МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

КУРСОВОЙ ПРОЕКТ
ЗАЩИЩЕН С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

доцент, канд. физ.-мат. наук		Н.А. Волкова
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ

Градиентные алгоритмы: Алгоритм сопряженных градиентов, ADAM, SQN (stochastic quasi-Newton), SARAH (StochAstic Recusive gRadient algoritHm)

по курсу: ПРИКЛАДНЫЕ МОДЕЛИ ОПТИМИЗАЦИИ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4132		Н.И. Карпов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

Содержание

1 ВВЕДЕНИЕ

1.1 Актуальность

Градиентные алгоритмы - это семейство методов оптимизации, для нахождения экстремумов функций. Идея градиентных методов проста, а многие алгоритмы итеративны, что делает их крайне популярными в современных задачах. Применение градиентные методы нашли во многих областях, требующих нахождение минимумов и максимумов функций. Этими областями являются экономика, физика, статистика, математика. Но самое распространенное и актуальное использование градиентных методов на данный момент - машинное обучение [8]. Во многом благодаря градиентным алгоритмам произошел мощный рывок в развитии различных моделей искусственного интеллекта, так как именно эти алгоритмы легли в основу самого эффективного до сих пор метода глубокого обучения - метода обратного распространения ошибки [5]. Данные направления являются быстро развивающимися и оказывают огромное влияние на науку и прикладные задачи, а одним из самых важных инструментов, которые они используют, являются градиентные алгоритмы. Поэтому важно и актуально исследовать градиентные алгоритмы с целью более эффективного и правильного их использования, а также с целью их дальнейшей модификации.

1.2 Цели и задачи

Цель на курсовую работу - установить общие принципы и основные различия градиентных алгоритмов, таких как алгоритм сопряженных градиентов, алгоритмы ADAM, SQN, SARAH; с помощью теории и практики определенить характеристики и сильные стороны конкретных градиентных алгоритмов.

Задачи на курсовую работу:

- изложить теоретические принципы основных градиентных алгоритмов;
- реализовать градиентные алгоритмы на языке программирования Python;
- изучить поведение алгоритмов в нескольких оптимизационных задачах;

- сделать выводы об использовании градиентных алгоритмов в оптимизационных задачах.

1.3 Объект и предмет исследования

Объект исследования: алгоритм сопряженных градиентов, алгоритмы ADAM, SQN, SARAH

Предмет исследования: характеристики и поведение градиентных алгоритмов при решении оптимизационных задач

1.4 Теоретическая основа

В качестве теоретической основы использовались англоязычные и русскоязычные статьи о градиентных алгоритмах и математических концепциях, которые приведены в списке используемых источников.

1.5 Методы изучения

Основными методами изучения являются теоретическое изучение принципов алгоритмов и эмпирическое изучение алгоритмов посредством их реализации и тестирования.

1.6 Новизна и практическая значимость

Новизна и значимость работы заключается в структуризации информации о градиентных алгоритмах, о их характеристиках, отличиях и о том, какую реализацию градиентного алгоритма выбрать для решения конкретной задачи.

1.7 Структура работы

В работе будут освещены алгоритм сопряженных градиентов, алгоритмы ADAM, SQN, SARAH. Для каждого будет дано теоретическое описание. Алгоритмы сопряженных градиентов, ADAM, SQN будет реализованы и протестированы. Для каждого алгоритма будут оценены

отличительные особенности и характеристики, а также будет сделан вывод по каждому из алгоритмов в частности и о градиентных методах в целом.

2 ОСНОВНАЯ ЧАСТЬ

2.1 Теоретический материал

2.1.1 Градиентный спуск

Чтобы рассмотреть все продвинутые градиентные алгоритмы необходимо начать с самого простого алгоритма - градиентного спуска, в англоязычной литературе известного как gradient descent. Задача всех градиентных алгоритмов состоит в нахождение экстремума функции. В контексте данной работы будет рассматриваться нахождение именно минимума функции.

Предположим, что стоит задача в минимизации функции $f(x) = x^2$.

Выберем случайную точку аргумента функции - начальное приближение. Пусть, это будет точка $x_0 = 4$.

Данная точка далека от минимума функции, но пока это ещё не известно.

Теперь вспомним, что производная функции характеризует величину возрастания функции. Это означает, что если в точке функция убывает, то её производная будет отрицательна и, наоборот, если функция возрастает, то её производная - положительна. На основании этого можно сделать вывод куда ”идет функция”. Если производная больше 0, то значение функции увеличивается, что в рамках задачи минимизации - плохо. Поэтому следует ”идти” против знака производной, тем самым и при убывании и при возрастании значение функции на каждом шаге будет приближаться к минимальному.

Таким образом, можно создать алгоритм, по которому на каждом шаге будет вычисляться производная в точке, затем относительно этой точки будет производиться шаг по аргументу в направлении, противоположном производной. Шаг в практических задачах выбриают как $\alpha \cdot f'(x)$, где α - неизменный шаг, или learning rate, а $f'(x_n)$ - значение производной в текущей точке. Учитывая, что двигаться нужно против знака производной, получаем формулу (1):

$$x_{n+1} = x_n - \alpha f'(x_n) \tag{1}$$

Теперь рассмотрим далее практический пример минимизации. Положим шаг равным $\alpha = 0.2$ (чем меньше шаг - тем точнее результат, но медленее решение) Затем найдём производную функции: $f'(x) = 2x$.

Рассчитаем значение производной в точке x_0, получим: $f'(x_0) = 2x_0 => f'(x_0) = 8$.

Теперь применим формулу (1) и найдём следующее приближение: $x_1 = x_0 - \alpha \cdot f'(x_0) => x_1 = 4 - 0.2 \cdot 10 <=> x_1 = 2$ Наконец, повторяя предыдущий шаг сколь угодно раз, задача минимизации может быть решена с достаточно высокой точностью! $x_2 = 2 - 0.2 \cdot 4 = 1.2 x_3 = 1.2 - 0.2 \cdot 2.4 = 0.72 x_4 = 0.72 - 0.2 \cdot 1.44 = 0.432$ Видно, что решение сходится к 0 (что верно). При этом вычисления максимально просты и требуют только пересчет градиента в точке по заранее выведенной формуле производной.

Теперь необходимо рассмотреть многомерный случай. Очевидно, что теперь придется работать с частными производными и, соответственно, с градиентами.

Решая всё ту же задачу минимизации функция будет иметь вид $y = f(x)$, где y - значение функции; x - многомерный вектор аргументов функции. Перепишем формулу (1) для многомерного случая. Заменим производную функции одной переменной градиентом многомерной функции: $\nabla_f = (dy/dx_1, dy/dx_2, ..., dy/dx_n)$, где n - количество параметров функции (количество измерений минус 1).

Градиент, аналогично производной, отражает направление наискорейшего роста функции, но уже для многомерного случая. Ввиду этого остальные формулы останутся практически без изменений. Так, общая формула для градиентного спуска (2):

$$x_{n+1} = x_n - \alpha \nabla_f \tag{2}$$

2.1.2 Динамический шаг

Формулы (2) достаточно для многих практических задач. Но для полного описания алгоритма необходимо описать и другой подход к выбору шага градиентного спуска. Статичный шаг хоть и является крайне удобным, его эффективность очевидно невысока. Логично брать более большие шаги там, где функция далека от минимума, то есть она убывает с большой скоростью,

и, наоборот, подбираясь к решению стоит выбирать шаг крайне маленьким, чтобы обеспечить высокую точность.

Логично, что шаг нужно выбирать так, чтобы спуск к минимуму осуществлялся до тех пор, пока функция убывает. Так будет обеспечена наибольшая скорость движения по антиградиенту в каждой точке. Поиск такого шага является задачей одномерной оптимизации и может быть решён многими методами, например, методом Фибоначчи, методом золотого сечения, методом бисекций. Но в этой работе будет использоваться метод Ньютона–Рафсона, так как он обеспечивает наиболее быструю сходимость. Но данный метод предполагает вычисление матрицы Гессе (то есть вторые производные), ввиду чего он является методом второго порядка и слабо применим в некоторых практических задачах, таких как машинное обучение.

В этой работе для более общего описания алгоритмов и ухода от статического шага будет использоваться именно этот метод, но в практических задачах, требовательных к скорости вычислений или с ограничением к нахождению вторых производных, вполне разумно использовать другие методы или вовсе статический шаг.

Итак, для одномерной минимизации функции по шагу можно использовать формулу (3):

$$\alpha = \frac{f'^{\intercal}(x_n)\nabla_f}{\nabla_f^{\intercal} f''(x_n)\nabla_f} \tag{3}$$

2.1.3 Алгоритм сопряженных градиентов

Алгоритм сопряженных градиентов или conjugate gradient descent (CGD) [9] - усовершенствование стандартного градиентного спуска, который использует идею корректировки направления антиградиента так, чтобы новый вектор антиградиента был сопряжен с предыдущим.

Рассмотрим целевую функцию $f(x) = Ax^{\top}x - bx + c$, где A - квадратная, симметричная, положительно–определенная матрица коэффициентов, x - вектор аргументов функции, b - вектор коэффициентов, c - скалярная константа. Данная функция представляет собой многомерную квадратичную функцию, для случая когда x представляет собой вектор из двух аргументов:

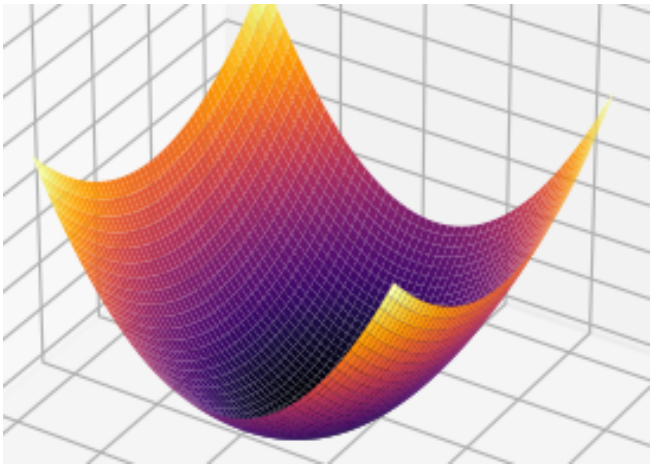

Рис. 1: эллиптический параболоид

Если построить линии уровня такой функции и отразить на них работу обычного градиентного спуска, то можно получить такую ситуацию:

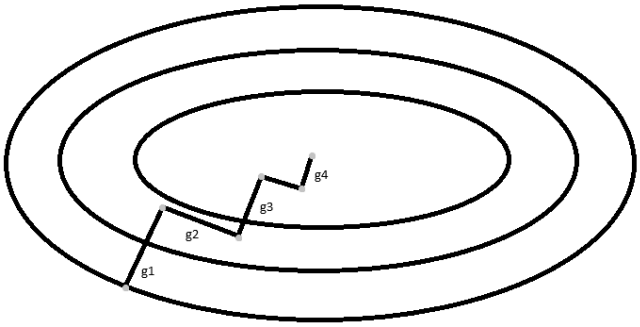

Рис. 2: градиентный спуск

Очевидно, что алгоритм градиентного спуска здесь выбирает такое направление минимизации функции, которое является ортогональным предыдущему. Но такое поведение оказывается эффективным только в ”невытянутых” квадратичных функциях (с матрицей A, где по диагонали коэффициенты одинаковы). От этого недостатка свободен подход с нахождением сопряженных относительно ранее упомянутой матрицы коэффициентов A направлений.

Сопряженным вектором относительно g_1 является такой вектор g_2, для которого скалярное произведение g_1 и A_{g_2} равно нулю, то есть векторы g_1 и g_2 ортогональны по отношению к матрице A: $g_1^\intercal A_{g_2} = 0$.

Это означает, что направления будут выбираться подобным образом:

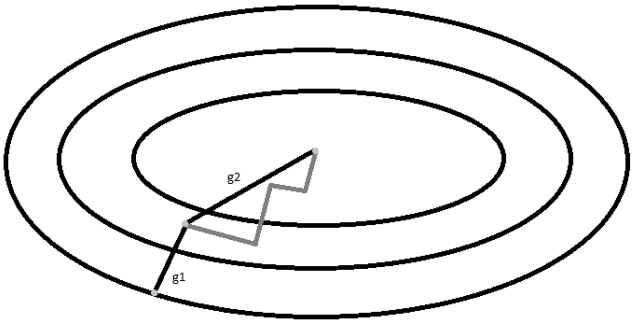

Рис. 3: метод сопряженных градиентов

Метод сопряженных градиентов для такого случая гарантирует решение за d шагов, где d - количество аргументов функций, т.е. количество измерений минус 1. Но в общем случае такое решение не гарантировано.

