Методы первого порядка

Малюгин Руслан, Б05-921

19 декабря 2021 г.

Аннотация

В данной работе дан обзор на актуальные методы первого порядка, сравнение методов первого порядка с методами нулевого и второго порядка. Разобраны различные методы порядка, приведены их реализации. Приводится анализ использования методов в различных задачах и их применение на практике.

1 Сравнение методов различного порядка

В данной работе будут исследованы численные методы, которые в общем случае решают задачу оптимизации. К методам первого порядка относятся алгоритмы, в которых в процессе поиска кроме информации о самой функции используется информация о производных первого порядка. Соответственно методами нулевого порядка, называют те методы, где не используется производная функции, а вычисляется только значение функции в различных точках. Методами же второго порядка, называются методы, которые в своей работе используют гессиан функции.

Примеры методов:

- 0-порядок: метод Нелдера-Мида, симплекс-метод
- 1-порядок: различные градиентные спуски и их модификации
- 2-порядок: метод Ньютона

Важно заметить, что методы большего порядка имеют большую скорость сходимости, но в свою увеличивают вычислительные затраты при подсчете производных более выского порядка(гессиан многомерной функции бывает сложно вычислим) и использование памяти при увеличении размерности задачи. Скорость сходимости методов первого порядка - линейная, второго - квадратичная. В случаях методов нулевого порядка сложно оценить скорость сходимости. Так же важно понимать, что использование методов более высокого порядка накладывает большее количество условий. В то время, как методы нулевого порядка, применимы к шумным, негладким функциям, методы первого порядка применимы лишь к гладким функциям, а методы второго порядка накладывают еще более жесткие условия (см. Теорема Канторовича).

В следствии этого методы первого порядка имеют наиболее широкое применение, так как являются серединой между эффективностью и условиями применения. Методы первого порядка применяются, например при обучении моделей машинного обучения, или при численном решении задач.

Далее приведены графики сходимости различных методов.

Стоит заметить, что и по реальному времени результаты схожие.

6-размерная задача:

(0.03789925575256348, 0.019950389862060547, 0.016925334930419922)

3-размерная задача:

(0.008976459503173828, 0.008976936340332031, 0.0069844722747802734)

(0, 1, 2 порядки соответственно)

Но опять же, стоит повторить, что данная задача слишком хорошо для методов второго порядка, нежели, чем, например для нулевого. Но даже несмотря на это метод второго порядка не слишком опережает метод первого.

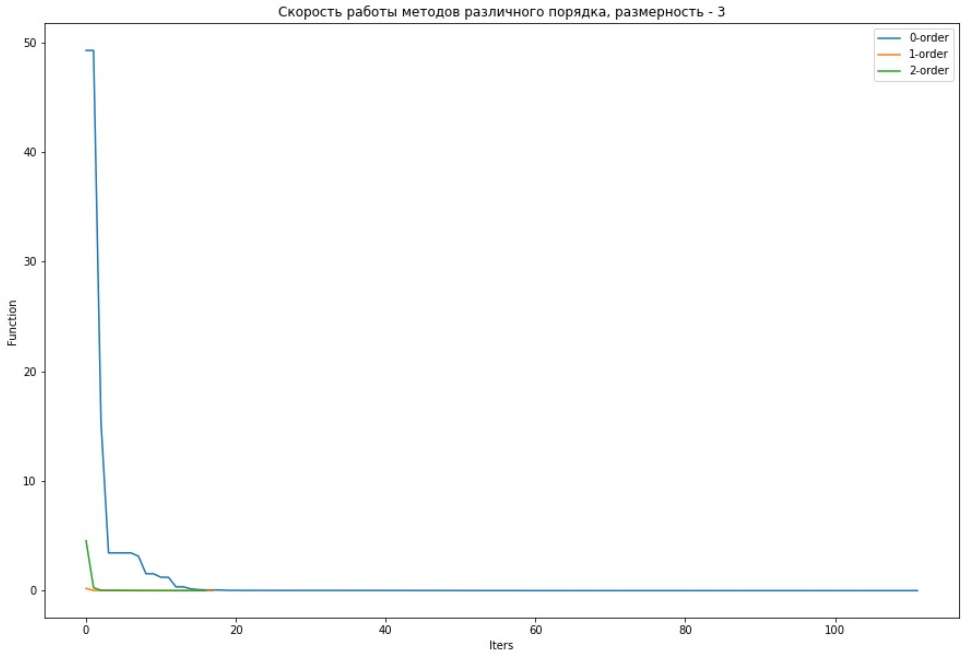

Рис. 1: График 1.

