1 Стохастический градиентный спуск (SGD)

Стохастический градиентный спуск (SGD) — итерационный метод для оптимизации целевой функции. Его отличия от обычного градиентного спуска заключается в том, что он заменяет реальный градиент, вычисленный из полного набора данных, оценкой, вычисленной из случайно выбранного подмножества данных. Логчино, что SGD будет использовать меньше вычислительных ресурсов и тратить меньше времени на каждую итерацию. С другой стороны, уменьшается скорость сходимости, так как точность градиента на каждой итерации ниже из-за использования лишь подмножества данных. Метод рассматривает задачу минимизации целевой функции, имеющей форму суммы.

Приведем пример (который, собственно, будет рассматриваться в данной работе). Цель - приблизить прямую $y = a * x_1 + b * x_2 + c$ с тренировочным набором данных $(\vec{x}_1, \vec{x}_2, ..., \vec{x}_n)$ и реальные значения на этих данных $(\grave{y}_1, \grave{y}_2, ..., \grave{y}_n)$. За функцию ошибки возьмем среднеквадратичную функцию потерь: $\frac{1}{n}\sum_i^n (\grave{y}_i - y_i)^2$, где y_i - это значение прямой, вычисленной при текущих коэффициентах a, b, c. Собственно, эту функцию потерь мы и будем стараться минимизировать. В обычном градиентном спуске мы бы считали градиент во всех n точках. Но в SGD у нас есть, так называемый, батч (подмножество тренировочного набора), с которым мы будем работать. И его размер может быть любым, от 1 до n.

Тогда давайте для удобства коэффициенты при признаках (аргументах) прямой обозначим за вектор $coefs$, батч обозначим за X_{batch} - набор векторов тренировочных признаков. Значения на этих векторах обозначим за y_{batch}. Полученные значения при текущих коэффициентах обозначим за y_{exec}.

Тогда оптимизировать коэффициенты будем следующим образом ($learning\_rate$ - шаг градиентного спуска; идем по прежнему вдоль антиградиента):

$$coefs := coefs - \frac{learning\_rate}{n}((\grave{y}_{batch} - y_{exec})^2)'_{coefs}$$
$$c := c - \frac{learning\_rate}{n}((\grave{y}_{batch} - y_{exec})^2)'_{c}$$

Подставляем значение y_{exec}:

$$coefs := coefs - \frac{learning\_rate}{n}((\grave{y}_{batch} - (X_{batch} * coefs + c))^2)'_{coefs}$$
$$c := c - \frac{learning\_rate}{n}((\grave{y}_{batch} - (X_{batch} * coefs + c))^2)'_{c}$$

Тогда получаем:

$$coefs := coefs - \frac{learning\_rate}{n}(-2X_{batch} * (\grave{y}_{batch} - y_{exec}))$$
$$c := c - \frac{learning\_rate}{n}(-2(\grave{y}_{batch} - y_{exec}))$$

Именно таким образом мы будем изменять наши коэффициенты. Заметим, что на каждой итерации мы будем вычислять функцию и градиент функции потерь не во всех тренировочных точках, как в обычном градиентном спуске, а только в его подмножестве размера $batch\_size$. К тому же, будем выбирать

участвующие точки рандомным образом.

2 Собственная реализация SGD для решения линейной регрессии

Напишем программу, которая будет перебирать разные $batch\_size$ и $learning\_rate\_scheduling$ и вызывать SGD с этими параметрами.

Перед запуском SGD будем генерировать тренировочные данные (размера 1000), а потом использовать их часть в самой функции SGD. Так же будем генерировать и тестовые данные, на которых будем тестировать итоговые коэффициенты и считать среднеквадратичную ошибку (mse).

В качестве реальных коэффициентов возьмем: $a = 3, b = 8, c = 50$. А начальные коэффициенты будем генерировать случайным образом.

Функция изменения шага будет выглядить следующим образом:

```
# меняем шаг
if learning_rate_scheduling == "exponential":
    learning_rate *= 0.995
elif learning_rate_scheduling == "stepwise":
    if i % 100 == 0:
        learning_rate *= 0.6
```

То есть, при экспоненциальной изменении изменяем шаг на каждой итерации, а при ступенчатой - на каждой сотой итерации.

