Выполнил: А. Зеленцов, 201 группа

Стохастические методы оптимизации.

План работы.

- Постановка задачи
- Теоретическое описание найденных методов
- Практическая реализация описанных выше методов на Python
- Практическое применение методов к разного рода задачам оптимизации
- Вывод по каждому из методов

Постановка задачи

Общая постановка была такой:

В современном машинном обучении типично приходится решать задачи оптимизации вида сумма по объектам обучающей выборки, причём количество этих объектов (= слагаемых в сумме) может быть очень большим. Поэтому использование стандартных методов оптимизации типа градиентного спуска и его аналогов, когда на каждой итерации оптимизации нужно вычислять точное значение градиента функции, является затруднительным в силу вычислительной трудоёмкости. Соответственно, ваша задача самостоятельно найти, какие стохастические методы оптимизации существуют, и далее их подробно описать в тексте реферата вместе со всеми необходимыми формулами и пояснениями, откуда они берутся.

Для начала нужно математически описать, что такое задача оптимизации в рассматриваемом случае.

Имеется:

1. Допустимое множество $\mathbb{X} \subseteq \mathbb{R}^n$
2. Функционал $f(X) : \mathbb{X} \to \mathbb{R}$

Задача:

Найти такой $X^* \in \mathbb{X}$, что $f(X^*) = min_{X \in \mathbb{X}} f(X)$ либо доказать, что такового не существует.

Замечания:

Часто описанную выше задачу сводят к поиску локального минимума, т. е. поиску такой точки X^*, что для любого X из некоторой окрестности X^* выполнено $f(X) \geq f(X^*)$.

Задачу нахождения максимума функционала можно свести к задаче нахождения минимума просто поменяв знак перед функционалом, поэтому нет смысла рассматривать ее отдельно

В данной работе будем подбирать такие задачи, в которых нужно именно найти хотя бы локальный минимум, а не доказывать его несуществование.

Методы решения поставленной задачи делятся на детерминированные и стохастические. Детерминированные методы не используют случайность в процессе решения задачи, а стохастические - используют. Таким образом, стохастические методы отличаются в основном тем, что имея одни и те же данные и параметры можно получить разные результаты.

Теоретическое описание

Одним из возможных решений описанной проблемы будет считать градиент не точно, а приближенно, но во-первых, мы не можем гарантировать, что это упростит вычисления. Во-вторых, такой метод нельзя назвать стохастическим, поэтому подобные методы не рассматриваются.

1. SGD

Stohastic Gradient Descent. Этот метод не позволяет нам избавиться от вычисления точного значения градиента, но я не мог не включить его сюда, так как он является стохастической версией градиентного спуска, упомянутого в постановке задачи.

Для начала разберемся, что же такое градиентный спуск. Алгоритм такой:

1. Случайно выбираем начальное приближение $X_0 \in \mathbb{X}$
2. Пока не выполнено условие останова:
$$X_i = X_{i-1} - \eta_i * \nabla_X f(X_{i-1})$$
где i - номер итерационного шага, η_i - положительный параметр для текущего шага. Мы будем рассматривать случай, когда η одинакова для всех i, но в общем случае это не так.

Условие останова в общем случае произвольное. Часто алгоритм останавливают либо когда превышено заданное заранее число итераций, либо когда величина обновления по модулю не превосходит некоторого (близкого к нулю) числа, либо если мы знаем ТНГ функции и достаточно близки к ней.

На каждом шаге мы считаем градиент и движемся в обратную от него сторону, то есть в сторону наибольшего убывания функции.

Так же мы предполагаем, что направление градиента функции пренебрежимо мало изменяется в окрестности каждого приближения. Размер окрестности существенно зависит от параметра η_i. Поэтому для гладких функций всегда можно подобрать последовательность η_i, такую, чтобы алгоритм сходился хотя бы к локальному минимуму. Стоит отметить, что этот алгоритм хорошо работает на гладких функциях даже при выборе одной и той же η для каждого шага.

SGD начинает отличаться от обычного GD, только когда вводятся такие понятия, как выборка и объект выборки. Поставим задачу так:

Имеется:

1. Выборка объектов $X = \{x_1, x_2, \dots, x_n, \dots\}$, $x_i \in \mathbb{R}^m$ - независимые одинаково распределённые случайные векторы ($i.i.d.$)
2. Множество параметров $\Theta \in \mathbb{R}^k$
3. Функция $f(\theta, x_i) \to \mathbb{R}$
4. Функция $F = \frac{1}{n}\sum_{i=1}^{n} f(\theta, x_i)$

Задача: Найти $\theta^* \in \Theta$, такое что $F(\theta^*, X) = min_{\theta \in \Theta} F(\theta, X)$ или хотя бы локальный минимум

Недостатки GD при такой постановке задачи:

- Выборка не может быть очень большой, так как нужно одновременно всю выборку держать в оперативной памяти
- Выборка обязана быть конечной
- Одна итерация алгоритма вычислительно дорогая

Идея SGD: брать из выборки один (или небольшое число) случайных объектов и делать шаг GD только на них.