На первый взгляд они совсем не ортогональны, однако если мысленно ”сузить” квадратичную функцию, то можно увидеть, что эти векторы - перпеникулярны:

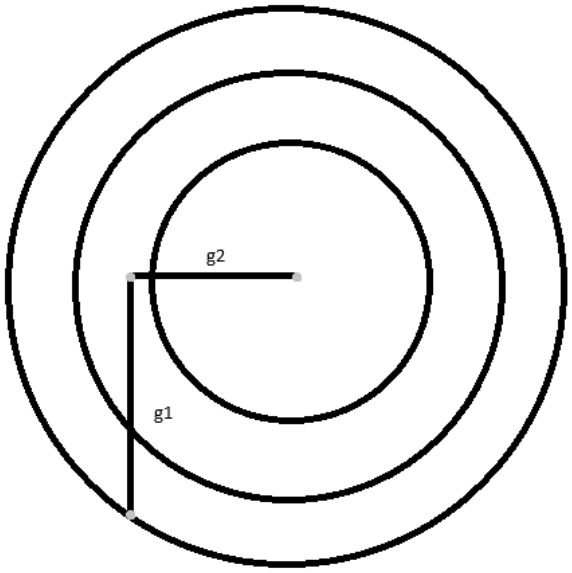

Рис. 4: сопряженность градиентов

Реализуется метод сопряженных градиентов аналогично градиентному спуску, за исключением корректировки направления антиградиента на каждом шаге согласно какому-либо методу ортогонализации, например методу Флетчера-Ривса (4):

$$\beta_n = \frac{\nabla_n^\top \nabla_n}{\nabla_{n-1}^\top \nabla_{n-1}} \tag{4}$$

Вычисленный коэффициент является скаляром. Поскольку в общем случае решение может сходится за куда большее количество шагов, чем количество измерений, метод предполагает выполнять обнуление или "сброс" коэффициента β каждые $d + 1$ шагов.

Корректировка градиента осуществляется по формуле (5):

$$\nabla_{n+1} = \nabla_{n+1} + \beta\nabla_n \tag{5}$$

Итак, теперь можно записать весь алгоритм решения оптимизационных задач методом сопряженных градиентов:

1) Вычислить градиент в точке начального приближения x_0 ($n = 0$);
2) Решить задачу одномерной минимизации функции $f(x_n - \alpha_n\nabla_n)$, то есть найти такой максимальный шаг, при котором функция в направлении антиградиента убывает, например, по формуле 3. Данный шаг пропускается, если коэффициент α принято считать статичным;
3) Вычислить новое приближение аргумента по формуле 2: $x_{n+1} = x_n - \alpha_n\nabla_n$;
4) Вычислить градиент в новой точке $\nabla_{n+1} = f'(x_{n+1})$;
5) Если итерация алгоритма не первая ($n > 0$), то:
    a) Вычислить коэффициент корректировки β, например, по формуле 4;
    b) Если $n\%d = 0$, то "сбросить" направление, т.е. положить, что $\beta = 0$
    c) Скорректировать ранее вычисленный градиент по формуле (5);
6) Перейти к шагу 2, если точности полученного приближения не хватает для завершения вычислений.

Несмотря на то, что был рассмотрен случай для параболоида, данный метод применим к любым другим функциям.

2.1.4 Алгоритм ADAM (Adaptive Moment Estimation)

Вышеописанные алгоритмы имеют один большой недостаток - они не выбираются из локальных минимумов. Разумеется, в практических задачах, где функции крайне сложны и имеют множество локальных минимумов, эта проблема серьёзно снижает точность решения:

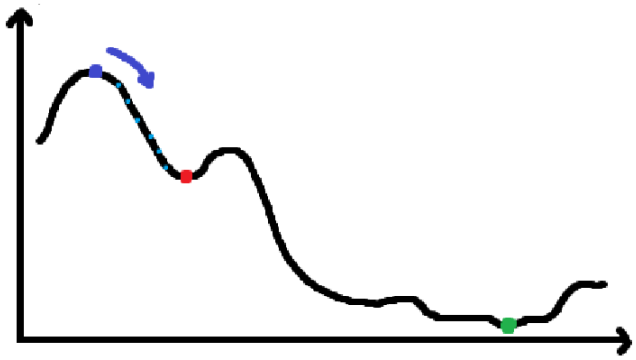

Рис. 5: красным отмечен локальный минимум, зеленым - глобальный. GD и CGD останутся на красной точке

От этого недостатка свободны методы с применением так называемых ”моментов” или ”импульсов”, в числе которых есть один из самых часто применяемых в задачах машинного обучения алгоритм ADAM [1].

ADAM объединяет в себе идеи метода импульсов (Momentum Method) и метода RMSProp [2] по использованию скользящей средней [6] при движении по антиградиенту.

В данном алгоритме на каждом шаге текущее значение градиента усредняется с помощью двух экспоненциальных скользящих средних, аккумулирующих в себе все предыдущие значения градиентов. Первая оценка импульса осуществляется по формуле (6):

$$\nu_n = \beta_1 \nu_{n-1} + (1 - \beta_1) \nabla_n \tag{6}$$

Здесь ν - экспоненциальное скользящее среднее, β_1 - статический коэффициент затухания импульса в пределах от 0 до 1 (не включительно). Чем меньше коэффициент, тем быстрее ”забываются” предыдущие значения градиентов, что и выражено в затухании импульса.

Вторая оценка осуществляется по формуле (7):

$$\mu_n = \beta_2 \mu_{n-1} + (1 - \beta_2) \nabla_n \odot \nabla_n \tag{7}$$

Здесь аналогично mu - взвешенное скользящее среднее, β_2 - статический коэффициент затухания квадратов градиентов в пределах от 0 до 1 (не включительно).

Суть первой оценки заключается в нахождении затухающего импульса на основе значений предшествующих градиентов. На каждой итерации скользящее среднее будет содержать усредненные значения градиентов на предыдущих шагах (среднее постепенно уменьшается), что позже будет использоваться для добавления импульса к шагам и более быстрой сходимости. Вторая оценка отличается от первой только в поэлементном умножении градиента на самого себя. Но её дальнейшее использование позволяет контролировать шаг движения вдоль антиградиента для каждой переменной функции в отдельности, избегая глобальной усредненной скорости. Обе оценки являются векторами.

Далее эти оценки нормируются так, чтобы на начальных шагах иметь большие значения, а на последующих - меньшие значения. Это реализуется следующими формулами (8 и 9):

$$\hat{\nu_n} = \frac{\nu_n}{1 - \beta_1^{n+1}} \tag{8}$$

$$\hat{\mu_n} = \frac{\mu_n}{1 - \beta_2^{n+1}} \tag{9}$$

Эти формулы крайне просты и имеют следующий смысл: так как гиперпараметры меньше единицы, то возведение их в степень с большим основанием будет давать меньшие результаты. Так как итерация алгоритма n увеличивается, то ν^{n+1} будет уменьшаться с каждым шагом. При вычитании полученного значения из единицы будет получаться постепенно возрастающее значение в знаменателе, меньшее единицы. Наконец, при делении оценки на знаменатель она будет аналогично увеличиваться, но с каждым шагом степень увеличения будет затухать. Так будут получаться более быстрые шаги на начальных этапах, что ускорит сходимость.

Важно заметить, что в общем случае $\beta_1 \neq \beta_2$, поэтому оценки будут нормироваться поразному.

Далее выполняется самый важный шаг: первая оценка, умноженная на шаг α, делится на квадратный корень второй, сохраняя от первой глобальный импульс, а от второй - уточненный для каждого из параметров. Вычисленная характеристика вычитается из предыдущего приближения по аналогии с формулой 2:

$$x_{n+1} = x_n - \alpha \frac{\hat{\nu_n}}{\hat{\mu_n} + \varepsilon} \tag{10}$$

Где ε - это очень малая константа, которая исключает деление на 0.

На практике ADAM показывает очень хорошие результаты, так как именно усреднение градиентов позволяет им не ”утихать” при достижении локального минимума (так как градиент не сразу становится равным очень малому числу, а усредняется с предшествующими градиентами и приобретает большее значение), что даёт возможность выбраться из локального минимума и найти более выгодный оптимум. Также стоит отметить, что сами шаги становятся больше, когда алгоритм далёк от оптимума, так как скользящие средние увеличивают шаг в направлении антиградиента, что ускоряет сходимость.

Другими словами, на каждом шаге к градиенту добавляется экспотенциальное затухающее на основе предыдущих градиентов, которое является импульсом, подталкивающим решение. Также добавляется экспотенциальное затухающее на основе квадратов предыдущих градиентов, которое добавляет скорость изменения тем параметрам, которые изменяются мало, и наоборот.

Итак, алгоритм ADAM можно представить так:

1) Вычислить градиент в точке начального приближения x_0 ($n = 0$);

2) Решить задачу одномерной минимизации функции $f(x_n - \alpha_n \nabla_n)$, то есть найти такой максимальный шаг, при котором функция в направлении антиградиента убывает, например, по формуле 3. Данный шаг пропускается, если коэффициент α принято считать статичным;

3) Вычислить экспоненциальные скользящие средние на основе градиентов и квадратов градиентов по формулам 6 и 7;

4) Осуществить нормировку скользящих средних согласно формулам 8 и 9;

5) Вычислить новое приближение аргумента по формуле 10;

6) Вычислить градиент в новой точке $\nabla_{n+1} = f'(x_{n+1})$;

7) Перейти к шагу 2, если точности полученного приближения не хватает для завершения вычислений.

2.1.5 Алгоритмы SQN (stochastic quasi-Newton)

Все вышеописанные алгоритмы являются методами первого порядка, так как не подразумевают нахождение вторых производных. Алгоритмы SQN - это семейство алгоритмов, которые используют матрицу Гессе, или приближения к ней для решения оптимизационных задач.

Глобально такие алгоритмы отличаются от градиентного спуска формулой вычисления нового приближения:

$$x_{n+1} = x_n + \alpha H_n p_n \tag{11}$$

Где $p_n = -\nabla_n$ - направление вдоль антиградиента.

Можно заметить, что здесь матрица вторых производных фактически умножается на вектор первых производных, что в итоге даёт также вектор.

Также в SQN шаг α принято находить таким, чтобы он удовлетворял условиям Вольфе [7]:

$$f(x_n + \alpha_n * p_n) \leqslant f(x_n) + c_1 \alpha_n p_n^\top \nabla_n \tag{12}$$

$$-p_n^\top \nabla_{n+1} \leqslant -c_2 p_n^\top \nabla_n \tag{13}$$

Где c_1 - константа, близкая к нулю, c_2 - константа, близкая к 1. Первое неравенство говорит о том, что новое приближение функции должно быть меньше предыдущего. Второе неравенство - о том, что проекция градиента в новом приближении должна изменить направление

или величину.

Коэффициент α, удовлетворяющий условиям Вольфе, обеспечивает такой шаг вдоль направления, что функция будет принимать своё наименьшее значение вдоль этого шага. По сути, это вновь задача одномерной минимизации функции от α.

Итак, алгоритм SQN можно представить так:

1) Вычислить градиент в точке начального приближения x_0 ($n = 0$);
2) Положить направлением движения антиградиент, т.е. $p_n = -\nabla_n$;
3) Решить задачу одномерной минимизации и найти такое значение скаляра α, которое будет удовлетворять условиям Вольфе (12-13);
4) Рассчитать значение матрицы Гессе в точке x_n;
5) Вычислить новое приближение аргумента по формуле 11;
6) Вычислить градиент в новой точке $\nabla_{n+1} = f'(x_{n+1})$;
7) Перейти к шагу 2, если точности полученного приближения не хватает для завершения вычислений.

Вычисление матрицы Гессе является весьма трудоёмким и накладывает ограничения на целевую функцию. Поэтому существует несколько вариантов SQN [4], которые используют приближения к матрице Гессе, именно они и используются для решения практических задач. Такие алгоритмы итеративно высчитывают приближение матрицы Гессе, ввиду чего они остаются методами первого порядка.

2.1.6 Алгоритм SARAH (StochAstic Recusive gRadient algoritHm)

Метод SARAH [3] имеет смысл рассматривать в задачах, где на вход алгоритма подаётся большое количество оптимизируемых целевых функций с одинаковым набором параметров, но разными коэффициентами перед ними. Самая популярная такая задача - машинное обучение. В

ней целевые функции - функции ошибок, параметры - веса, а коэффициенты - входные данные. Далее эти целевые функции будут называться выборкой.

Поскольку данный алгоритм широко используется именно в контексте машинного обучения, в этом разделе вместо знака точек x будет использоваться знак весов ω.

Не в стохастических алгоритмах необходимо вычислять градиенты для каждого элемента выборки, затем складывать их и усреднять, что крайне неэффективно. Стохастические алгоритмы используют другой подход: на каждом шаге выбирается случайный элемент выборки, или m случайных элементов, и уже на основе их высчитываются градиенты и осуществляются приближения оптимизируемых параметров. Если стохастический градиентный спуск выполняет простое усреднение случайно выбранных элементов, то SARAH использует другой подход. Внутри глобальной итеративной цепочки вычисления приближений запускается новый цикл. В нем для каждого из m выбранных случайно элементов выборки высчитывается оценка градиента, основанная на предыдущих шагах этого же цикла:

$$\nu_t = \nabla f_{i_t}(\omega_t) - \nabla f_{i_t}(\omega_{t-1}) + \nu_{t-1} \tag{14}$$

Где t - номер внутренней итерации от 1 до m; f_{i_t} - случайный элемент выборки, то есть случайная целевая функция, по которому вычисляется градиент в точке ω_t.

Разница градиентов в текущей и предыдущей точках одной случайно выбранной функции, к которой прибавляется оценка с предыдущей итерации, формирует направление, против которого будет совершаться движение для вычисления ещё одного приближения:

$$\omega_{t+1} = \omega_t - \alpha * \nu_t \tag{15}$$

Шаг α принято считать статичным.

Так, за $m-1$ внутренних итераций, с учетом нулевой итерации с инициализацией значений для внутреннего цикла, будет получено m значений весов ω_n. То есть на глобальной итерации n мы получим m возможных приближений, каждое из которых было высчитано на основе случайно выбранной функции. Так как глобальное приближение нужно совершить одно, то слу-

чайно будет выбрана одна из m точек.