Вот, какие результаты получились:

batch size	learning rate	learning rate sheduling	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
1	0.01	none	500	0.0686	0.0567	0.5425	0.0998	36.676	22534
1	0.01	exponential	500	0.0580	0.0709	0.7403	1.23548	39.0157	19723
1	0.01	stepwise	500	0.1636	0.0658	2.42890	7.32698	34.9988	19502
100	0.01	none	500	0.0072	0.0078	0.4895	0.0851	37.6703	25332
100	0.01	exponential	500	0.0334	0.017	0.7219	1.26054	37.974	25332
100	0.01	stepwise	500	0.0656	0.0053	2.474	7.630884	38.024	25332
700	0.01	none	500	0.0063	0.0083	0.4896	0.08519	61.9871	49332
700	0.01	exponential	500	0.0289	0.0083	0.7198	1.25511	82.112	49332
700	0.01	stepwise	500	0.0660	0.0018	2.47346	7.61020	56.956	49332
1000	0.01	none	500	0.0056	0.0108	0.4876	0.07557	87.218	79332
1000	0.01	exponential	500	0.0288	0.0054	0.7186	1.34217	69.004	79332
1000	0.01	stepwise	500	0.0643	0.0008	2.47438	7.62747	73.877	79332

Что мы можем заметить? Коэффициенты a и b всегда считаются с довллно хорошей точностью, чего не скажешь о коэффициенте c. Его погрешность достигает 2 единиц при использовании ступенчатой функции изменения шага. Возможно, это происходит из-за того, что коэффициент c имеет большое значение относительно двух других. А ступенчатая функция изменения шага не позволяет алгоритму дойти до нужного значения. Попробуем решить эту проблему, умножая шаг на 0.8 вместо 0.6 в ступенчатой функции или увеличивая количество итераций:

batch size	learning rate	learning rate sheduling	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
1	0.01	stepwise (0.8)	500	0.01474	0.00410	0.3162	0.14423	37.5089	19502
100	0.01	stepwise (0.8)	500	0.00509	0.00642	0.3071	0.08969	73.340	25332
700	0.01	stepwise (0.8)	500	0.00685	0.00628	0.31056	0.08916	56.9472	61332
1000	0.01	stepwise (0.8)	500	0.00703	0.005876	0.3108	0.08908	61.9945	79332
1	0.01	stepwise	1500	0.1451	0.09054	1.53023	3.98740	141.499	51888
100	0.01	stepwise	1500	0.0983	0.0867	1.57892	4.17870	124.463	57828
700	0.01	stepwise	1500	0.1017	0.08402	1.58266	4.1879	199	93828
1000	0.01	stepwise	1500	0.09860	0.08344	1.58116	4.1816	188.062	111828

Видим, что уменьшение изменения шага на каждой итерации действительно увеличило точность вычислений. Время и память остались примерно теми же, что ожидаемо. Однако увеличение количества итераций не только не справилось с проблемой точности, так еще и потребовало больше памяти и времени. В принципе, это логично, так как если алгоритм раньше не успевал дойти до нужного значения, и при этом шаг уже стал маленький, увеличение количества итераций в несколько раз не поможет решить проблему, потому что шаг уже маленький, так еще и продолжит уменьшаться. В итоге, алгоритм будет топтаться на месте, тратя время и память.

Что еще можем увидеть из полученных данных. При увеличении $batch\_size$ объем памяти увеличивается. Это объясняется тем, что при больших размерах батча на каждой итерации мы используем больше данных и, соответственно, больше памяти. Время работы программы так же увеличивается по той же причине. На каждой итерации происходит больше арифметических операций, что снижает скорость работы программы.