В случае, когда берется небольшое число объектов, алгоритм также называют Mini-batch SGD, а малую подвыборку объектов называют батч.

Таким образом:

- Выборка может быть сколь угодно большой или даже бесконечной
- Мы можем генерировать выборку "на ходу"
- Одна итерация становится дешевой
- Увеличивается количество итераций, но если учитывать, что объекты выборки $i.i.d.$, то сходимость наступает за меньшее число вычислений, чем в случае GD

На практике сложно гарантировать, чтобы $X \sim i.i.d$, однако SGD всё равно дает хорошую сходимость.

В заключение отметим так же, что есть масса улучшений SGD, например, Nesterov accelerated gradient, Adam, но все они основаны на той же идее движения в сторону антиградиента и, следовательно, не избавляют нас от потребности вычислять градиент. Стохастичность всех этих методов заключается только в случайном выборе батчей. Поэтому из этого широкого семейства оптимизаторов мы будем рассматривать только SGD, как самый простой и самый наглядный.

2. Случайный поиск с возвратом.

Алгоритм очень прост и опирается только на случайность.

Алгоритм такой:

1. Выбираем случайный $X_0 \in \mathbb{X}$
2. Пока не выполнено условие останова:
   A. Выбираем случайный $X_i \in \mathbb{X}$
   B. Если $f(X_i) > f(X_{i-1})$, то $X_i := X_{i-1}$

Условие останова в общем случае произвольное. Это может быть максимальное число итераций или оптимальность очередного X_i, если, например, функция $f(X)$ дискретна и мы знаем, какой должен быть минимум.

Алгоритм не пригоден для большинства задач, однако он справляется хорошо, когда

- Пространство решений содержит высокую долю подходящих решений
- Нельзя определить признаки приближения к оптимальному X^* - в частности, когда функция возвращает только два значения - подходит / не подходит

3. Генетический алгоритм.

Алгоритм основан на идее моделирования эволюционного отбора. Здесь мы будем оперировать таким понятием, как популяция - множество $\{X_1, \ldots, X_n\}$ приближений. Мы будем стараться сделать так, чтобы на каждом следующем шаге популяция в среднем была как можно ближе к оптимальному X^*

Формально выглядит так:

1. Случайно генерируем начальную популяцию: $\{X_1, \ldots, X_n\}$
2. Естественный отбор:
   A. Считаем $f(X)$ для каждого объекта популяции
   B. Наверняка оставляем "элиту" - $elite \in [0, 100)$ процентов объектов с наименьшим $f(X)$, остальные "выживают" с маленькой вероятностью $p_{alive} \in [0, 1)$
3. Скрещивание: пока популяция снова не станет размера n
   A. Берем два случайных объекта-родителя
   B. В общем случае, наша задача - получить объект-дитя, который получит что-то от обоих родителей, и добавить его в популяцию. В моей реализации я буду брать взвешенную сумму со случайными коэффициентами для чисел. А для векторов и двух-и-более-мерных конструкций я буду для каждой позиции выбирать случайно элемент у одного из родителей с вероятностью 0.5.
4. Мутации:
   A. Каждый объект с маленькой вероятностью $p_{mut} \in [0, 1)$ претерпевает какие-то случайные изменения. Договоримся прибавлять к случайным компонентам случайное число, по модулю не превосходящее максимальный модуль компоненты, умноженный на 5.
5. Если после этих преобразований условие останова не выполнено, то возвращаемся к пункту 2.
6. В конце выбираем из популяции объект, для которого $f(X)$ - наименьшее

Отметим, что в итоге на последних шагах нам важно не чтобы вся популяция была ближе к оптимуму, а чтобы в популяции был объект, достаточно близкий к оптимуму.

Поэтому возможные критерии останова такие:

- Превышено максимальное количество популяций
- $f(X_{best}) - min_X f(X) < \epsilon$, где X_{best} - "лучший" объект популяции, а ϵ - число > 0
- $Patience \in \mathbb{N}$ популяций подряд $f(X_{best})$ не меняется

Отметим, что как недостатком, так и преимуществом этого алгоритма является обилие структурных параметров для настройки:

- n - размер популяции
- $elite$
- p_{alive}
- p_{mut}
- Максимальное количество популяций
- ϵ, если мы знаем $min_X f(X)$ заранее
- $Patience$

Очевидным преимуществом этого алгоритма является возможность применять его к широкому классу задач, потому что он почти ничего не требует от $\mathbb{X}$ или $f(X)$ и это могут быть абсолютно произвольные объекты.

Главные требования - чтобы функция $f(X)$ была определена для любого $X \in \mathbb{X}$ и чтобы скрещиванием и мутацией любого объекта из $\mathbb{X}$ получался снова объект из $\mathbb{X}$. Хотя эти требования тоже необязательны, по сути, мы можем просто выкидывать неугодные X.