В итоге на каждой глобальной итерации не просто высчитывается градиент случайного элемента выборки, а выбирается случайное приближение из возможных m, каждое из которых содержит информацию о предшествующих ему (в том числе и нулевое, так как оно аналогично было выбрано на предыдущей глобальной итерации).

Алгоритм SARAH можно записать так:

1) Случайно выбрать целевую функцию для первой итерации, инициализировать начальное приближение, рассчитать градиент случайной функции для $n = 0$, положить n равным 1. Положить d равным числу параметров целевой функции, инициализоровать константу m;
2) Положить ω_0 равным ω_{n-1};
3) Начальной оценкой положить $\nu_0 = \nabla_{n-1}$;
4) Вычислить $\omega_1 = \omega_0 - \alpha\nu_0$;
5) Начать цикл от t=1 до m:
   - Случайно выбрать функцию f_{i_t};
   - Вычислить стохастическую оценку ν_t по формуле 14;
   - Вычислить приближение ω_{t+1} по формуле 15;
6) Обновить ω_n случайно выбранным приближением: $\omega_n = \omega_t$, где $t = 0, 1, 2, ..., m$;
7) Перейти к шагу 2, если точности полученного приближения не хватает для завершения вычислений.

Как можно заметить, SARAH предназначен именно для специализированных задач (в рассматриваемой в этой работе общей задачи минимизации двумерной функции SARAH будет идентичен обычному градиентному спуску), поэтому его сравнительный анализ с другими алгоритмами будет проводится только на теоретической основе.

2.2 Реализации алгоритмов

2.2.1 Общий обзор

Для реализации всех вышеописанных алгоритмов использовался язык программирования Python 3. Для каждого алгоритма был написан оптимизатор - класс, который выполняет пошаговое решение или решение задачи целиком. Кроме того, была написана вся необходимая инфраструктура для решения оптимизационных задач: несколько целевых функций, классы для отрисовки решения, управляющий класс.

Для операций над матрицами и векторами используется библиотека numpy, для некоторых математических расчетов используется scipy, для построения графиков - библиотека matplotlib.

Код всех файлов приведен в приложении А.

2.2.2 Реализация алгоритма сопряженных градиентов

Алгоритм сопряженных градиентов был реализован следующим образом:

```
def count_a_step(gradient, second_order_differential_in_dot_x):
    """
    Вычисляет динамический шаг градиента альфа()

    Шаг вычисляется по формуле:

    alpha = (gradient.T * gradient) / (gradient.T * (f''(x) * gradient))
    """

    det = (gradient.dot(second_order_differential_in_dot_x.dot(gradient)))

    if det == 0:
        return 0

```

```
15      return (gradient.dot(gradient)) / (gradient.dot(
        second_order_differential_in_dot_x.dot(gradient)))
16
17  # Начальное приближение
18  self.x = self.initial_x
19
20  # Номер итерации
21  self.n = 0
22
23  # Градиент и антиградиент в точке начального приближения
24  self.gradient = self.target_function.differential_in_dot(self.x)
25
26  def make_step(self):
27      self.x = self.initial_x
28
29          # Номер итерации
30          self.n = 0
31
32          # Градиент и антиградиент в точке начального приближения
33          self.gradient = self.target_function.differential_in_dot(self.x)
34
35  def make_step(self):
36      # Высчитываем динамический шаг см(. util.count_a_step)
37      alpha = count_a_step(self.gradient, self.target_function.
        second_order_differential_in_dot(self.x))
38
39      # Находим новое приближение
40      self.x = self.x - alpha * self.gradient
41      y = self.target_function(self.x)
42
43      # Находим градиент в точке нового приближения
44      new_gradient = self.target_function.differential_in_dot(self.x)
45
```

```
46      # Не на нулевой итерации вычисляем сопряженное направление, обновляем
    градиент
47      if self.n != 0:
48          # Знаменатель
49          det = self.gradient.dot(self.gradient)
50
51          # Для избежания деления на 0
52          if det == 0:
53              beta = 0
54          else:
55              # Формула ФлетчераРивса-
56              beta = new_gradient.dot(new_gradient) / det
57
58              # Сброс каждые n + 1 шагов
59              if self.n % (self.x.shape[0] + 1) == 0:
60                  beta = 0
61
62          # Обновляем градиент
63          self.gradient = new_gradient + beta * self.gradient
64      else:
65          self.gradient = new_gradient
66
67      self.n += 1
68
69      return [self.x, y]
```

В данной реализации используются методы унаследованных от класса $TargetFunction$ классов для нахождения значения функции в текущей точке, производной в точке и второй производной точке. Ввиду этого реализация является методом второго порядка. Здесь и далее target_function.differential_in_dot(x) означает нахождение вектора градиентов в точке на основе целевой функции, target_function.second_order_differential_in_dot(x) - нахождение матрицы Гессе в точке, а count_a_step - подсчет динамического шага на основе метода Ньютона-Рафсона.

Также был реализован алгоритм простого наискорейшего спуска для сравнения с моди-

фикацией, рассмотренной в этом разделе. Его код приведен в приложениях вместе с остальными оптимизаторами.

2.2.3 Реализация алгоритма ADAM

Алгоритм ADAM был реализован следующим образом:

```
1  def __init__(self, func: TargetFunction, initial_x, alpha, beta1=0.9,
   beta2=0.999, epsilon=1e-8):
2      super().__init__(func, initial_x)
3
4      self.x = self.initial_x
5
6      self.k = 0
7
8      self.gradient = self.target_function.differential_in_dot(self.x)
9
10     # Булева переменная об использовании динамического шага
11     self.is_dynamic_step_used = False
12
13     if alpha is None:
14         self.is_dynamic_step_used = True
15     else:
16         self.alpha = alpha
17
18     # последнее скользящее среднее градиентов (moving_average)
19     self.last_ma = 0.
20     # последнее скользящее среднее квадратов градиентов (
   moving_average_of_squares)
21     self.last_mas = 0.
22
23     # коэффициент для скользящего среднего
24     self.beta1 = beta1
25
```

```
26          # коэффициент для скользящего среднего квадратов градиентов
27          self.beta2 = beta2
28
29          # слагаемое для избежания деления на 0
30          self.epsilon = epsilon
31
32      def make_step(self):
33          # подсчитываем динамический шаг, если не указано, что его нужно
        считать статическим, по методу НьютонаРафсона-
34          if self.is_dynamic_step_used:
35              self.alpha = count_a_step(self.gradient, self.target_function.
        second_order_differential_in_dot(self.x))
36
37          # высчитываем скользящее среднее аналогично методу импульсов
38          # v = v_n-1 * betta + (1 - betta) * alpha * gradient
39          self.last_ma = self.beta1 * self.last_ma + (1 - self.beta1) * self.
        gradient
40
41          # высчитываем скользящее среднее квадратов градиентов аналогично
        методу RMSProp
42          # v = v_n-1 * betta + (1 - betta) * alpha * gradient ^ 2
43          self.last_mas = self.beta2 * self.last_mas + (1 - self.beta2) * (self
        .gradient ** 2)
44
45          # Выполняем нормировку скользящих средних (v = v / (1 - betta^(k+1)))
        . С ростом числа проделанных шагов
46          # значение нормализованных величин будет уменьшаться.
47          normalized_ma = self.last_ma / (1 - np.power(self.beta1, self.k + 1))
48          normalized_mas = self.last_mas / (1 - np.power(self.beta2, self.k +
        1))
49
50          # x = x - alpha * (v / (sqrt(G) + eps))
```

```
51          self.x = self.x - self.alpha * (normalized_ma / (np.sqrt(
    normalized_mas) + self.epsilon))
52
53          y = self.target_function(self.x)
54
55          self.gradient = self.target_function.differential_in_dot(self.x)
56
57          # увеличиваем счетчик шагов
58          self.k += 1
59
60          return [self.x, y]
```

В данной реализации стоит отметить то, что была оставлена возможность рассчитывать динамический шаг α. Как правило, алгоритм ADAM использует статический шаг, поэтому расчет динамического шага - опционален. Для его отключения необходимо передать в конструктор класса значение статического шага.

Кроме алгоритма ADAM был реализован метод импульсов, его код приведен в приложениях вместе с остальными оптимизаторами.

2.2.4 Реализация алгоритма SQN

Алгоритм SQN был реализован следующим образом:

```
1 self.gradient = self.target_function.differential_in_dot(self.x)
2
3 def make_step(self):
4     hessian = self.target_function.second_order_differential_in_dot(self.x)
5
6     # высчитываем направление для движения вдоль антиградиента как матричное
    умножение якобиана на градиент
7     direction = -np.dot(hessian, self.gradient)
8
```

```
9     # выполняем линейный поиск константы альфа шага( вдоль направления) для
      удовлетворения условиям Вольфа
10    # фактически, мы ищем такой шаг альфа, при котором целевая функция в
      точке следующего шага будет минимальна
11    line_search = sp.optimize.line_search(self.target_function, self.
      target_function.differential_in_dot,
12                                          self.x, direction)
13    alpha = line_search[0]
14
15    if alpha is None:
16        alpha = 0.0
17
18    self.x = self.x + alpha * direction
19    y = self.target_function(self.x)
20
21    self.gradient = self.target_function.differential_in_dot(self.x)
22
23    return [self.x, y]
```

Данный метод, аналогично, использует матрицу Гессе, значение которой в текущей точке нужно рассчитывать на каждом шаге.

Также можно заметить, что линейный поиск шага, удовлетворяющего условиям Вольфе, выполнялся с помощью библиотеки scipy.

Также был реализован метод BFGS, как пример метода первого порядка с использованием гессиана. Его код приведен в приложениях вместе с остальными оптимизаторами.

2.3 Результаты решения оптимизационных задач

2.3.1 Общий обзор

Чтобы отразить преимущества различных алгоритмов были разработаны несколько целевых функций. Для нахождения минимумов каждой функции были запущены градиентные ал-

горитмы в двух режимах: отрисовка пошагового решения и решение задачи целиком с засеканием времени работы и количества шагов.

2.3.2 Метод сопряженных градиентов на практике

Как видно из теоретической справки, метод сопряженных градиентов идеален для случая с квадратичными функциями. Сравнивая метод наискорейшего спуска и метод сопряженных градиентов, оба из которых используют динамический шаг, можно сделать вывод, что второй оказывается намного эффективнее:

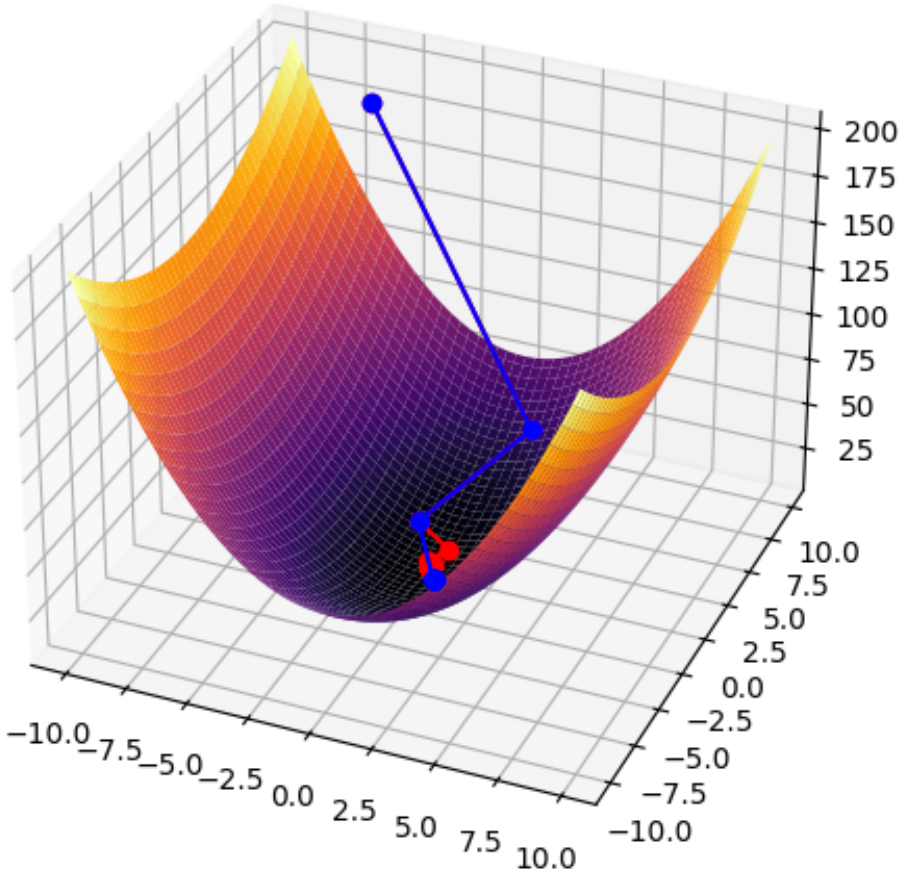

Рис. 6: красным отмечен наискорейший спуск, синим - метод сопряженных градиентов

Первые шаги алгоритмов совпадают, но далее вступает в дело подбор сопряженных направлений, что ускоряет сходимость. Особенно чётко разницу можно увидеть на последующих шагах, где методу наискорейшего спуска приходится делать множество шагов с затухающей

амплитудой, пока второй алгоритм получает направление прямиком к минимуму:

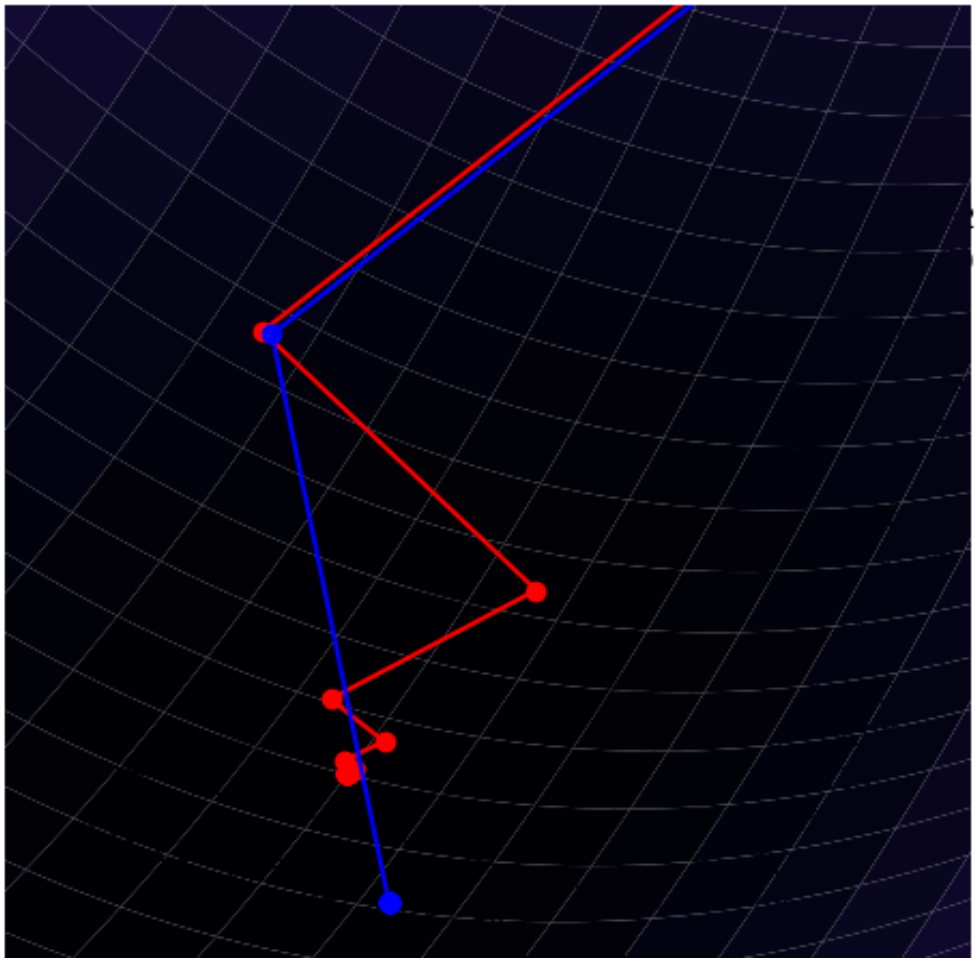

Рис. 7: сопряженные направления ускоряют поиск

Однако, можно заметить, что количество шагов метода сопряженных градиентов, хоть и мало, но больше заявленных теоретических d (d - количество аргументов целевой функции)! Такое происходит ввиду округлений математических операций, так как вычислительные системы не позволяют хранить бесконечно точные значения.

Как и ожидалось, алгоритм работает и на других, не квадратичных функциях. Разумеется, теперь нельзя ожидать количество шагов на уровне предыдущей задачи, но заметные ”рывки” вдали от минимума несколько ускоряют поиск:

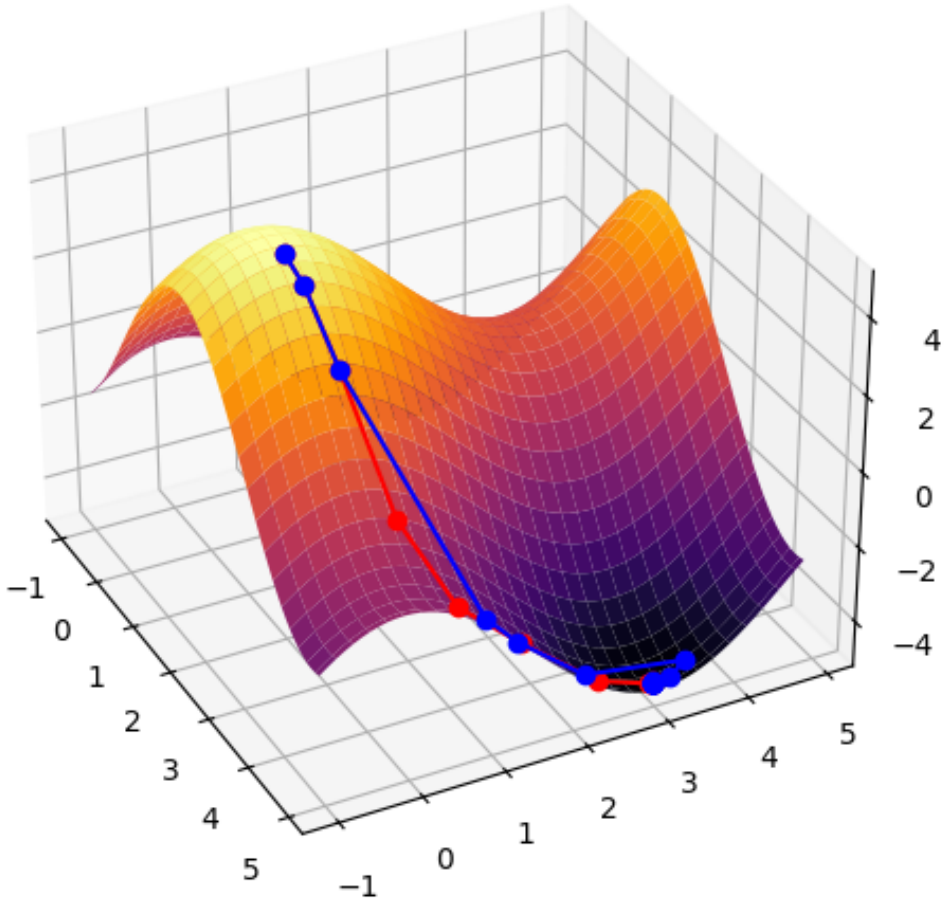

Рис. 8: поведение сопряженных градиентов на функции вида $f(z) = 3sin(x) + 2cos(y)$

Разница с градиентным спуском со статическим шагом ещё более выразительна, что касается всех алгоритмов, которые могут использовать динамический шаг:

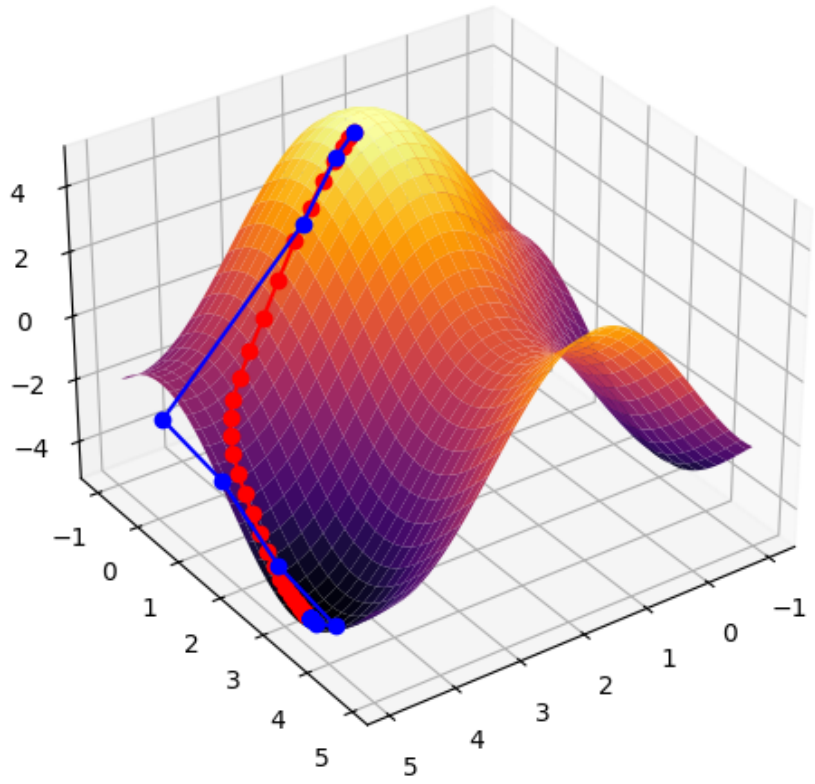

Рис. 9: градиентный спуск со статическим шагом

2.3.3 Метод ADAM на практике

На практике ADAM оказывается несколько медленным:

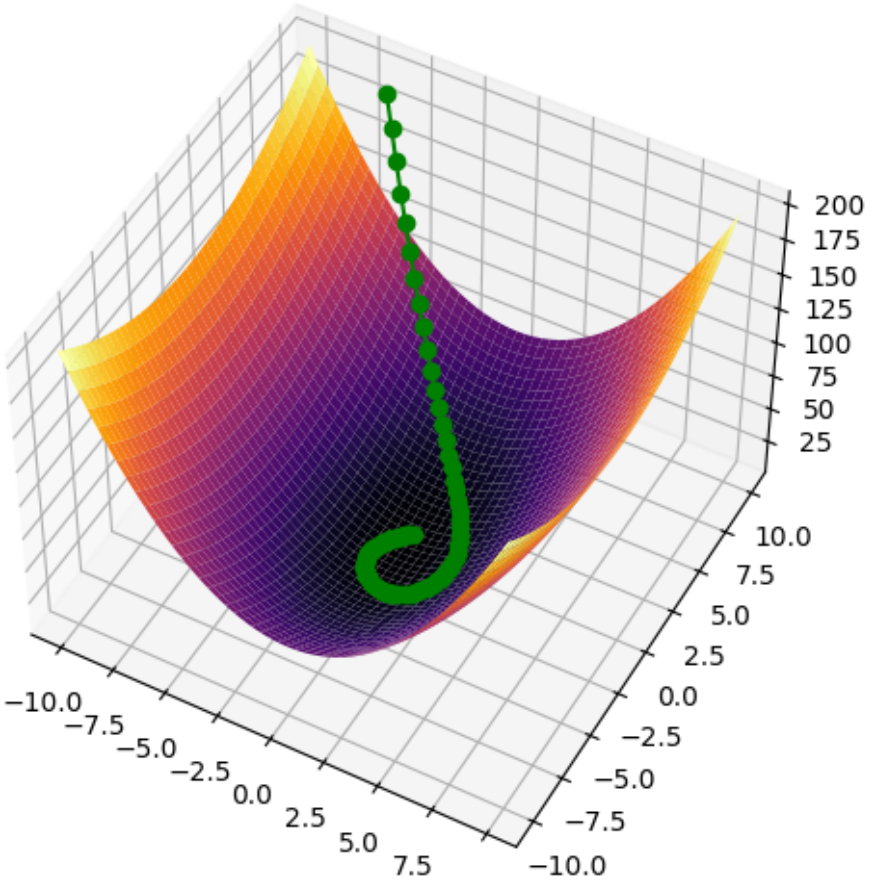

Рис. 10: алгоритм ADAM на параболоиде

Кажется, что количество шагов слишком велико, особенно в сравнении с градиентным спуском с тем же статическим шагом:

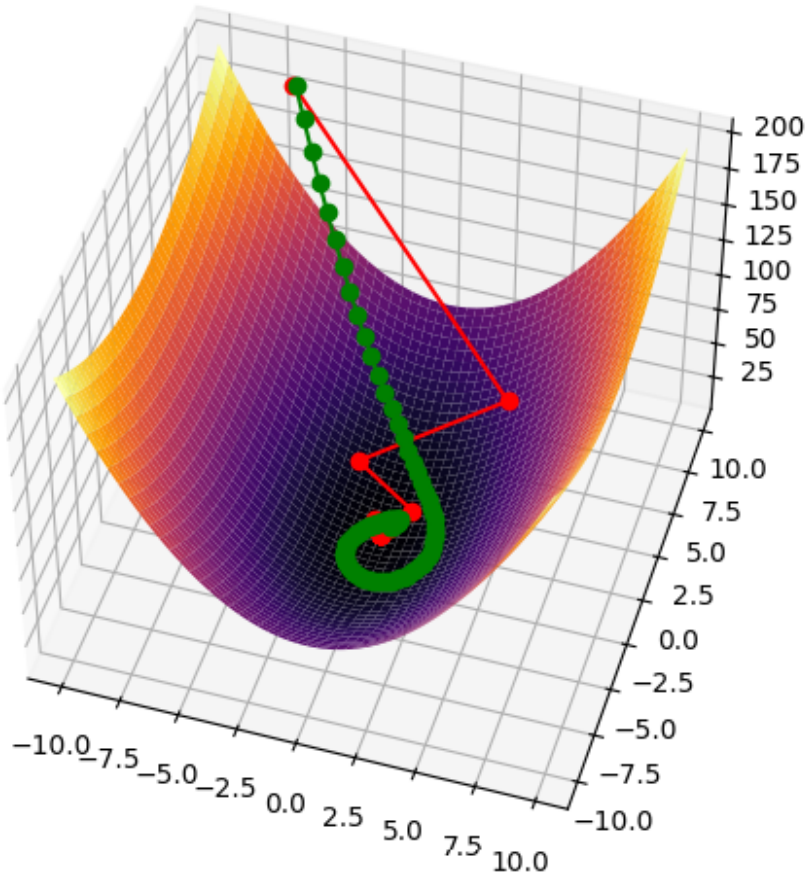

Рис. 11: сравнение ADAM с GD, $\alpha = 0.5$

Однако у ADAM есть большое преимущество, которое перекрывает скорость, особенно в сложных практических задачах:

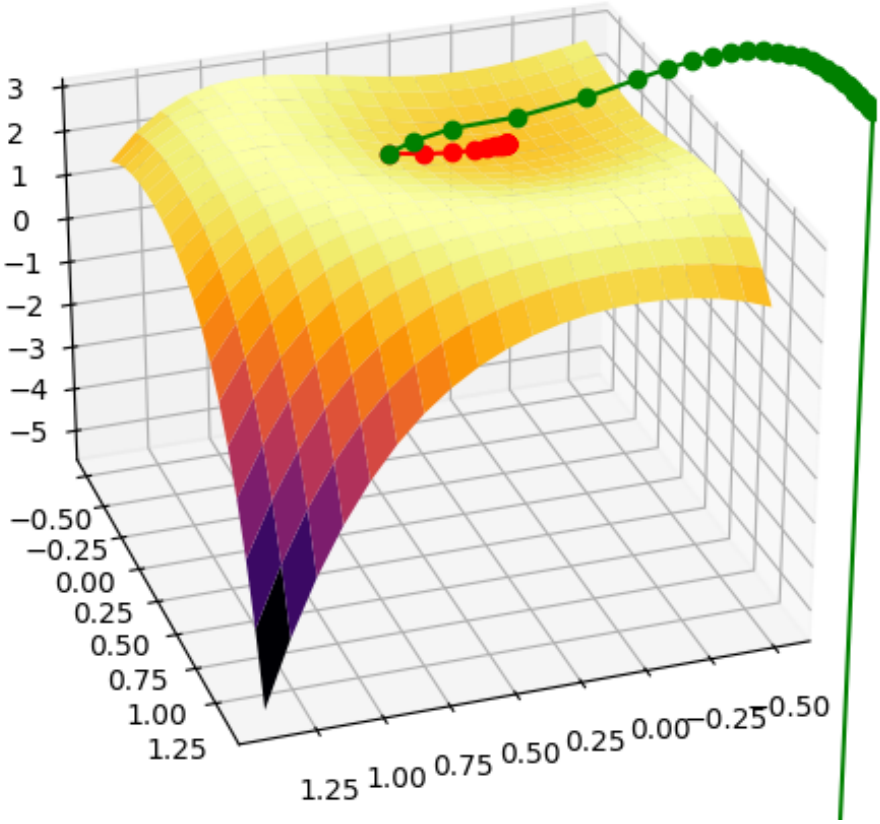

Рис. 12: выход из локального минимума

Легко заметить, что функция, минимум которой предстояло найти, имеет локальный оптимум в ”ямке”, а глобальный - находится где-то далеко за ней. Именно ADAM может ”перепрыгнуть ямку” и отправиться на поиски глобального минимума. Такую возможность ему дарят скользящие средние, которые были описаны в теоретической справке, именно они придают импульс текущему решению.

Кроме того, ADAM является усовершенствованием метода импульсов и RMSProp, поэтому его сходимость быстрее, например для метода импульсов:

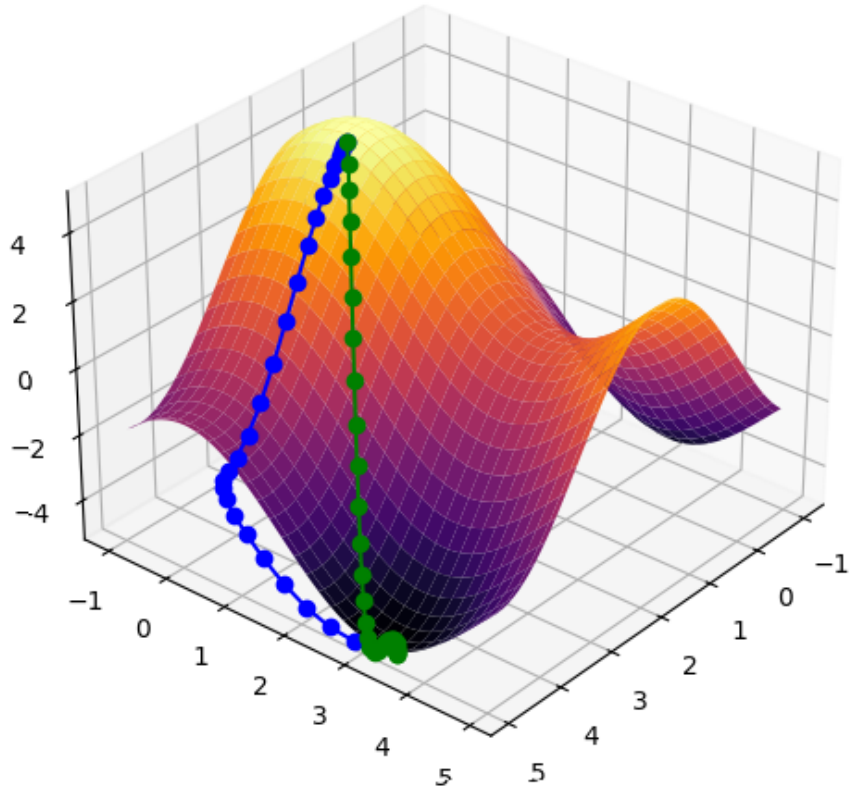

Рис. 13: синим отмечен метод импульсов, зеленым - ADAM

Недостатки ADAM частично решаются рядом его модификаций, которые выходят за рамки этой работы.

2.3.4 Метод SQN на практике

Пожалуй, самым быстрым алгоритмом для общих задач является метод SQN:

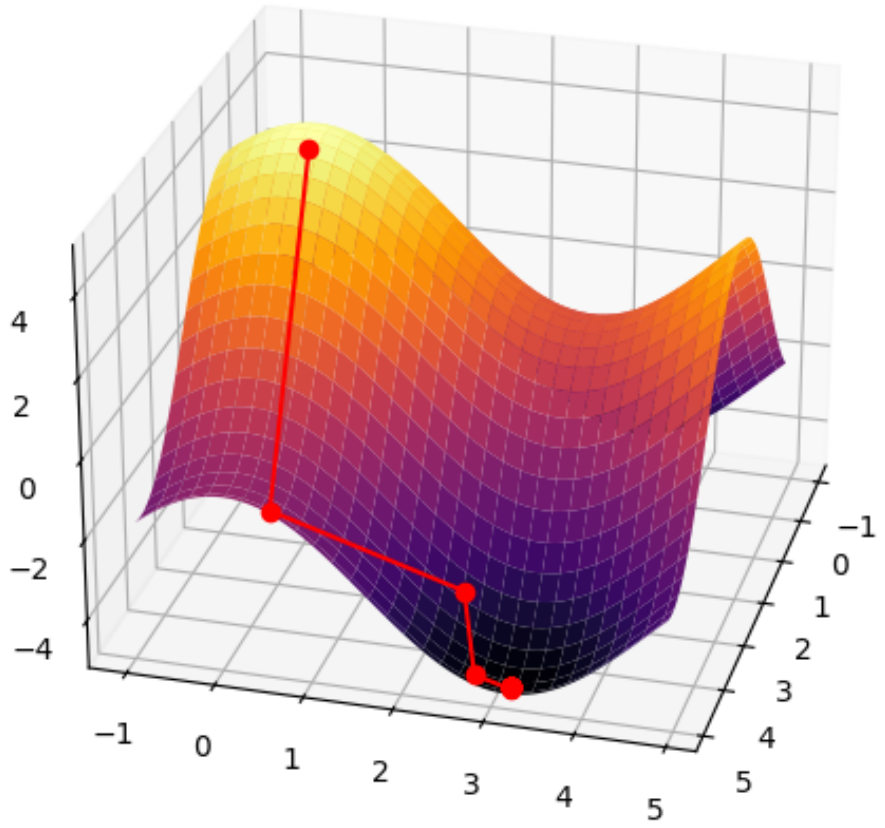

Рис. 14: метод SQN на синусоидальной функции

Это не удивительно, так как он напрямую использует матрицу Гессе, учитывая выпуклости функции.

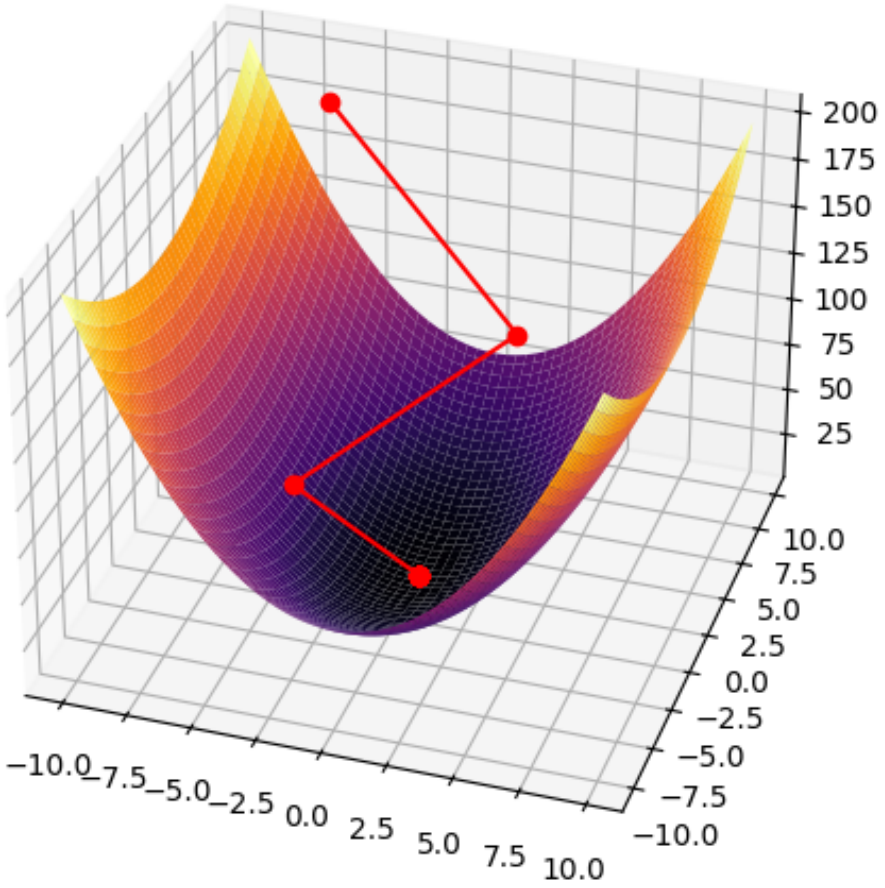

Рис. 15: метод SQN на квадратичной функции

Но проблема SQN в фактической непригодности к таким задачам, как машинное обучение, где получение второй производной от целевой функции с множеством параметров вычислительно нецелесообразно. Поэтому на практике используют некоторые реализации SQN, не требующие вычисления матрицы Гессе, например, алгоритм BFGS:

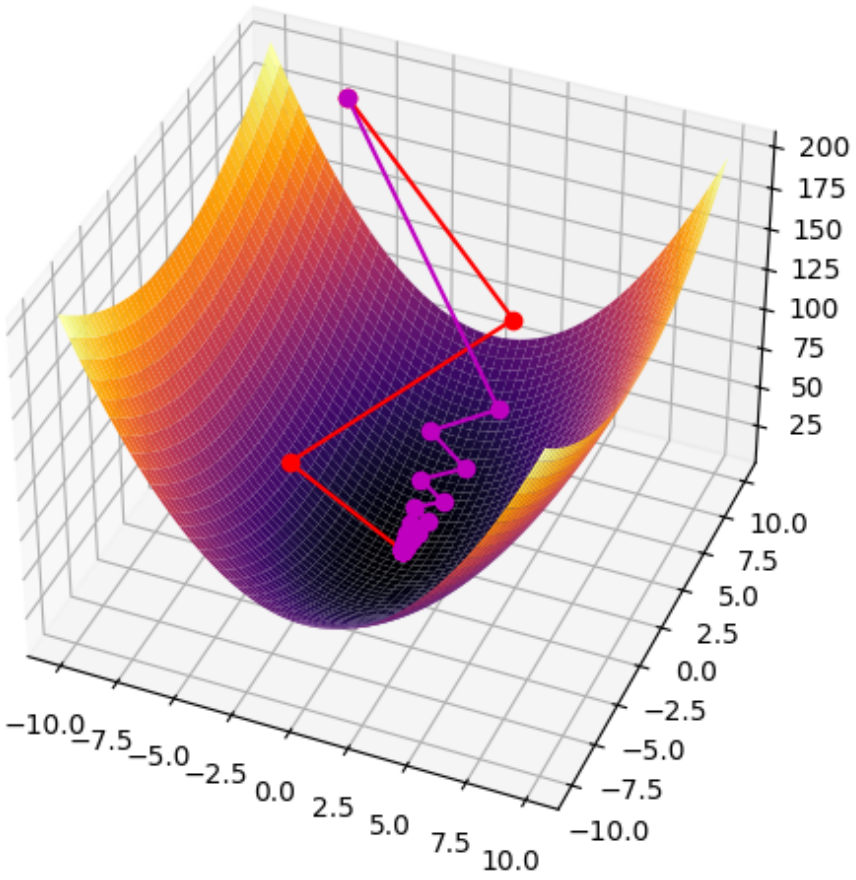

Рис. 16: сравнение SQN и BFGS. SQN отмечен красным, BFGS - фиолетовым

Очевидно, что так как BFGS использует приближение к матрице Гессе, вычисляемое итеративно вместе с вычислением приближений к минимуму функции, он намного медленнее обычного SQN. Но, разумеется, этот недостаток исключает необходимость вычислять матрицу Гессе для целевой функции.