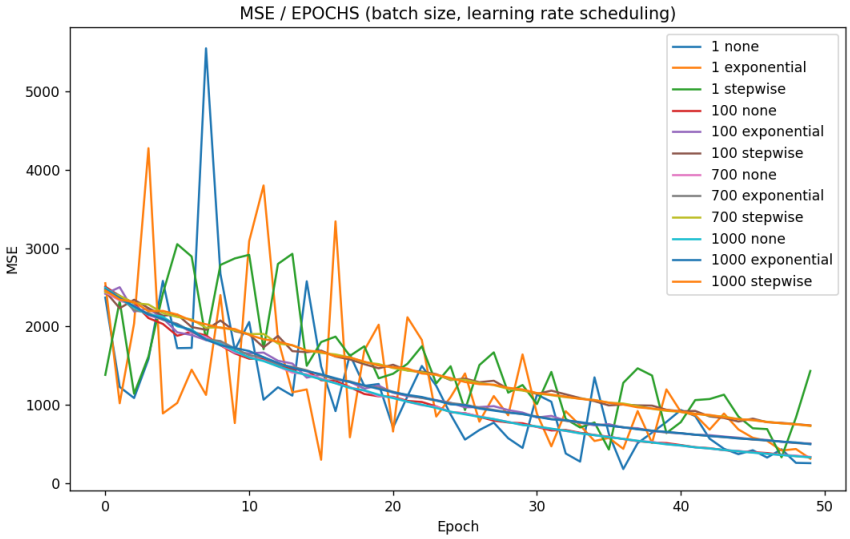
Но что мы можем сказать про сравнение функций изменения шага? Неужели без изменения шага алгоритм всегда работает не хуже, чем с изменением? Подумаем, для чего нам нужна функция изменения шага. Для того, чтобы на начальных этапах скорость сходимости была высокая, а на конечных этапах не расходиться в силу уменьшения шага. Так давайте посмотрим на работу алгоритма при большом начальном шаге. Например, 0.5.

batch size	learning rate	learning rate sheduling	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
1	0.5	none	500	overflow	overflow	overflow	overflow	45.065	51936
1	0.5	exponential	500	overflow	overflow	overflow	overflow	43.773	19739
1	0.5	stepwise	500	overflow	overflow	overflow	overflow	43.984	19502
100	0.5	none	500	overflow	overflow	overflow	overflow	42.528	25332
100	0.5	exponential	500	0.00691	0.00293	0.4977	0.10812	38	25332
100	0.5	stepwise	500	0.0099	0.0128	0.4961	0.111	40.010	25332
700	0.5	none	500	overflow	overflow	overflow	overflow	63.5850	382564
700	0.5	exponential	500	0.0087	0.0038	0.4925	0.10902	52.489	61332
700	0.5	stepwise	500	0.0041	0.0023	0.4941	0.1086	45.9971	61332
1000	0.5	none	500	overflow	overflow	overflow	overflow	49.527	79332
1000	0.5	exponential	500	0.00624	0.00115	0.49415	0.1080	62.0009	79332
1000	0.5	stepwise	500	0.0078	0.003284	0.4929	0.1088	52.528	79332

Итак, что мы видим. Во-первых, для размера батча, равному 1, все вариации переполняются. Но с увеличением размера батча вариации с

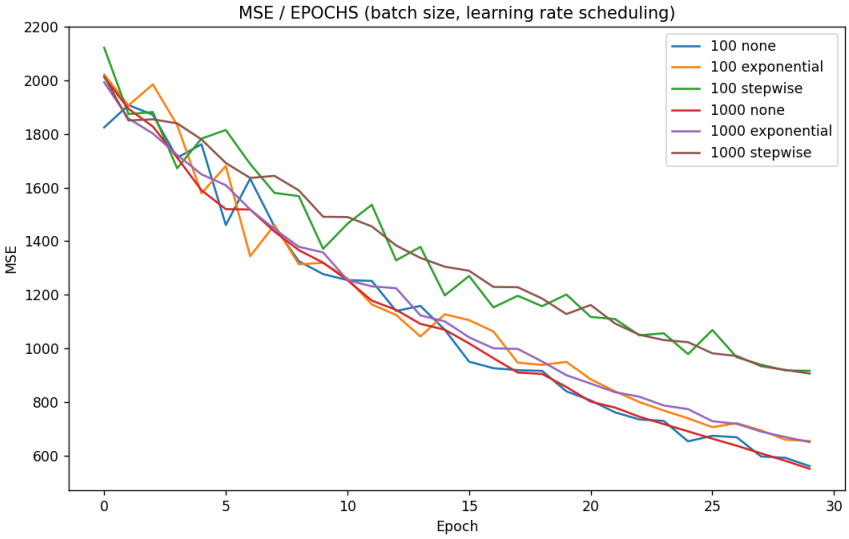
изменением шага начинают работать правильно, так еще и выдавать хорошую погрешность. В отличие от постоянного шага, который не может отработать хорошо. В этом и есть преимущество функций изменения шага. При изначальном маленьком шаге он продолжает уменьшаться и в итоге не доходит до нужной точки. Зато при большом начальном шаге алгоритм работает хорошо и в большинстве случаев не переполняется. Для маленького размера батча, скорее всего, не хватило точности вычислений, поэтому все вариации ушли в бесконечность, не успев уменьшить шаг.