Очевидным недостатком этого алгоритма является то, что под каждую конкретную задачу нужно выбирать свои способы скрещивания и мутации из огромного множества интуитивно подходящих функций. В наших экспериментах ограничимся функциями, указанными выше.

4. Алгоритм роя частиц.

Этот алгоритм опять же из биологии. Он основан на наблюдениях за поведением стаи птиц. Одной птице очень сложно найти много пищи, поэтому они обмениваются информацией друг с другом. Далее каждая птица учитывает свой собственный опыт (где она видела пищу) и опыт других птиц. В задаче оптимизации одна "птица" задается тремя параметрами: текущая позиция, текущая скорость, лучшая позиция. Позиции - это объекты $X \in \mathbb{X}$. Чем $f(X)$ меньше, тем лучше позиция, тем больше пищи нашла птица в этой точке. Так же, мы храним в памяти лучшую позицию из всех, которые посетила хотя бы одна птица, - точка, в которой стая нашла больше всего пищи, и ориентируемся на эту точку во время движения.

Таким образом, сформируем основные положения этого алгоритма:

- птицы живут в мире, где время дискретно
- птицы оценивают свою позицию с помощью $f(X)$
- каждая птица знает позицию в пространстве, в которой она нашла наибольшее количество пищи (своя наилучшая позиция)
- каждая птица знает позицию в пространстве, в которой найдено наибольшее количество пищи среди всех позиций, в которых были все птицы (общая наилучшая позиция)
- птицы имеют тенденцию стремиться и к лучшим позициям, в которых были сами, и к общей наилучшей позиции
- птицы случайным образом меняют свою скорость, так что описанная тенденция определяет лишь усредненное движение птиц
- птицы обладают инерцией, поэтому их скорость в каждый момент времени зависит от скорости в предыдущий момент
- птицы не могут покинуть заданную область поиска

Формально опишем базовую реализацию этого алгоритма. Разумеется, он имеет множество модификаций, но я буду использовать для экспериментов именно эту версию:

1. Случайно задаем начальные позиции птиц $\{X_1, \dots, X_n\} \in \mathbb{X}$ и их начальные скорости $V_1, \dots, V_n \in V$, где V - множество допустимых скоростей (в нашей реализации ограничим по модулю сверху и снизу компоненты скоростей, считая, что птица не может лететь слишком быстро или слишком медленно). Из начальных позиций выбираем лучшую X_{best}.
2. Высчитываем для каждой птицы скорость по формуле:
$$V_{new} = w * V_{old} + \alpha_1 * (X_i^{best} - X_i) * rnd_1 + \alpha_2 * (X_{best} - X_i) * rnd_2$$
где V_{old} - текущая скорость птицы, w - инерция, α_1, α_2 - коэффициенты, насколько птица опирается на свой опыт и опыт стаи, X_i^{best} - лучшая позиция данной птицы, $rnd1, rnd2$ - случайные числа в диапазоне $[0, 1)$
3. Заменяем каждую координату скоростей на ближайшую, удовлетворяющую условиям из 1.
4. Обновляем позицию каждой птицы: $X_{new} = X_{old} + V_{new}$, если так получилось, что птица "вылетела" из множества $\mathbb{X}$, то возвращаемся в старую позицию. (В нашей реализации разрешим птицам "вылетать", так как (спойлер) все рассматриваемые в экспериментах функции определены на $\mathbb{R}^n$)
5. Пересчитываем лучшую позицию для каждой птицы по отдельности и для всей стаи в принципе
6. Если критерий останова не выполнен, то возвращаемся к шагу 2.
7. В конце ответом будет лучшая позиция среди всех птиц X_{best}.

Возможные критерии останова такие:

- Превышено максимальное количество "перелётов"
- $f(X_{best}) - min_X f(X) < \epsilon$, где ϵ - число > 0
- $Patience \in \mathbb{N}$ "перелётов" подряд $f(X_{best})$ не меняется

Структурные параметры у этого алгоритма такие. Их, опять же, достаточно много:

- Количество птиц
- V_{min}, V_{max} - минимальный и максимальный модуль допустимой скорости
- w, α_1, α_2 - учет инерции, собственного опыта и опыта стаи в пересчете скорости
- Максимальное количество "перелётов"
- ϵ, если мы знаем $min_X f(X)$ заранее
- $Patience$

Алгоритм рассчитан на непрерывность множества $\mathbb{X}$ и на хотя бы кусочную непрерывность и определённость функции $f(X)$ на множестве $\mathbb{X}$.

Тем не менее, если что-то идет не так, птица может просто вернуться в предыдущую точку. Поэтому условия выше не то чтобы обязательные, но тут уже вопрос в вероятности такого возвращения в конкретной задаче. Если вероятность попадания в "плохую" точку большая, то алгоритм, очевидно, будет неэффективен.

5. Алгоритм роя пчёл.

