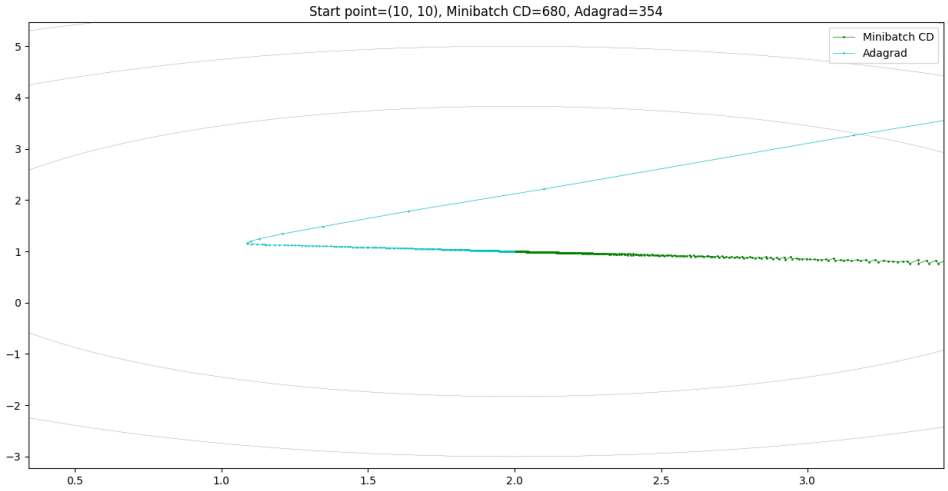
$Momentum$ довольно быстро находит точку, идет как вверх, так и вниз. Задействует больше области, чем другие методы. А $Nesterov$ сразу находит нужную траекторию, и сглаживает выпады $Momentum$. Если установить $alpha$ у $Momentum$ побольше, то можно увидеть это замедление.



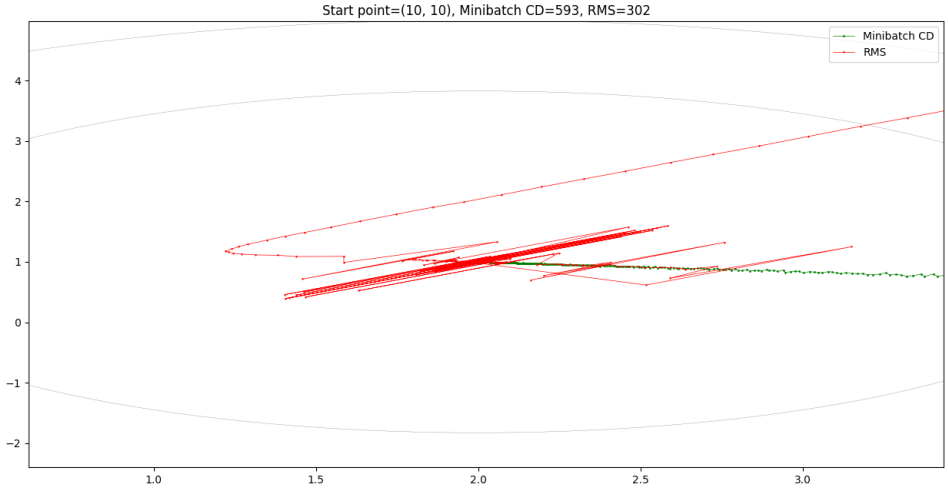
И второй тип графиков: $Adagrad$, RMS, $Adam$. Они идут по другой траектории. они не прижимаются к какой-то оси. Они идут пропорционально к точки.



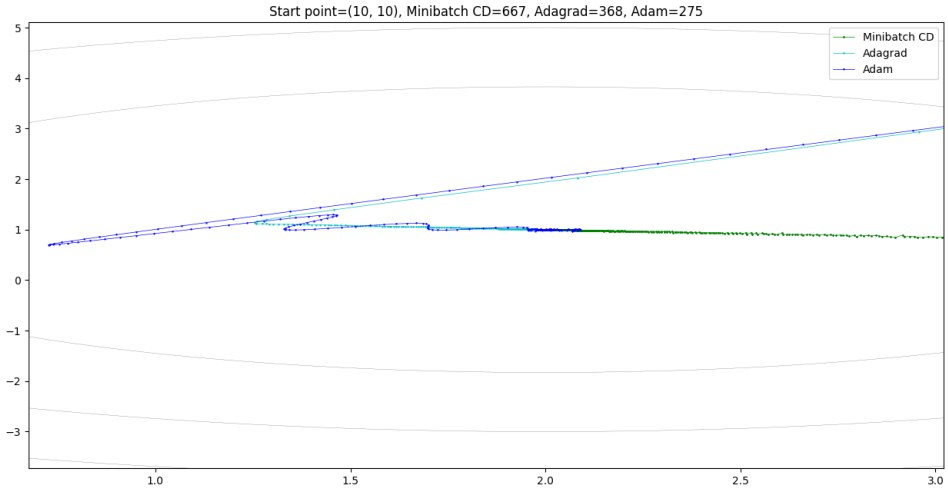
В чем же их отличия?



Adagrad быстро находит область рядом с нужной точкой. А потом медленно начинает уменьшать *eps*. Ведь чем больше у него итераций, тем меньше будет *alpha*.



RMS хранит не все значения *alpha* и когда начинает двигаться "обратно то он начинает сходиться не очень хорошо, довольно рандомно. Скорее всего это вызвано из-за того, что *batch* установлен на половину. И если бы *batch* равнялся n, то точность была бы выше.



Adam убирает рандомность у RMS и сглаживает график, что выглядит гораздо стабильнее.

Вывод

Таким образом, каждый из этих методов имеет свои достоинства и недостатки, и выбор конкретного метода зависит от конкретной задачи и данных.