Теперь давайте посмотрим на графики среднеквадратичной ошибки от номера итерации.



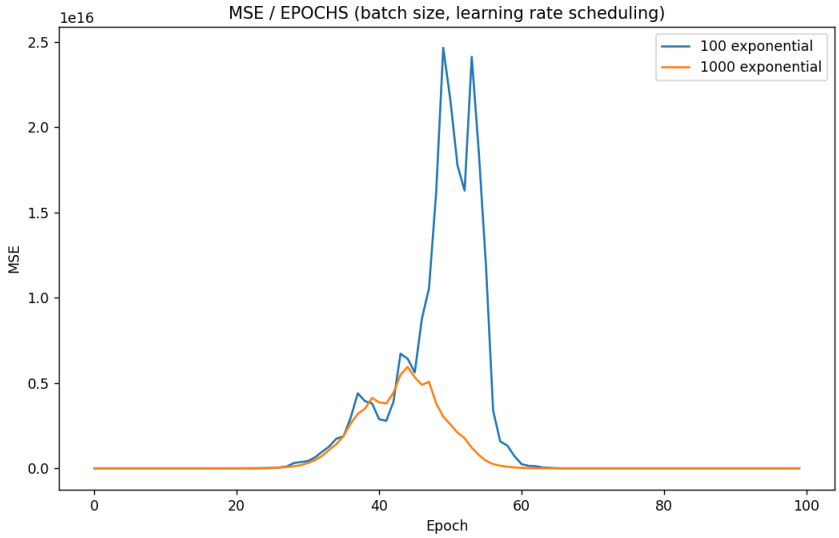
Видно, что при размере батча, равным 1, программа ведет себе неадекватно. Однако алгоритм все равно сходится к нужному значению и постепенно минимизирует среднеквадратичную ошибку. Вероятно, это происходит из-за того, что на каждой итерации значения сравниваются только с одной точкой, что дает меньшую пользу и большую погрешность. Однако алгоритм все равно сходится, как и ожидалось.

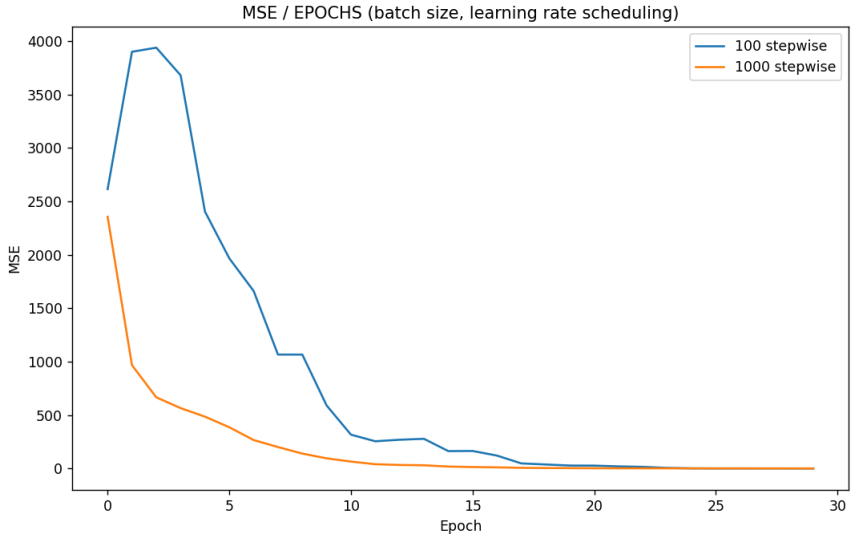
Давайте посмотрим на график без размера батча 1.