Алгоритм основан на наблюдениях за поведением пчёл в естественной среде обитания. В начале из улея вылетают пчёлы-разведчики, которые выявляют места на поляне, где много нектара. Затем в места, где больше всего нектара, вылетают пчёлы-работники и собирают нектар. В нашем случае количество нектара в точке будет оцениваться $f(X)$. Чем функция меньше, тем больше нектара.

Формализованно, алгоритм работает так:

1. Высылаем n_{scout} пчёл-разведчиков в случайные точки.
2. Определяем n_{best} лучших точек (с наименьшим $f(X)$) и $n_{perspective}$ точек, идущих сразу после них.
3. Высылаем в окрестность каждой из n_{best} точек c_{best} пчел-работников, позиция каждой пчёлы-работника считается как: $X = X_{best,i} + rnd * rad$, где rnd - случайный вектор со значениями от -1 до 1, а rad - "радиус" рассматриваемой кубической окружности точки $X_{best,i}$. (т.е. половина стороны куба с центром в точке $X_{best,i}$).
4. Аналогично шагу 3, высылаем $c_{perspective}$ пчёл-работников в окрестности каждой из $n_{perspective}$ точек.
5. Вычисляем лучшее положение среди пчёл-работников, обновляем лучший результат, если это необходимо.
6. Если не выполнено условие останова, то возвращаемся к шагу 1.
7. В конце ответом будет лучший результат на всё итерации.

Критерии останова могут быть следующие:

- Превышено максимальное количество сборов нектара
- $f(X_{best}) - min_X f(X) < \epsilon$, где ϵ - число > 0
- $Patience \in \mathbb{N}$ сборов нектара подряд $f(X_{best})$ не меняется

Настраиваемые параметры:

- n_{scout}
- $n_{best}, n_{perspective}$
- $c_{best}, c_{perspective}$
- rad
- Максимальное количество сборов нектара
- ϵ, если мы знаем $min_X f(X)$ заранее
- $Patience$

Много настраиваемых параметров, нужно подбирать под задачу.

Алгоритм рассчитан на непрерывность множества $\mathbb{X}$ и на хотя бы кусочную непрерывность и определённость функции $f(X)$ на множестве $\mathbb{X}$.

Хотя, как и в предыдущем алгоритме, можно считать, что пчела, попавшая в неудачную точку, "недолетела" и не учитывать её. Другой вопрос, что если пчёлы часто будут "недолетать", то эффективность алгоритма снизится.

Практическая реализация.

Практическая реализация описанных выше методов и экспериментов над ними доступна по ссылке: https://github.com/AZelentsov343/stohastic_optimization_methods (https://github.com/AZelentsov343/stohastic_optimization_methods)

(заранее извиняюсь, если вы сейчас читаете это с бумаги). В данной статье - только описание экспериментов и их результатов. По этой же ссылке можно посмотреть графики сходимости.

Практическое применение. Эксперименты.

Вот совершенно разные задачи, которые я смог придумать (вспомнить, взять откуда-то). Они разные по постановке, ограничениям, сложности.

В каждой задаче нас будет интересовать лучший результат с подбором параметров, лучший набор параметров, время выполнения, количество итераций за которое этот результат был получен.

Сразу оговоримся, что цель - сравнить алгоритмы, а не найти лучшие параметры для каждого, поэтому настройка параметров будет производиться "на глаз".

Также, если в задаче есть возможность обозначить, что значит удовлетворительный ответ, то нас будет итересовать так же скорейшее время получения удовлетворительного ответа.

Время на поиск лучшего результата договоримся ограничивать минутой. То есть будем брать такое число итераций, чтобы алгоритм работал не дольше минуты.

Так же, реализуя эти алгоритмы, я не ставил перед собой цели написать самую эффективную реализацию из возможных, поэтому в общем случае они могут работать быстрее, если, например, отказаться от питоновских циклов и реализовать все операции с помощью numpy.

Все графики сходимости можно посмотреть по ссылке с реализацией.

Задача 1. Поиск простого числа.

Задача очень просто поставлена - найти любое простое пятизначное число. Оптимизируемый функционал:

$$f(x) = \begin{cases} 0, & rnd(x) \text{ - простое} \\ 1, & rnd(x) \text{ - не простое} \end{cases}$$

метод	наименьшее значение	время выполнения	количество итераций
GD	1	-	-
Случайный поиск	0	160 ms	17
Генетический алгоритм	0	131 ms	1
Алгоритм роя частиц	0	129 ms	1
Алгоритм роя пчёл	0	152 ms	1

Стоит отметить, что GD не справился с заданием, потому что функция кусочно-постоянная и, как следствие, в точке производная либо 0, либо не существует, поэтому итеративный процесс просто не двигался, наугад выбрав неподходящее число.

Это как раз тот случай, когда много подходящих ответов, но нельзя определить признаки приближения к оптимальному ответу, поэтому случайный поиск в этом случае работает хорошо.