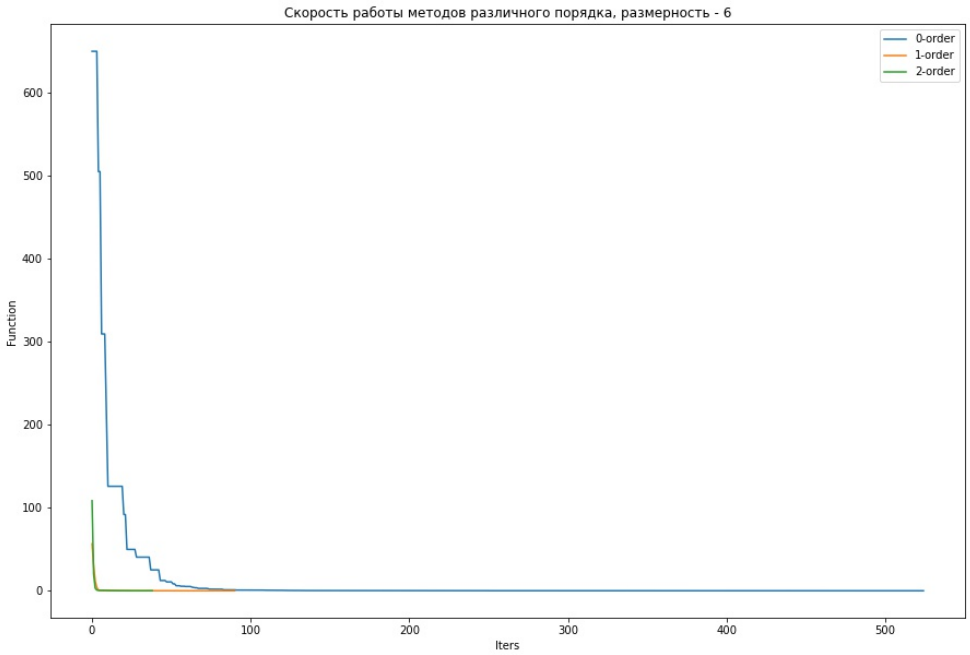

Рис. 2: График 2.

2 Методы первого порядка

Стоит сказать, что все методы первого порядка, по-сути являются улучшением или модификациями градиентного спуска. Поэтому будем постепенно рассматривать их, начиная с самого простого варианта. Обозначим цель и моменты, которые будут использоваться дальше. О сходимости методов поговорим далее.

Целевая функция - $F(\vec{x}) : X \to R$.

Задача оптимизации - $F(\vec{x}) \to \min_{\vec{x} \in X}$.

Так же в обзоре методов не будет указан выбор шага, об этом в следующей главе.

2.1 Градиентный спуск

Самый простой метод первого порядка. Заключается в движении в сторону антиградиента.

Алгоритм.

1. Задаются начальная точка x_0, выбирается точность ε, максимальное количество шагов n_{max}, критерий остановки $IsStop(...)$, который возвращает $True$, если нужно остановится или $False$ иначе.
2. Шаг метода : $x_{i+1} = x_i - \lambda \nabla F(x_i)$.
3. Если $IsStop(...) = False$ переходим к шагу 2.

Приведем код алгоритма на Python.

```
def GradientDescent(f, gradf, x0, epsilon, max_iter, rho=0.7, beta1=0.3):
    x = x0.copy()
    iteration = 0
    while True:
        # Step choose
        beta2 = 1. -beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)

        gradient = gradf(x)
        x = x - alpha * gradient
        iteration += 1

        # Stop criteria
        if np.linalg.norm(gradf(x)) < epsilon:
            break
        if iteration >= max_iter:
            break
    return x
```

2.2 Улучшение с методом моментов

Следующие два метода имеют под собой аналогии из физики. Градиентный спуск с импульсом запоминает изменение Δw на каждой итерации и определяет следующее изменение в виде линейной комбинации градиента и предыдущего изменения.

Шаг: $w_{i+1} = \alpha w_i - \lambda \nabla F(x_i)$
$x_{i+1} = x_i + w_{i+1}$

Эту идею можно дополнить следующим предположением :"Объекты имеют тенденцию сохранять направление движения некоторое время в будущем". Таким образом мы будем запоминать насколько мы сдвинулись в прошлый раз и добавлять эту величину, как бы "слегка забегая вперед".

Шаг: $w_{i+1} = \alpha w_i - \lambda \nabla F(x_i)$
$x_{i+1} = x_i - a w_i + (1 + \alpha) w_{i+1}$

В данном случае у метода два гиперпараметра - α и размер шага.