Как ожидалось из полученных результатов, все алгоритмы сходятся, но при шаге 0.01 точнее оказывается алгоритм с константным шагом.

Теперь посмотрим на графики с размером шага 0.5 (не будем смотреть на неизменяемый шаг, так как он всегда переполняется).





Можно заметить разное поведение данных графиков. В случае экспоненциальной функции у алгоритма сначала хорошая погрешность, затем он как будто начинает расходиться, но потом опять возвращается к искомой точке благодаря уменьшению шага. В случае же со ступенчатой функцией, она сразу имеет хорошую погрешность и постепенно, без скачков, идет к искомой точке. С самого начала, конечно, график так же может уйти вверх, но со временем вернется в нужную окрестность.

3 Библиотечная реализация SGD для решения линейной регрессии

Будем использовать библиотеку tensorflow и брать оттуда все виды оптимизации SGD.

Расскажем про модификации SGD

- Momentum

  Стохастический градиентный спуск с импульсом запоминает изменение коэффициентов на каждой итерации и определяет следующее изменение в виде линейной комбинации градиента и предыдущего изменения.

- Nesterov

  Отличается от Momentum тем, что в классическом импульсе мы сначала корректируем свою скорость, а затем делаем большой шаг в соответствии с этой скоростью (и затем повторяете), но в импульсе Нестерова мы сначала делаем шаг в направлении скорости, а затем вносим коррекцию в вектор скорости на основе нового местоположения.

- AdaGrad

Является модификацией стохастического алгоритма градиентного спуска с отдельной для каждого параметра скоростью обучения. Эта стратегия увеличивает скорость сходимости по сравнению со стандартным стохастическим методом градиентного спуска в условиях, когда данные редки и соответствующие параметры более информативны. Большие изменения параметра ослабляются, в то время как параметры, получающие малые изменения получают более высокие скорости обучения

- RMSProp

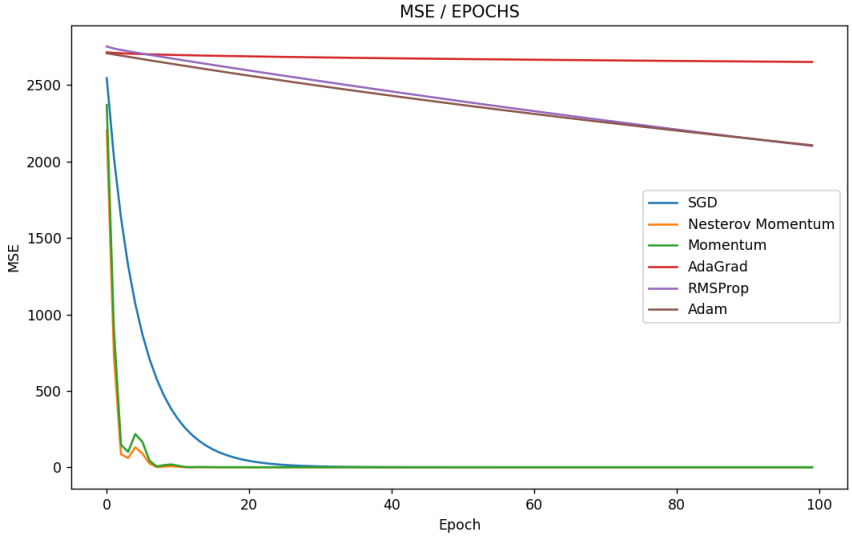
  Метод, в котором скорость обучения настраивается для каждого параметра. Идея заключается в делении скорости обучения для весов на скользящие средние значения недавних градиентов для этого веса

- Adam

  Это обновление оптимизатора RMSProp. В этом оптимизационном алгоритме используются скользящие средние как градиентов, так и вторых моментов градиентов.