Остальные же алгоритмы работали по принципу: больше точек просмотрено за одну итерацию - быстрее сойдётся итеративный процесс. Но из-за большого количества оптимальных ответов все сошлись за одну итерацию. Поэтому о выборе параметров речи не идёт.

Время у всех справившихся алгоритмов примерно одинаковое, поэтому можно предположить, что они рассмотрели примерно одинаковое количество точек.

Задача 2. Сумма квадратов.

Тоже простая задача - найти минимум функции вида $x_1^2 + x_2^2 + \ldots + x_n^2$. Будут рассмотрены варианты для n = 3, n = 10.

Немного подскажем нашим алгоритмам и скажем, что оптимум гарантированно находится где-то в кубе $[-500, 500]^n$. Так же скажем, что если значение функции меньше 0.001, то решение удовлетворительное.

$n = 3$

метод	наименьшее значение	время выполнения	количество итераций	
GD	0.0	4.84 s	471732	114 ms
Случайный поиск	12.136302174655393	1 min	2500000	
Генетический алгоритм	7.91319829892832e-05	1 min	691	
Алгоритм роя частиц	3.232286776098316e-09	1 min	36197	
Алгоритм роя пчёл	0.4904120596542764	1 min	120	

метод	среднее время получения удовлетворительного ответа	среднее количество итераций для удовл. ответа
GD	114 ms	478
Случайный поиск	-	-
Генетический алгоритм	10.6 s	129
Алгоритм роя частиц	211 ms	49
Алгоритм роя пчёл	-	-

GD/SGD

Гладкая функция, у которой элементарными образом можно посчитать градиент. Конечно, в этой ситуации GD показал лучший результат с большим отрывом.

Но нас больше интересует не GD, а то, как с задачей, характерной для GD, справляются стохастические методы.

Случайный поиск

Случайный поиск ожидаемо плохо себя показал, просто перебрав 2500000 точек и не попав наугад в удовлетворяющую.

Генетический алгоритм

Генетический алгоритм во время одного из прогонов показал хороший результат, но как выяснилось позже, это была случайность. В основном же, он показывал числа около 10^{-3} и иногда 10^{-2}, что тоже сравнительно неплохо.

Размер популяции в этой задаче не играет большой роли, при размере от 100 до 2000 результаты особо не менялись. Слишком маленькая популяция охватывает слишком мало точек, поэтому результат хуже. Слишком большая популяция долго пересчитывается и из-за этого процесс сходимости замедляется.

Если считать элитными слишком много (больше 60%) объектов, то процесс начинает ухудшаться, возможно даже расходиться. Оптимальное значение было 0.2-0.4.

Увеличивать вероятность выживания в этой задаче так же не стоит потому что в среднем это получается эквивалентным увеличению процента элитных объектов, только в элиту могут попасть уже любые объекты, не обязательно самые лучшие. Лучше не больше 0.1.

А вот мутацию увеличить стоило, лучшее значение - 0.3. Как я понял, это связано с тем, что с вероятностью 50% мутация уменьшает по модулю компоненты решения, что в данной ситуации уменьшает значение функции. Однако, если сделать мутацию слишком большой, то алгоритм начинает вести себя неадекватно и расходиться.

Алгоритм роя частиц.

Лучше всех показал себя из стохастических методов в этой задаче. Вероятно, это связано с тем, что птицы летят, ориентируясь друг на друга.

Количество птиц важно только при начальной инициализации, потому что далее они все равно начинают слетаться в нужную сторону при правильном подборе остальных параметров. Малое количество птиц увеличит количество итераций, нужных для сходимости. Большое количество птиц замедлит одну итерацию, таким образом превратив алгоритм в некое подобие случайного поиска, а нам это не нужно.

Чем меньше минимальная скорость, тем ближе результат будет к минимуму в данной задаче, потому что минимум один. В других случаях из-за малой скорости птица может застрять в точке локального минимума.

Повышение максимальной скорости может повысить сходимость, но так же птицы на высокой скорости могут улететь в другую сторону. Очень сильно зависит от размера региона поиска, я использовал 20.

Инерция, a_1, a_2 - самые важные параметры, их и нужно настраивать в первую очередь.

При слишком большой инерции птицы слишком сильно "помнят" скорость, которая была задана в начале случайным образом. Оптимальное значение - 0.2.

a_1, a_2 в этой задаче лучше всего сделать примерно одинаковыми. Чуть лучше сработал такой набор - $a_1 = 0.5, a_2 = 0.3$

Алгоритм роя частиц.

Алгоритм роя пчёл сработал хуже всех, не считая случайного поиска. Связано это с тем, что этот алгоритм не ищет закономерностей и каждый следующий шаг не зависит от предыдущего.

Оптимальной стратегией оказалось посылать много разведчиков (100000), потом отбирать из них 10 лучших и 5 перспективных и посылать в их окрестности 100 и 50 работников.

Размер окрестности было решено взять равным 5, вообще, большой размер окрестности хорош, когда мы далеко от оптимума и плох, когда близко. Поэтому 5 получилось просто подбором.