2.3.5 Сравнение временных параметров

Все реализованные алгоритмы были запущены для решения задачи минимизации функции эллиптического параболоида. Были подсчитаны время выполнения, количество затраченных шагов:

```
Running CGDOptimizer. Accuracy: 0.05
        Total steps: 6
        Solution: 5.000611318856555
        Requested time: 0.0010004043579101562

================

================

Running MomentumOptimizer. Accuracy: 0.05
Momentum rate: 0.9
Use static step: 0.2
        Total steps: 83
        Solution: 5.000071915011181
        Requested time: 0.0019989013671875

================

================

Running AdamOptimizer. Accuracy: 0.05
beta1: 0.95, beta2: 0.999.
Use static step: 0.2
        Total steps: 156
        Solution: 5.000827505068188
        Requested time: 0.0065114498138427734

================

================

Running SQNOptimizer. Accuracy: 0.05
        Total steps: 3
        Solution: 5.0
        Requested time: 0.26653170585632324

================

================

Running BFGSOptimizer. Accuracy: 0.05
        Total steps: 25
        Solution: 5.000675700528978
        Requested time: 0.0029981136322021484

================
```

Рис. 17: сравнение временных параметров при минимизации функции параболоида

В результате лучшие показатели по времени продемонстрировал метод сопряженных градиентов.

Несмотря на то, что количество шагов меньше всего у SQN, а точность решения - лучше чем у остальных алгоритмов, затраченное им время оказалось слишком большим. Поэтому SQN крайне сложен и требует большие вычислительные ресурсы. Особенно разница заметна на фоне

BFGS, который использует те же идеи, но вычисляет приближение к матрице Гессе.

Оптимизатор ADAM оказался самым медленным, не считая SQN, однако данная задача и не могла отразить его преимущества.

Аналогичный тест был запущен для функции $f(z) = 3sin(x) + 2cos(y)$:

```
Running CGDOptimizer. Accuracy: 0.05
        Total steps: 8
        Solution: -4.999667658842872
        Requested time: 0.0009992122650146484

================

================

Running MomentumOptimizer. Accuracy: 0.05
Momentum rate: 0.9
Use static step: 0.2
        Total steps: 90
        Solution: -4.999668551555834
        Requested time: 0.0019986629486083984

================

================

Running AdamOptimizer. Accuracy: 0.05
beta1: 0.95, beta2: 0.999.
Use static step: 0.2
        Total steps: 124
        Solution: -4.999535685845273
        Requested time: 0.0039975643157958984

================

================

Running SQNOptimizer. Accuracy: 0.05
        Total steps: 5
        Solution: -4.99994619461428
        Requested time: 0.24986958503723145

================

================

Running BFGSOptimizer. Accuracy: 0.05
        Total steps: 7
        Solution: -4.999756598837244
        Requested time: 0.0010027885437011719

================
```

Рис. 18: сравнение временных параметров при минимизации синусоидальной функции

В этот раз были получены схожие результаты. Стоит отметить, что метод сопряженных градиентов, потерял своё преимущество, но остался крайне эффективным.

Для ADAM очевидным остаётся то, что его параметры должны быть сконфигурированы более точным образом для получения лучших результатов.

2.3.6 Вывод

Самым оптимальным вариантом для решения оптимизационных задач можно считать алгоритмы BFGS и метод сопряженных градиентов.

Алгоритм ADAM лучше использовать для задач общего назначения, где есть локальные минимумы. Также ADAM необходимо конфигурировать для конкретной оптимизационной задачи, чтобы ускорить сходимость.

SQN, хоть и поразительно точен, остаётся крайне затратным.

3 ЗАКЛЮЧЕНИЕ

В работе были рассмотрены градиентные алгоритмы ADAM, SQN, SARAH, метод сопряженных градиентов, а также их аналоги. Теперь, основываясь на теоретических и эмпирических знаниях, можно сделать следующие выводы:

- Алгоритм сопряженных градиентов является лучшим выбором для нахождения экстремумов квадратичных функций, заданных квадратной, симметричной, положительно–определенной матрицей A;
- Алгоритм ADAM является хорошим выбором для задач общего назначения, особенно там, где целевая функция сложна и имеет множество локальных минимумов;
- Алгоритм SQN является одним из самых быстрых градиентных алгоритмов, но требует вычисление матрицы Гессе, поэтому его использование в практических задачах ограничего. Однако многие реализации SQN являются достаточно быстрыми и могут использоваться в задачах общего назначения;
- Алгоритм SARAH является хорошим выбором для задач машинного обучения, он имеет хорошую сходимость, вычислительно не затратен и не требует использование большого количества памяти.

Подводя итог, можно сказать, что градиентные методы - очень мощный инструмент, который может решать множество прикладных задач от машинного обучения до экономических задач.

Градиентные алгоритмы не используют много памяти, они итеративны, вычисления во многих алгоритмах крайне просты и интуитивны. Существует большое количество реализаций для самых узкоспециализированных задач. Градиентные алгоритмы легко проектировать стохастическими, что находит своё применение в задачах с большим количеством данных.

Однако градиентные алгоритмы не лишены недостатков: самые эффективные алгоритмы являются методами второго порядка, что ограничивает возможность их использования в прикладных задачах. Алгоритмы, которые используются на практике, часто являются эвристиками,

не имеющими под собой научной базы. Их использование обусловлено тем, что они быстрее простейших алгоритмов, но их инициализация проста и не накладывает ограничений.

Тем не менее, градиентные алгоритмы остаются одним из главных инструментов оптимизации, создаются новые методы с улучшенной сходимостью. Всё это делает градиентные алгоритмы перспективным направлением для изучения.

Список литературы

[1] Rahul Agarwal. *Complete Guide to the Adam Optimization Algorithm*. URL: https://builtin.com/machine-learning/adam-optimization.

[2] Глоссарий DeepAI. *Understanding RMSProp: An Adaptive Learning Rate Method*. URL: https://deepai.org/machine-learning-glossary-and-terms/rmsprop.

[3] K. Scheinberg L. Nguyen J. Liu. “SARAH: A Novel Method for Machine Learning Problems Using Stochastic Recursive Gradient”. B: (2017), с. 9.

[4] Cong-Ying Han Tian-De Guo Yan Liu. “An Overview of Stochastic Quasi-Newton Methods for Large-Scale Machine Learning”. B: *Springer* (2023), с. 275. DOI: https://link.springer.com/article/10.1007/s40305-023-00453-9.

[5] Wikipedia. *Backpropagation*. URL: https://en.wikipedia.org/wiki/Backpropagation.

[6] Wikipedia. *Moving average*. URL: https://en.wikipedia.org/wiki/Moving_average.

[7] Wikipedia. *Wolfe Conditions*. URL: https://en.wikipedia.org/wiki/Wolfe_conditions.

[8] Онлайн школа Калифорнийского университета Беркли. *What Is Machine Learning (ML)?* URL: https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/.

[9] Николай Некипелов. *Метод сопряженных градиентов — математический аппарат*. URL: https://basegroup.ru/community/articles/conjugate.

4 Приложение А - листинги программного решения

Управляющий файл main.py:

```
import numpy as np

# graphics
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# target functions
import targetfunctions as tfs

# optimizers
import optimizers as opts

# drawers
import drawers


class Main:
    """
    Управляющий класс. Выполняет задачу создания целевых функций, их оптимизаторов, рисовщиков графиков.

    Для использования необходимо инициализировать класс с параметрами initialX – начальное приближение
    и необязательными startX=-10.0 правая( граница отрисовки графика), endX=10.0 левая( граница отрисовки графика),
    step=0.1 шаг( построения графика).

    Затем необходимо инициализировать необходимую функцию с помощью методов типа init<Required Function>.
    Может быть выбрана лишь одна функция.
```

```
27
28      Далее следует добавить оптимизаторы с помощью методов типа add<Required
        Optimizer>, которых может быть сколь угодно много.
29      После этого можно выполнить функцию drawSolutions() для отрисовки
        решения оптимизаторов. Все оптимизаторы начинают в одной точке.
30
31      Отрисовка решений возможна только для трехмерных функций. В противном
        случае возникнет исключение.
32      """
33
34      def __init__(self, target_function, initial_x, start_x=-10.0, end_x=10.0,
         step=0.1):
35          # генерируем трехмерную систему координат
36          self.fig = plt.figure()
37          self.ax_3d = self.fig.add_subplot(projection='3d')
38          self.optimizers = []
39          self.optimizers_drawers = []
40
41          self.initial_x = initial_x
42          self.start_x = start_x
43          self.end_x = end_x
44          self.step = step
45
46          self.target_function = target_function
47          drawers.GraphicDrawer(self.target_function, self.fig, self.ax_3d,
        self.start_x, self.end_x, self.step).draw()
48
49      def add_gd_optimizer(self, linestyle="ro-", alpha=None):
50          """
51          Вызовите этот метод, чтобы добавить оптимизатор с использованием
        наискорейшего спуска.
52          Укажите тип линии для рисовщика решения, например по умолчанию
        установлена красная сплошная линия: "ro-"
```

```
53          """
54          gd_optimizer = opts.GDOptimizer(self.target_function, initial_x=self.
    initial_x, alpha=alpha)
55          gd_optimizer_drawer = drawers.OptimizerDrawer(gd_optimizer, self.
    ax_3d, linestyle=linestyle)
56
57          self.optimizers.append(gd_optimizer)
58          self.optimizers_drawers.append(gd_optimizer_drawer)
59
60      def add_cgd_optimizer(self, linestyle="bo-"):
61          """
62          Вызовите этот метод, чтобы добавить оптимизатор с использованием
    метода сопряженных градиентов.
63          Укажите тип линии для рисовщика решения, например по умолчанию
    установлена синяя сплошная линия: "bo-"
64          """
65          cgd_optimizer = opts.CGDOptimizer(self.target_function, initial_x=
    self.initial_x)
66          cgd_optimizer_drawer = drawers.OptimizerDrawer(cgd_optimizer, self.
    ax_3d, linestyle=linestyle)
67
68          self.optimizers.append(cgd_optimizer)
69          self.optimizers_drawers.append(cgd_optimizer_drawer)
70
71      def add_adam_optimizer(self, linestyle="go-", alpha=0.05, beta1=0.9,
    beta2=0.999, epsilon=1e-8):
72          """
73          Вызовите этот метод, чтобы добавить оптимизатор с использованием
    метода ADAM.
74          Укажите параметры betta1, betta2 и epsilon поумолчанию(- - beta1=0.9,
     beta2=0.999, epsilon=1e-8).
75          О предназначении параметров см. optimizers.AdamOptimizer
```

```
76          Укажите тип линии для рисовщика решения, например по умолчанию установлена зеленая сплошная линия: "go-"
77          """
78          adam_optimizer = opts.AdamOptimizer(self.target_function, initial_x=self.initial_x, alpha=alpha, beta1=beta1,
79                                              beta2=beta2, epsilon=epsilon)
80
81          adam_optimizer_drawer = drawers.OptimizerDrawer(adam_optimizer, self.ax_3d, linestyle=linestyle)
82
83          self.optimizers.append(adam_optimizer)
84          self.optimizers_drawers.append(adam_optimizer_drawer)
85
86      def add_momentum_optimizer(self, linestyle="bo-", alpha=0.05, betta=0.9):
87          """
88          Вызовите этот метод, чтобы добавить оптимизатор с использованием метода импульсов.
89          Укажите параметр betta.
90          О предназначении параметров см. optimizers.MomentumOptimizer
91          Укажите тип линии для рисовщика решения, например по умолчанию установлена синяя сплошная линия: "bo-"
92          """
93          momentum_optimizer = opts.MomentumOptimizer(self.target_function, initial_x=self.initial_x, momentum_rate=betta, alpha=alpha)
94
95          momentum_optimizer_drawer = drawers.OptimizerDrawer(
            momentum_optimizer, self.ax_3d, linestyle=linestyle)
96
97          self.optimizers.append(momentum_optimizer)
98          self.optimizers_drawers.append(momentum_optimizer_drawer)
99
100     def add_sqn_optimizer(self, linestyle="ro-"):
101         """
```

```
102         Вызовите этот метод, чтобы добавить оптимизатор с использованием алгоритма SQN.
103         Укажите тип линии для рисовщика решения, например по умолчанию установлена красная сплошная линия: "ro-"
104         """
105         newton_optimizer = opts.SQNOptimizer(self.target_function, initial_x=self.initial_x)
106         newton_optimizer_drawer = drawers.OptimizerDrawer(newton_optimizer, self.ax_3d, linestyle=linestyle)
107
108         self.optimizers.append(newton_optimizer)
109         self.optimizers_drawers.append(newton_optimizer_drawer)
110
111     def add_bfgs_optimizer(self, linestyle="mo-"):
112         """
113         Вызовите этот метод, чтобы добавить оптимизатор с использованием алгоритма BFGS.
114         Укажите тип линии для рисовщика решения, например по умолчанию установлена фиолетовая сплошная линия: "mo-"
115         """
116         newton_optimizer = opts.BFGSOptimizer(self.target_function, initial_x=self.initial_x)
117         newton_optimizer_drawer = drawers.OptimizerDrawer(newton_optimizer, self.ax_3d, linestyle=linestyle)
118
119         self.optimizers.append(newton_optimizer)
120         self.optimizers_drawers.append(newton_optimizer_drawer)
121
122     def find_solutions(self):
123         for i in range(len(self.optimizers)):
124             self.optimizers[i].find_solution()
125
126     def draw_solutions(self):
```

```
127         anis = []
128 
129         for i in range(len(self.optimizers_drawers)):
130             anis.append(
131                 FuncAnimation(self.fig, self.optimizers_drawers[i].
    draw_solution, frames=np.arange(0, 100), blit=True,
132                               repeat=False))
133 
134         plt.tight_layout()
135         plt.show()
136 
137 
138 def init_paraboloid(A, b, c):
139     """
140     Вызовите этот метод, чтобы инициализировать эллиптический параболоид.
141     """
142     return tfs.EllipticalParaboloid(A, b, c)
143 
144 
145 def init_oscillator():
146     """
147     Вызовите этот метод, чтобы инициализировать двумерную осциллирующую
    функцию.
148     """
149     return tfs.OscillatoryBivariateFunction()
150 
151 
152 def init_unimod():
153     """
154     Вызовите этот метод, чтобы инициализировать унимодальную функцию с ямкой
    для( демонстрации методов с импульсами).
155     """
156     return tfs.UnimodalPitFunction()
```

```
157
158
159 if __name__ == "__main__":
160     # example with paraboloid
161     # main = Main(init_paraboloid(A=[[3, 0], [0, 1]], b=[0, 0], c=5), [-7.5,
    12])
162
163     # example with oscillator
164     main = Main(init_oscillator(), [1.9, 0.1], -1.0, 5.0)
165
166     # example with unimod
167     # main = Main(init_unimod(), [0.5, 0.10], -0.5, 1.5)
168
169     main.add_gd_optimizer(alpha=0.2)
170     main.add_cgd_optimizer()
171     main.add_momentum_optimizer(alpha=0.2)
172     main.add_adam_optimizer(alpha=0.5, beta1=0.95, beta2=0.999)
173     main.add_sqn_optimizer()
174     main.add_bfgs_optimizer()
175
176     # testing solutions:
177     main.find_solutions()
178
179     # main.draw_solutions()
```