Посмотрим на первые результаты для $batch\_size = 200$

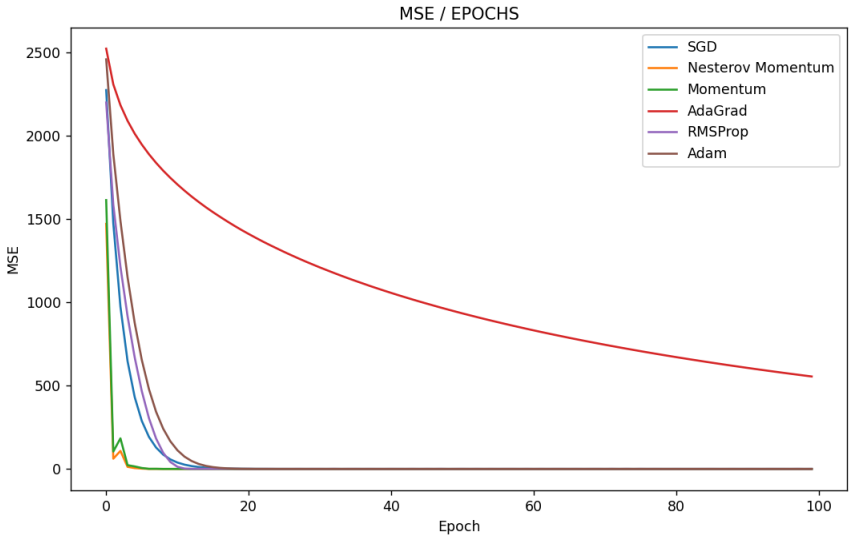
batch size	learning rate	modification	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
200	0.01	SGD	100	0.0045	0.0016	0.4884	0.08739	5094	3017897
200	0.01	NESTEROV	100	0.0045	0.00053	0.489	0.08742	10992	1225114
200	0.01	MOMENTUM	100	0.0026	0.00010	0.4910	0.08763	14023	1092137
200	0.01	ADAGARD	100	2.849	6.841	49.567	2102	12941	1061328
200	0.01	RMSPROP	100	0.4654	3.82424	44.917	1844	11523	1115714
200	0.01	ADAM	100	0.856	3.138	45.108	1882	7559	1394675



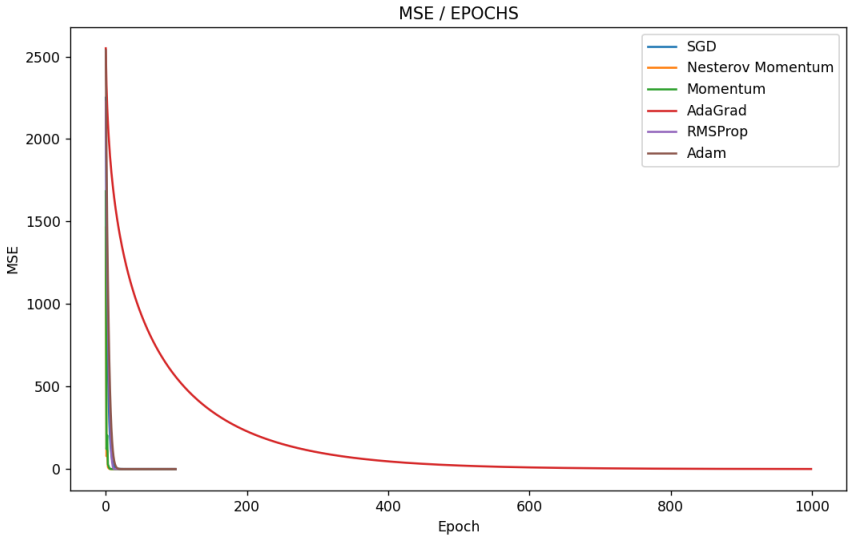
Видим, что SGD, Nesterov и Momentum хорошо справляются со своей задачей. А вот AdaGrad, RMSProp и Adam - не справляются. В принципе, из описаний методов логично предположить такой исход, так как в нашем