$n = 10$

метод	наименьшее значение	время выполнения	количество итераций
GD	0.0	4.45 s	465858
Случайный поиск	26843.801272205634	1 min	2500000
Генетический алгоритм	8.55750725393341	1 min	711
Алгоритм роя частиц	3.24234234248267	1 min	7257
Алгоритм роя пчёл	20595.61136221198	1 min	120

метод	среднее время получения удовлетворительного ответа	среднее количество итераций для удовл. ответа
GD	130 ms	515
Случайный поиск	-	-
Генетический алгоритм	-	-
Алгоритм роя частиц	-	-
Алгоритм роя пчёл	-	-

Как мы видим, при переходе в 10мерное пространство у алгоритмов возникают проблемы. GD масштабируем и его работа на 3мерном пространстве не отличается от 10мерного случая.

Случайный поиск и алгоритм роя пчёл начинают работать хуже потому что теперь им нужно "угадать" 10 раз. Даже если они выбрали близким к нулю одно из значений, другое может оказаться большим, и такое решение выкинут.

Генетический алгоритм, наоборот, догнал рой частиц. Связано это как раз с тем, что генетический алгоритм тоже мастабируем и может делать отдельные компоненты очень малыми из-за скрещивания. То есть генетический алгоритм бы хорошо сошелся, если бы лимит по времени был не минута, а, скажем, 30.

Рой частиц внезапно "слетелся" к вектору из 1 и -1. Лучший результат был получен, когда в векторе большинство значений было ближе к нулю, чем к единице. Пытался повышать минимальную скорость - становится сильно хуже. Скорее всего это связано с тем, что $|b^2| < |b|$, если $|b| < 1$. Видимо, опять же, нужно больше времени, чтобы случайная птица вылетела из области точки $(1, \dots, 1)$ в более выгодную.

Задача 3. Линейная регрессия.

Дана функция вида $f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{n} w_i * x_i$. Нужно по значениям и аргументам функции восстановить вектор w. Будем рассматривать только вариант $n = 3$, потому что по опыту прошлой задачи при увеличении размерности становится хуже.

Еще одна типичная задача, которую обычно решают градиентным спуском. Похожие задачи оптимизации возникают в машинном обучении, когда речь идёт о линейных моделях.

В этой задаче загадаем веса, равные 1, 2, 3 соответсвенно и будем выдавать объекты мини батчами по 10 штук, пытаясь отгадать веса с подсказкой, что веса находятся где то на отрезке $[-100, 100]$. Лосс-функция - MSE.

$$f(w) = \frac{1}{10} \sum_{i=1}^{10} (\sum_{i=1}^{3} w_i * x_i - \sum_{i=1}^{3} w_i^{true} * x_i)$$

где x_i - объект батча

Удовлетворительный ответ - $MSE < 0.001$

метод	наименьшее значение	время выполнения	количество итераций
SGD	3.2869204384208823e-29	1.43 s	21223
Случайный поиск	78.07806595780144	1 min	758999
Генетический алгоритм	0.6084039163620631	1 min	1033
Алгоритм роя частиц	1.9249170331813867e-13	1 min	1990
Алгоритм роя пчёл	0.28858572152216594	1 min	38

метод	среднее время получения удовлетворительного ответа	среднее количество итераций для удовл. ответа
SGD	151 ms	332
Случайный поиск	-	-
Генетический алгоритм	-	-
Алгоритм роя частиц	726 ms	19
Алгоритм роя пчёл	-	-

Оптимальные параметры для для данных алгоритмов не отличаются от найденных в предыдущем задании, потому что задания похожи (и там и там квадратичный гладкий функционал).

В этом задании уже была использована стохастическая версия градиентного спуска, единственное выявленное отличие - "ребристость" графика. Действительно, когда речь идёт о батчах, значения лосса могут скакать.

Причины неудачи случайного поиска объяснять, видимо, не имеет смысла - он не очень хорош на гладких функциях.

Рой частиц ожидаемо немного уступает градиентному спуску.

Неожиданностью в этой задаче является то, что алгоритм роя пчёл решает её лучше, чем генетический алгоритм. Связано это скорее всего с тем, что в этой задаче область поиска меньше по сравнению с предыдущей и рой пчёл банально захватывает больше отрезков.

Эксперимент так же проведен для других наборов весов, но результаты не отличаются существенно.

Задача 4. Странные функции

Минимизировать функцию $x_1^2 * sin(x_1) + x_2^2 * sin(x_2) + x_3^2 * sin(x_3)$.

Будем инициализировать алгоритмы значениями из $[-200, 200]^3$, какие-то алгоритмы не смогут выйти из этого куба, а какие-то смогут, то это вовсе не значит, что вторые найдут значение меньше. У этой функции нет глобального минимума, зато куча локальных, поэтому критерий удовлетворительности решения задать нельзя. Просто дадим каждому методу поработать 5 раз по минуте и посмотрим, что этот метод найдёт.