Целевые функции targetfunctions.py:

```
1 import numpy as np
2
3
4 class TargetFunction:
5     """
6     Данный класс представляет обобщение для всех многомерных функций.
7
```

```
8       Класс оперирует вектором x, где каждый элемент вектора – переменная, по
    которой строится функция.
9
10      Для методов второго порядка определена функция
    secondOrderDifferentialInDot, возвращающая значение
11      второй производной функции в точке с координатами x.
12      """
13
14      def __call__(self, x):
15          pass
16
17      def differential_in_dot(self, x):
18          """
19          Для применения градиентных методов в классах, унаследованных от
    TargetFunction, должен
20          быть определен метод differentialInDot, возвращающий значение
    производной функции в точке с координатами x.
21          """
22          pass
23
24      def second_order_differential_in_dot(self, x):
25          """
26          Для применения методов второго порядка в классах, унаследованных от
    TargetFunction, должен быть определен
27          метод secondOrderDifferentialInDot, возвращающий значение второй
    производной функции в точке с координатами x.
28          """
29          pass
30
31
32  class EllipticalParaboloid(TargetFunction):
33      """
34      Данный класс представляет эллиптический параболоид заданный в общем виде:
```

```
35          f(x) = xT * A * x - bT * x + c
36      где A - квадратная матрица размером nxn, x - вектор переменных размером 1xn, b и c - вектора размером 1xn
37      """
38
39      def __init__(self, A, b, c):
40          self.b = np.array(b)
41          self.c = c
42          self.A = np.array(A)
43
44      def __call__(self, x):
45          return (1 / 2) * np.array(x).T.dot(self.A).dot(x) - self.b.dot(x) + self.c
46
47      # differential is: xT * A - b
48      def differential_in_dot(self, x):
49          return np.dot(x, self.A) - self.b
50
51      # second order differential is: A
52      def second_order_differential_in_dot(self, x):
53          return self.A
54
55
56  class OscillatoryBivariateFunction(TargetFunction):
57      """
58      Данный класс представляет двумерную осциллирующую функцию:
59          3 * sin(x0) + 2 * cos(x1)
60      где x - вектор переменных, в котором учитываются только первая и вторая (x0 и x1)
61      """
62
63      def __call__(self, x):
64          return 2 * np.cos(x[1]) + 3 * np.sin(x[0])
```

```
65
66      # differential in x0: 3 * cos(x0); in x1: - 2 * sin(x1)
67      def differential_in_dot(self, x):
68          return np.array([3 * np.cos(x[0]), - 2 * np.sin(x[1])])
69
70      # second order differential in x0: [3, 0]; in x1: [0, 2]
71      def second_order_differential_in_dot(self, x):
72          return np.array([[3, 0], [0, 2]])
73
74
75  def count_squares_sum(x):
76      return x[0] * x[0] + x[1] * x[1]
77
78
79  class UnimodalPitFunction(TargetFunction):
80      """
81      Данный класс представляет двумерную функцию вида:
82          - (x0^2 + x1^2)^2 + 2 * (x0^2 + x1^2) + 2
83      где x - вектор переменных, в котором учитываются только первая и вторая (
        x0 и x1)
84      """
85
86      def __call__(self, x):
87          return -(x[0]**2 + x[1]**2)**2 + 2 * (x[0]**2 + x[1]**2) + 2
88
89      # differential in x0: -4 * x0 * (x0^2 + x1^2) + 4 * x0;
90      # in x1: -4 * x1 * (x0^2 + x1^2) + 4 * x1
91      def differential_in_dot(self, x):
92          return np.array([-4 * x[0] * (x[0]**2 + x[1]**2) + 4 * x[0],
93                           -4 * x[1] * (x[0]**2 + x[1]**2) + 4 * x[1]])
94
95      # second order differential in x0: [-12 * x0^2 + 12 * (x0^2 + x1^2) + 4,
        -8 * x0 * x1];
```

```
    # in x1: [-8 * x0 * x1, -12 * x1^2 + 12 * (x0^2 + x1^2) + 4]
    def second_order_differential_in_dot(self, x):
        return np.array([[12 * x[0]**2 + 4, -8 * x[0] * x[1]],
                         [-8 * x[0] * x[1], 12 * x[1]**2 + 4]])
```

Файл с рисовщиком решения drawers.py:

```
import numpy as np

from targetfunctions import TargetFunction
from optimizers import Optimizer


class GraphicDrawer:
    """
    Данный класс предназначен для отрисовки целевой функции.

    Но класс не отрисовывает функцию при вызове draw(), а лишь
    добавляет её к уже имеющейся системе координат ax_3d.

    Кроме того, класс работает только с трехмерными функциями
    """

    def __init__(self, func: TargetFunction, fig, ax_3d, start_x, end_x, step
    ):
        """
        Укажите целевую функцию типа TargetFunction, fig из библиотеки
    matplotlib,
        трехмерную систему координат, в которой будет работать данный класс,
    а также
        startX правая( граница отрисовки графика), endX левая( граница
    отрисовки графика),
        step шаг( построения графика).
        """
```

```
24          self.targetFunction = func
25
26          self.fig = fig
27          self.ax_3d = ax_3d
28
29          self.start_x = start_x
30          self.end_x = end_x
31          self.step = step
32
33      def draw(self):
34          x1 = np.arange(self.start_x, self.end_x, self.step)
35          x2 = np.arange(self.start_x, self.end_x, self.step)
36
37          x1grid, x2grid = np.meshgrid(x1, x2)
38
39          data_f = np.empty_like(x1grid)
40
41          for i in range(x1grid.shape[0]):
42              for j in range(x1grid.shape[1]):
43                  data_f[i, j] = self.targetFunction([x1grid[i, j], x2grid[i, j
    ]])
44
45          self.ax_3d.plot_surface(x1grid, x2grid, data_f, cmap='inferno')
46
47
48  class OptimizerDrawer:
49      """
50      Данный класс предназначен для отрисовки решения оптимизатора по шагам.
51
52      Но класс не отрисовывает решение при вызове drawSolution(), а лишь
53      добавляет её к уже имеющейся системе координат ax_3d.
54      """
55
```

```
    def __init__(self, optimizer: Optimizer, ax_3d, linestyle):
        """
        Укажите оптимизатор типа Optimizer, трехмерную систему координат, в которой будет работать данный класс,
        а также стиль линии, которой должен отрисовываться график.
        """
        self.line, = ax_3d.plot([], [], [], linestyle)
        self.points_x = [optimizer.initial_x]
        self.points_y = [optimizer.target_function(optimizer.initial_x)]

        self.optimizer = optimizer

    def draw_solution(self, frame):
        x, y = self.optimizer.make_step()

        self.points_x.append(x)  # Добавляем текущую точку в список
        self.points_y.append(y)  # Добавляем текущую точку в список

        self.line.set_data(*zip(*self.points_x))  # Разделяем координаты точек для отрисовки линий
        self.line.set_3d_properties(self.points_y, 'z')
        return self.line,
```

Классы общего назначения util.py:

```
import numpy as np


def count_a_step(gradient, second_order_differential_in_dot_x):
    """
    Вычисляет динамический шаг градиента альфа()

    Шаг вычисляется по формуле:

```

```
    alpha = (gradient.T * gradient) / (gradient.T * (f''(x) * gradient))
    """

    det = (gradient.dot(second_order_differential_in_dot_x.dot(gradient)))

    if det == 0:
        return 0

    return (gradient.dot(gradient)) / (gradient.dot(
    second_order_differential_in_dot_x.dot(gradient)))

def transposeVector(vector):
    return vector[:, np.newaxis]
```