случае нет редких данных или чего то подобного. Поэтому скорость сходимости на таких алгоритмах падает.

batch size	learning rate	modification	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
200	0.01	SGD	100	0.0045	0.0016	0.4884	0.08739	5094	3017897
200	0.01	NESTEROV	100	0.0045	0.00053	0.489	0.08742	10992	1225114
200	0.01	MOMENTUM	100	0.0026	0.00010	0.4910	0.08763	14023	1092137
200	0.5	ADAGARD	100	0.252	0.617	23.016	532	6216	960186
200	0.5	RMSPROP	100	0.530	0.339	0.8554	1.33898	4978	1240172
200	0.5	ADAM	100	0.0072	0.0023	0.502	0.102	4891	1460558



Из данных результатов можно понять, что RMSProp и Adam начинают показывать хорошие результаты при большом размере шага. AdaGrad тоже начинает работать лучше, но пока не идеально.



Видим, что AdaGrad действительно сходится, но ему нужен большой шаг и много итераций.

batch size	learning rate	modification	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
200	0.5	SGD	100	overflow	overflow	overflow	overflow	5296	3069542
200	0.5	NESTEROV	100	overflow	overflow	overflow	overflow	5285	1342937
200	0.5	MOMENTUM	100	0.0595	0.1613	0.4720	0.1870	5312	1061431
200	0.8	ADAGARD	200	0.123	0.039	4.1082	22.098	11358	1148014
200	0.5	RMSPROP	100	0.530	0.339	0.8554	1.33898	4978	1240172
200	0.5	ADAM	100	0.0072	0.0023	0.502	0.102	4891	1460558

С другой стороны, видно, что при шаге 0.5 GSD и Nesterov переполняются. При больших размерах шага Momentum так же переполняется.

Еще в глаза бросается большое использование памяти и времени по сравнению с собственными реализациями. Скорее всего, это происходит потому, что библиотечные реализации устроены гораздо сложнее и требуют дополнительных вычислений или информации.

batch size	learning rate	modification	epochs	a_diff	b_diff	c_diff	mse	time (ms)	memory (bytes)
200	0.5	SGD	100	overflow	overflow	overflow	overflow	5296	3069542
200	0.5	NESTEROV	100	overflow	overflow	overflow	overflow	5285	1342937
200	0.5	MOMENTUM	100	0.0595	0.1613	0.4720	0.1870	5312	1061431
200	0.8	ADAGARD	200	0.123	0.039	4.1082	22.098	11358	1148014
200	0.5	RMSPROP	100	0.530	0.339	0.8554	1.33898	4978	1240172
200	0.5	ADAM	100	0.0072	0.0023	0.502	0.102	4891	1460558

Замечаем, что от увеличения размера батча память не сильно увеличилась. А вот при большем количестве итераций время увеличивается. Скорее всего, фокусы в библиотечной реализации с памятью.

Итак, в данных модификациях есть различия. Многое зависит от изначального набора данных. На конкретно наших данных алгоритмы SGD, Momentum, Nesterov работают примерно одинаково. Примерно одинаковая погрешность, скорость. Только объем потребляемой памяти больше у SGD, но время

его работы в среднем меньше. Все эти три модификации ломаются при большом шаге.

Также похоже работают RMSProp и Adam, что неудивительно, так как один метод является модификацией второго. Сходимость на наших данных при маленьком шаге очень низкая, а при большом - высокая. Потребление памяти и времени примерно одинаковое, но в среднем Adam потребляет больше памяти.

AdaGrad выделяется из этой шестерки. На нашем наборе данных ему нужны специальные условия - большие шаг и количество итераций. И все равно погрешность остается высокой. Возможно, это потому, что у нас нет редких данных.

В среднем у библиотечных функций погрешность меньше, чем у собственных реализаций SGD. Однако по времени работы и потреблению памяти явно лучше смотрится ручная реализация. Скорее всего, это потому что библиотечные функции рассчитаны на сложные случаи, используют дополнительную информацию, методы и прочее. А ручная реализации рассчитана только на простенький пример и не имеет замысловатых вычислений.