метод	наименьшее значение	среднее	максимальное
GD	-3982.4934421718185	1156.8961500232176	8191.420870074659
Случайный поиск	-112719.784829333	-94463.06910247222	-28318.066374647693
Генетический алгоритм	-1.2e308	-7.9e307	-3.5e307
Алгоритм роя частиц	-89019.92803420116	-67699.4731380947	-16028.144048996117
Алгоритм роя пчёл	-115201.87974716499	-113018.12418362121	-109447.3974420336

GD на этой задаче показал себя хуже всех, потому что у функции нет глобальных экстремумов и в целом градиент ведёт себя очень сложно. GD колебался между разными точками, в итоге в конце итерационного процесса оказываясь непонятно где.

Случайный поиск работает в целом неплохо, на удивление, однако он нестабилен и может выдавать разные результаты. Также этот алгоритм не может выбраться из куба, поэтому диапазон значений ограничен.

А вот генетический алгоритм способен выбраться из куба благодаря мутациям, поэтому минимум, который находит он, близок к -inf. В целом, генетический алгоритм показывает лучший с большим отрывом результат в этой задаче. Это связано с тем, как происходит скрещивание. Если всё удачно совпадает, то вектора с компонентами, дающими маленькие результаты на своей позиции, пересекаются и получается еще более маленький результат.

Алгоритм роя частиц, хоть и способен выйти из куба, находит не те значения и не выходит. Это связано с тем, что из предложенных методов он больше всех похож на градиентный спуск, поэтому птицы летают хаотично и минимизации не получается. Этот метод работает лучше GD, потому что он сохраняет лучшую полученную точку.

Алгоритм роя пчёл работает стабильно хорошо, и, учитывая, что он не может выбраться из куба, скорее всего находит числа, близкие к минимуму в кубе.

Удивительно, но к этой задаче так же подошли параметры из предыдущих. При изменении набора параметров качество ухудшалось.

Задача 5. Жизнь в обратную сторону

По заданному состоянию игры "Жизнь" Конвея (https://ru.wikipedia.org/wiki/Игра_«Жизнь» (https://ru.wikipedia.org/wiki/Игра_«Жизнь»)) найти предыдущее или максимально похожее на предыдущее. Поле 10x10, соседи крайних клеток вне поля считаются мертвыми.

Будем пытаться найти предыдущее состояние как матрицу 10x10 состоящую из 0 и 1. Минимизируемый функционал
$f(X) = MAE(makestep(X), Y_{true})$

Функция кусочно-постоянная, как меняется на кусках в 100мерном пространстве - сказать сходу сложно. Продифференцировать это вроде возможно, но дифференцировать свёртку, логическую операцию и модуль - крайне сложно. Поэтому не будем пытаться решить это с помощью GD. Скорее всего, даже если попытаться, то не получится хорошо, потому в задаче не один правильный ответ и уменьшение функционала не означает приближения к правильному ответу.

Будем считать удовлетворительными решения $f(X) = 0$

метод	наименьшее значение	время выполнения	количество итераций
Случайный поиск	0.28	1 min	504841
Генетический алгоритм	0.11	1 min	1247
Алгоритм роя частиц	0.23	1 min	1354
Алгоритм роя пчёл	0.28	1 min	14

Не один из алгоритмов не добился удовлетворительного результата, однако генетический алгоритм и рой частиц показали лучшие результаты.

Требовалась небольшая настройка параметров. В генетическом алгоритме для достижения лучшего результата была уменьшена вероятность мутации и выживания до 0.05, но увеличен процент "элиты": оптимальное значение - 60%.

В алгоритме роя частиц нужно было только поменять максимальную скорость - она стала равной единице. То есть значение каждого элемента "птицы" может поменяться за одну итерацию. Это дало существенный прирост по сравнению с прогонами, когда максимальная скорость была маленькой. Остальные параметры остались примерно такими же, как в прошлых задачах.

Для алгоритма роя пчёл перенастройка параметров не требовалась.

Выводы.

В этой секции попробуем выделить плюсы и минусы каждого из рассматриваемых алгоритмов исходя из поставленных экспериментов и простой логики.

SGD.

Плюсы:

- Если объекты выборки $i.i.d.$, то содержит в себе все плюсы GD, а именно - быструю сходимость на гладких функциях имеющих глобальный минимум. Именно поэтому в базовых моделях машинного обучения обычно подбирается подходящая функция и используется GD(SGD) или их аналоги.
- Масштабируем, как и GD. С увеличением размерности качество не ухудшается, так как с каждой компонентой работает отдельно.
- Избавляет от вычислительной сложности, присущей GD, потому что теперь на каждой итерации работа идёт лишь с малой частью выборки фиксированного размера. Это делает выборку теоретически бесконечно большой.
- Если объекты выборки $i.i.d.$ и функция подходящая, то приходит к разумному решению быстрее чем GD, потому что веса обновляются чаще и, так как объекты $i.i.d.$, то в среднем веса обновляются в нужную сторону.