Реализации градиентных алгоритмов optimizers.py:

```
import numpy as np
import scipy as sp
import time
import random

from targetfunctions import TargetFunction
from util import count_a_step
from util import transposeVector


class Optimizer:
    """
    Данный класс представляет обобщение для всех оптимизаторов.

    Класс предназначен для нахождения минимума целевой функции из точки
    initialX.

    Поиск может осуществляться как целиком за раз функция( findSolution),
```

```
    так и по шагам
    функция( makeStep), что используется в рисовщиках алгоритмов минимизации.
    """

    def __init__(self, func: TargetFunction, initial_x):
        self.x = None
        self.target_function = func
        self.gradient = func.differential_in_dot(initial_x)

        self.initial_x = initial_x

    def find_solution(self, accuracy=0.05):
        """
        Функция предназначена для поиска минимума целевой функции (
    TargetFunction) из точки initialX,
        а также вывода количества шагов и потребовавшегося времени.
    Возвращает вектор точек [x, y]
        """

        print("=" * 16, "\n")

        print(f"Running {self.__class__.__name__}. Accuracy: {accuracy}")
        self.additional_info()

        step = 0
        y = 0.

        not_satisfied_accuracy = True
        start_time = time.time()

        while not_satisfied_accuracy:
            y = self.make_step()[-1]
            current_accuracy = np.linalg.norm(self.gradient)
```

```
48
49              if current_accuracy < accuracy:
50                  not_satisfied_accuracy = False
51
52              step += 1
53
54          end_time = time.time()
55
56          print(f"\tTotal steps: {step}")
57          print(f"\tSolution: {y}")
58          print(f"\tRequested time: {end_time - start_time}")
59
60          print()
61          print("=" * 16, "\n")
62
63          return [self.x, y]
64
65      def make_step(self):
66          """
67          Функция предназначена для выполнения одного шага поиска минимума
    целевой функции
68          (TargetFunction) из точки initialX, с учетом предыдущих шагов
69          """
70          raise NotImplementedError("make_step method must be implemented in
    child classes")
71
72      def additional_info(self):
73          """
74          Функция вывода дополнительной информации об оптимизаторе, например
    использование динамического шага,
75          специфические константы и др.
76          """
77          pass
```

```
78
79
80  class GDOptimizer(Optimizer):
81      """
82      Данный класс представляет простейший оптимизатор с использованием
        наискорешего спуска (Gradient Descent Optimizer).
83
84      Если в конструкторе задан статический шаг, то будет использоваться он. В
        противном случае будет высчитываться
85      динамический шаг по методу НьютонаРафсона- метод( второго порядка в
        данном случае)
86
87      На каждом шаге подсчитывается антиградиент в точке и динамический шаг, а
        затем с данным шагом
88      осуществляется переход по вектору x в направлении антиградиента.
89      """
90
91      def __init__(self, func: TargetFunction, initial_x, alpha):
92          super().__init__(func, initial_x)
93          self.x = self.initial_x
94          self.gradient = self.target_function.differential_in_dot(self.x)
95
96          # Булева переменная об использовании динамического шага
97          self.is_dynamic_step_used = False
98
99          if alpha is None:
100             self.is_dynamic_step_used = True
101         else:
102             self.alpha = alpha
103
104     def make_step(self):
105         # подсчитываем динамический шаг, если не указано, что его нужно
        считать статическим, по методу НьютонаРафсона-
```

```
106         if self.is_dynamic_step_used:
107             self.alpha = count_a_step(self.gradient, self.target_function.
    second_order_differential_in_dot(self.x))
108 
109         self.x = self.x - self.alpha * self.gradient
110         y = self.target_function(self.x)
111 
112         self.gradient = self.target_function.differential_in_dot(self.x)
113 
114         return [self.x, y]
115 
116 
117 class CGDOptimizer(Optimizer):
118     """
119     Данный класс представляет оптимизатор с использованием метода сопряженых
    градиентов (Conjugate Gradient Descent Optimizer).
120 
121     На каждом шаге подсчитывается антиградиент в точке и динамический шаг, а
    затем с данным шагом
122     осуществляется переход по вектору x в направлении антиградиента, далее
    высчитывается градиент
123     в новой точке, высчитывается коэффициент betta, учитывающий новый и
    старый градиенты, с его
124     использованием уточняется новый антиградиент (newAntiGradient = -
    newGradient + betta * oldAntiGradient).
125     """
126 
127     def __init__(self, func: TargetFunction, initial_x):
128         super().__init__(func, initial_x)
129         # Начальное приближение
130         self.x = self.initial_x
131 
132         # Номер итерации
```

```
133         self.n = 0
134 
135         # Градиент и антиградиент в точке начального приближения
136         self.gradient = self.target_function.differential_in_dot(self.x)
137 
138     def make_step(self):
139         # Высчитываем динамический шаг см(. util.count_a_step)
140         alpha = count_a_step(self.gradient, self.target_function.
    second_order_differential_in_dot(self.x))
141 
142         # Находим новое приближение
143         self.x = self.x - alpha * self.gradient
144         y = self.target_function(self.x)
145 
146         # Находим градиент в точке нового приближения
147         new_gradient = self.target_function.differential_in_dot(self.x)
148 
149         # Не на нулевой итерации вычисляем сопряженное направление,
    обновляем градиент
150         if self.n != 0:
151             # Знаменатель
152             det = self.gradient.dot(self.gradient)
153 
154             # Для избежания деления на 0
155             if det == 0:
156                 beta = 0
157             else:
158                 # Формула ФлетчераРивса-
159                 beta = new_gradient.dot(new_gradient) / det
160 
161                 # Сброс каждые n + 1 шагов
162                 if self.n % (self.x.shape[0] + 1) == 0:
163                     beta = 0
```

```
164
165             # Обновляем градиент
166             self.gradient = new_gradient + beta * self.gradient
167         else:
168             self.gradient = new_gradient
169
170         self.n += 1
171
172         return [self.x, y]
173
174
175 class MomentumOptimizer(Optimizer):
176     """
177     Данный класс представляет оптимизатор с использованием метода моментов
178
179     На каждом шаге подсчитывается антиградиент в точке и динамический шаг,
    затем вычисляется скользящее среднее,
180     содержащее информацию обо всех предыдущих градиентах. Далее вычисляется
    новое значение x с учетом скользящего
181     среднего
182
183     Если в конструкторе задан статический шаг, то будет использоваться он. В
    противном случае будет высчитываться
184     динамический шаг по методу НьютонаРафсона- метод( второго порядка в
    данном случае)
185     """
186
187     def __init__(self, func: TargetFunction, initial_x, momentum_rate, alpha)
    :
188         super().__init__(func, initial_x)
189
190         self.x = self.initial_x
191
```

```
192         self.k = 0
193 
194         self.gradient = self.target_function.differential_in_dot(self.x)
195 
196         # последнее скользящее среднее градиентов (moving_average)
197         self.last_ma = 0.
198         self.momentum_rate = momentum_rate
199 
200         # Булева переменная об использовании динамического шага
201         self.is_dynamic_step_used = False
202 
203         if alpha is None:
204             self.is_dynamic_step_used = True
205         else:
206             self.alpha = alpha
207 
208     def additional_info(self):
209         print(f"Momentum rate: {self.momentum_rate}")
210         if self.is_dynamic_step_used:
211             print("Use dynamic step.")
212         else:
213             print(f"Use static step: {self.alpha}")
214 
215     def make_step(self):
216         # подсчитываем динамический шаг, если не указано, что его нужно
    считать статическим, по методу НьютонаРафсона-
217         if self.is_dynamic_step_used:
218             self.alpha = count_a_step(self.gradient, self.target_function.
    second_order_differential_in_dot(self.x))
219 
220         # v = v_n-1 * betta + (1 - betta) * alpha * gradient
221         self.last_ma = (self.momentum_rate * self.last_ma + (1 - self.
    momentum_rate) * self.alpha * self.gradient)
```

```
222
223         # x = x - v
224         self.x = self.x - self.last_ma
225
226         y = self.target_function(self.x)
227
228         self.gradient = self.target_function.differential_in_dot(self.x)
229
230         return [self.x, y]
231
232
233 class AdamOptimizer(Optimizer):
234     """
235     Данный класс представляет оптимизатор с использованием метода ADAM (
    Adaptive Moment Estimation)
236
237     На каждом шаге подсчитывается градиент в точке и динамический шаг (alpha)
    . Затем вычисляются скользящее среднее
238     градиентов с( учетом параметра betta1 - коэффициент затухания для
    скользящего среднего градиентов) и скользящее
239     среднее квадратов градиентов с( учетом параметра betta2 - коэффициента
    затухания для скользящего среднего квадратов
240     градиентов). Далее эти переменные нормализуются с учетом номера итерации
    алгоритма получаем( vn и Gn).
241     Далее происходит вычисление обновленного значения для x по формуле x = x
    - alpha * (vn / (sqrt(Gn) + epsilon)).
242     epsilon используется для избежания деления на 0. Далее вычисляется
    целевая функция (y) и алгоритм повторяется.
243
244     Параметры по умолчанию: betta1=0.9, betta2=0.999, epsilon=1e-8
245
246
247     Если в конструкторе задан статический шаг, то будет использоваться он. В
```

```
        противном случае будет высчитываться
248      динамический шаг по методу НьютонаРафсона- метод( второго порядка в
        данном случае)
249      """
250
251      def __init__(self, func: TargetFunction, initial_x, alpha, beta1=0.9,
        beta2=0.999, epsilon=1e-8):
252          super().__init__(func, initial_x)
253
254          self.x = self.initial_x
255
256          self.k = 0
257
258          self.gradient = self.target_function.differential_in_dot(self.x)
259
260          # Булева переменная об использовании динамического шага
261          self.is_dynamic_step_used = False
262
263          if alpha is None:
264              self.is_dynamic_step_used = True
265          else:
266              self.alpha = alpha
267
268          # последнее скользящее среднее градиентов (moving_average)
269          self.last_ma = 0.
270          # последнее скользящее среднее квадратов градиентов (
        moving_average_of_squares)
271          self.last_mas = 0.
272
273          # коэффициент для скользящего среднего
274          self.beta1 = beta1
275
276          # коэффициент для скользящего среднего квадратов градиентов
```

```
277         self.beta2 = beta2
278 
279         # слагаемое для избежания деления на 0
280         self.epsilon = epsilon
281 
282     def additional_info(self):
283         print(f"beta1: {self.beta1}, beta2: {self.beta2}.")
284         if self.is_dynamic_step_used:
285             print("Use dynamic step.")
286         else:
287             print(f"Use static step: {self.alpha}")
288 
289     def make_step(self):
290         # подсчитываем динамический шаг, если не указано, что его нужно
    считать статическим, по методу НьютонаРафсона-
291         if self.is_dynamic_step_used:
292             self.alpha = count_a_step(self.gradient, self.target_function.
    second_order_differential_in_dot(self.x))
293 
294         # высчитываем скользящее среднее аналогично методу импульсов
295         # v = v_n-1 * betta + (1 - betta) * alpha * gradient
296         self.last_ma = self.beta1 * self.last_ma + (1 - self.beta1) * self.
    gradient
297 
298         # высчитываем скользящее среднее квадратов градиентов аналогично
    методу RMSProp
299         # v = v_n-1 * betta + (1 - betta) * alpha * gradient ^ 2
300         self.last_mas = self.beta2 * self.last_mas + (1 - self.beta2) * (self
    .gradient ** 2)
301 
302         # Выполняем нормировку скользящих средних (v = v / (1 - betta^(k+1)))
    . С ростом числа проделанных шагов
303         # значение нормализованных величин будет уменьшаться.
```

```
304         normalized_ma = self.last_ma / (1 - np.power(self.beta1, self.k + 1))
305         normalized_mas = self.last_mas / (1 - np.power(self.beta2, self.k +
    1))
306 
307         # x = x - alpha * (v / (sqrt(G) + eps))
308         self.x = self.x - self.alpha * (normalized_ma / (np.sqrt(
    normalized_mas) + self.epsilon))
309 
310         y = self.target_function(self.x)
311 
312         self.gradient = self.target_function.differential_in_dot(self.x)
313 
314         # увеличиваем счетчик шагов
315         self.k += 1
316 
317         return [self.x, y]
318 
319 
320 class SQNOptimizer(Optimizer):
321     """
322     Данный класс представляет оптимизатор SQN.
323 
324     На каждом шаге подсчитывается антиградиент в точке, гессиан в точке,
    динамический шаг, а затем с данным шагом
325     осуществляется переход по вектору x в направлении антиградиента,
    умноженном на гессиан.
326     """
327 
328     def __init__(self, func: TargetFunction, initial_x):
329         super().__init__(func, initial_x)
330         self.x = self.initial_x
331         self.gradient = self.target_function.differential_in_dot(self.x)
332 
```

```
333     def make_step(self):
334         hessian = self.target_function.second_order_differential_in_dot(self.
        x)
335 
336         # высчитываем направление для движения вдоль антиградиента как
        матричное умножение якобиана на градиент
337         direction = -np.dot(hessian, self.gradient)
338 
339         # выполняем линейный поиск константы альфа шага( вдоль направления)
        для удовлетворения условиям Вольфа
340         # фактически, мы ищем такой шаг альфа, при котором целевая функция в
        точке следующего шага будет минимальна
341         line_search = sp.optimize.line_search(self.target_function, self.
        target_function.differential_in_dot,
342                                               self.x, direction)
343         alpha = line_search[0]
344 
345         if alpha is None:
346             alpha = 0.0
347 
348         self.x = self.x + alpha * direction
349         y = self.target_function(self.x)
350 
351         self.gradient = self.target_function.differential_in_dot(self.x)
352 
353         return [self.x, y]
354 
355 
356 class BFGSOptimizer(Optimizer):
357     """
358     Данный класс представляет оптимизатор с использованием метода SQN (
        stochastic quasi-Newton), а именно - BFGS.
359 
```

```
360     На каждом шаге подсчитывается новое направление, динамический шаг,
    приближение к матрице Гессе.
361     """
362
363     def __init__(self, func: TargetFunction, initial_x):
364         super().__init__(func, initial_x)
365
366         self.x = self.initial_x
367
368         self.k = 0
369
370         self.gradient = self.target_function.differential_in_dot(self.x)
371
372         # Задаём начальный якобиан в виде единичной матрицы
373         self.identity_matrix = np.identity(len(self.x), dtype=float)
374         self.H = self.identity_matrix
375
376         # Изменение по аргументам функции
377         self.x_delta = np.zeros(len(self.x), dtype=float)
378
379         # Изменение по градиентам
380         self.gradient_delta = np.zeros(len(self.x), dtype=float)
381
382     def make_step(self):
383         # высчитываем направление для движения вдоль антиградиента как
    матричное умножение якобиана на градиент
384         direction = -np.dot(self.H, self.gradient)
385
386         # запоминаем текущие градиенты и вектор x
387         last_x = self.x
388         last_gradient = self.gradient
389
390         # выполняем линейный поиск константы альфа шага( вдоль направления)
```

```
для удовлетворения условиям Вольфа
391         # фактически, мы ищем такой шаг альфа, при котором целевая функция в
точке следующего шага будет минимальна
392         line_search = sp.optimize.line_search(self.target_function, self.
target_function.differential_in_dot,
393                                               self.x, direction)
394         alpha = line_search[0]
395 
396         if alpha is None:
397             alpha = 0.0
398 
399         # движемся вдоль полученного направления с вычисленным шагом
400         self.x = self.x + alpha * direction
401 
402         # вычисляем значение функции в новой точке и значение градиента в
новой точке
403         y = self.target_function(self.x)
404         self.gradient = self.target_function.differential_in_dot(self.x)
405 
406         # обновляем изменение x и градиента (x_delta = x_k+1 - x_k; (g_delta
= g_k+1 - g_k))
407         self.x_delta = self.x - last_x  # также обозначаем как sk
408         self.gradient_delta = self.gradient - last_gradient  # также
обозначаем как yk
409 
410         # вычисляем константу ро
411         ro = 1.0 / np.dot(self.gradient_delta, self.x_delta)
412 
413         # ----------------------------------------------------a1
------------------------a2
414         # выполняем обновление якобиана по формуле: H = (I - ro * ykT * sk) *
 H * (I - ro * skT * yk) + ro * skT * sk
415         a1 = self.identity_matrix - ro * transposeVector(self.gradient_delta)
```

```
 * self.x_delta
416         a2 = self.identity_matrix - ro * transposeVector(self.x_delta) * self
    .gradient_delta
417 
418         self.H = np.dot(a1, np.dot(self.H, a2)) + (ro * transposeVector(self.
    gradient_delta) * self.gradient_delta)
419 
420         self.k += 1
421 
422         return [self.x, y]
```