Минусы:

- Если объекты выборки не $i.i.d.$ может не работать, так как теоретически каждый батч может "тянуть" в разные стороны.
- Появляется "ребристость" на графике сходимости, может скакать.
- Зависимость от вида функции и её производной. Если функция не гладкая или не имеет глобального минимума, то метод сильно зависит от начального приближения и может давать плохой результат. Если функция не дифференцируема, то вовсе неприменим. Если функция имеет сложновычисляемую производную, то работает медленно.
- На выходе может дать не лучшее значение из найденных, так как забывает предыдущие значения.

Случайный поиск

Плюсы:

- Очень прост в использовании и написании, требует от функции только определённость на множестве поиска
- Не имеет настраиваемых параметров
- На каждом шаге гарантированно не становится хуже

Минусы:

- Очень прост, неприменим для большинства задач т.к. не знает о функции ничего, кроме ее значений в конкретных точках и на каждой итерации работает только с одной точкой, не пытаясь как то их связать.
- Требует задания ограниченной области поиска, далее не может найти ничего вне этой области.
- Немасштабируем. С увеличением размерности пространства качество резко снижается.

Генетический алгоритм

Плюсы:

- Нет заметной зависимости от начальной инициализации. Выходит из участка инициализации засчет мутаций и ищет подходящую точку на всем $\mathbb{R}^n$. Если это не нужно, то можно запретить это делать, просто изменив процесс мутации или не добавляя в популяцию неподходящие объекты.
- Самый масштабируемый из рассмотренных, не считая SGD. Связано это с тем, как происходит скрещивание.
- Не требует от функции по сути ничего кроме определённости на области поиска, а желательно на $\mathbb{R}^n$.
- Отлично справляется с функциями, имеющими много локальных, но не имеющими глобального минимума.

Минусы:

- Всё равно до конца масштабируемым назвать нельзя. С ростом размерности качество ухудшается.
- Достаточно много параметров, требуется их настройка, в разных задачах работают разные "стратегии". При неправильном подборе параметров и вовсе расходится.
- Следующий шаг может быть хуже предыдущего, если лучший объект мутировал не в ту сторону.

Рой частиц

Плюсы:

- Лучше всех, кроме GD, работает с гладкими функциями, имеющими глобальный минимум. Поэтому этот алгоритм с натяжкой можно назвать безградиентной версией градиентного спуска. Но в отличие от GD неплохо себя показывает на негладких функциях.
- Способен выбираться из области поиска. Если не нужно, это можно легко запретить.
- Не требует от функции ничего кроме определённости.(но втайне надеется на наличие у нее глобального минимума и непрерывность)
- Следующий шаг гарантированно не хуже предыдущего.

Минусы:

- Как и GD, плохо работает, когда у функции много локальных и нет глобального минимума. В таком случае сильно зависит от инициализации.
- Требуется настройка параметров. Есть разные стратегии полёта птиц, хотя во всех представленных задачах работала одна и та же.
- Скорее немасштабируем. Качество ухудшается с увеличением размерности.
- Следующий шаг зависит от предыдущих, алгоритм может застревать.

Рой пчёл

Плюсы:

- Не требует от функции совершенно ничего, кроме определённости. Хоть в основе и лежит гипотеза о том, что оптимальные точки часто расположены в окрестности друг друга, непрерывности функции не требует.
- Следующий шаг гарантированно не хуже предыдущего.
- Каждый шаг не зависит от других, таким образом, алгоритм не зависит от инициализации и не может застревать.

Минусы:

- Требует задания области поиска, из которой впоследствии не может выбраться (вообще то может, но не более, чем на rad, поэтому область поиска всё равно ограничена).
- Немасштабируем. При увеличении размерности качество резко ухудшается и алгоритм становится похож на случайный поиск.
- Работает только с текущим набором точек, не пытаясь связать их с теми, что видел ранее

Глобальный вывод.

В данной работе было рассмотрено 5 методов стохастической оптимизации. Были показаны как сильные стороны, так и слабости GD/SGD. Вывод: в определённом классе задач алгоритм идеален, в остальных почти неприменим. Какие могут быть замены? Для задач, где нужно найти ответ, принадлежащий дискретному множеству лучше всего себя проявил генетический алгоритм. С задачей поиска на множестве хорошо (чем меньше множество, тем лучше) справляются рой пчёл и случайный поиск. На гладких функциях лучше всех, не считая GD, работает рой частиц. Как я говорил выше этот метод с натяжкой можно назвать безградиентной версией градиентного спуска. Так же этот алгоритм неплохо работает в дискретных случаях.

Экспериментальный вывод: нельзя подобрать стохастический метод оптимизации, который хорошо бы справлялся с любым типом задач. Под каждый тип задач желательно подбирать алгоритм отдельно. При этом качества алгоритма нужно оценивать, усредняя по n прогонам, так как стохастическому методу в конкретном прогоне может "повезти" или наоборот.

Спасибо за внимание!