Приведем код на Python для этих методов

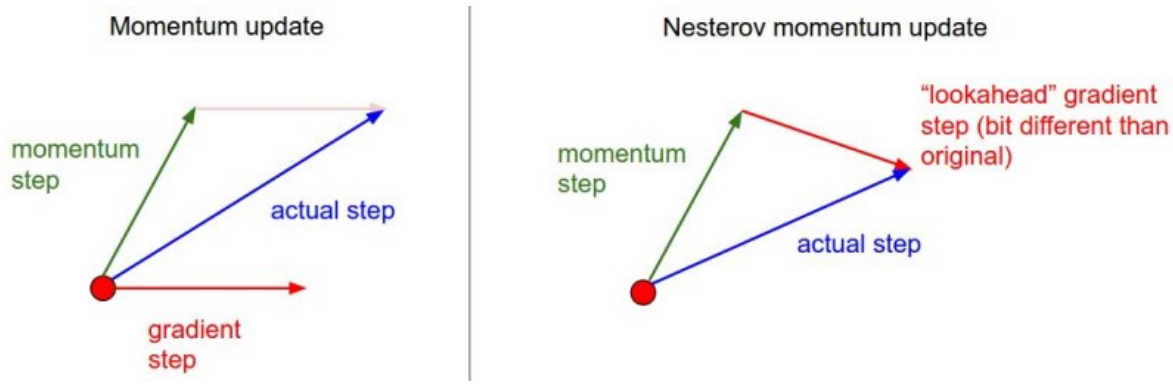

Рис. 3: Визуализация шага с методом моментов.

```
def NesterovAcceleratedGD(f, gradf, x0, momentum =0.9, max_iter=1000, tol=1e-8,
rho=0.5, beta1=0.2):
    x = x0.copy()
    iteration = 0
    change = np.zeros(x.shape)

    while True:
        # Step choose
        beta2 = 1. -beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)

        # Momentum update
        proj = x + momentum * change
        gradient = gradf(proj)
        change = momentum * change - alpha * gradient
        x = x  + change

        # Stop criteria
        iteration += 1
        if np.linalg.norm(gradf(x)) < tol:
            break
        if iteration >= max_iter:
            break
    return x

def AcceleratedGD(f, gradf, x0, momentum =0.9, max_iter=1000, tol=1e-8,  rho=0.5, beta1=
    x = x0.copy()
    iteration = 0
    change = np.zeros(x.shape)

    while True:
        # Step choose
        beta2 = 1. -beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)

        # Momentum update
        gradient = gradf(x)
        change = momentum * change - alpha * gradient
        x = x  + change
```

```
        # Stop criteria
        iteration += 1
        if np.linalg.norm(gradf(x)) < tol:
            break
        if iteration >= max_iter:
            break
    return x
```

2.3 AdaGrad, RMSProp

Эти два метода определяют различную скорость сходимости для каждой координаты. Это увеличивает скорость схождимости для координат с небольшими значениями и уменьшает скорость обучения для параметров с большими. Данные методы хорошо показывают себя в задачах машинного обучения, а именно при неравномерном распределении данных.

Запишем шаг для AdaGrad:

$s_{i+1} = s_i + \nabla F(x_i)^2$

$x_{i+1} = x_i - \dfrac{\lambda \nabla F(x_i)}{\sqrt{s_{i+1}}}$

В RMSProp мы же делаем небольшую нормализацию, которая препятствует неограниченному росту s_t.

Запишем шаг для RMSProp:

$s_{i+1} = \gamma s_i + (1-\gamma)\nabla F(x_i)^2$

$x_{i+1} = x_i - \dfrac{\lambda \nabla F(x_i)}{\sqrt{s_{i+1}}}$

Rmsprop не всегда сходится из-за затухания и может остановиться не дойдя до экстремума, с другой стороны AdaGrad может постоянно перескакивать экстремум из-за возрастания s_t

Так же можно внести небольшую модификацию, которая поможет избавиться от деления на ноль. А именно будем делить на $\sqrt{s_{i+1} + \varepsilon}$

Приведем код на Python для этих алгоритмов:

```
def RMSProp(f, gradf, x0, epsilon, max_iter, rho=0.7, beta1=0.3, gamma = 0.9):
    x = x0.copy()
    iteration = 0
    s = 0
    while True:
        # Step choose
        beta2 = 1. -beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)
        gradient = gradf(x)
        s = gamma * s + (1 - gamma) * gradient ** 2
        x = x - alpha * gradient / np.sqrt(s)
        iteration += 1

        # Stop criteria
        if np.linalg.norm(gradf(x)) < epsilon:
            break
        if iteration >= max_iter:
            break
    return x

def AdaGrad(f, gradf, x0, epsilon, max_iter, rho=0.7, beta1=0.3):
    x = x0.copy()
    iteration = 0
    s = 0
    while True:
        # Step choose
```

```
        beta2 = 1. - beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)
        gradient = gradf(x)
        s = s + gradient ** 2
        x = x - alpha * gradient / np.sqrt(s)
        iteration += 1

        # Stop criteria
        if np.linalg.norm(gradf(x)) < epsilon:
            break
        if iteration >= max_iter:
            break
    return x
```

2.4 Adam

Теперь мы попробуем объединить идеи прошлых методов в одном алгоритме. В этом оптимизационном алгоритме используются скользящие средние как градиентов, так и вторых моментов градиентов.

Шаг:

$m_{i+1} \leftarrow \beta_1 m_i + (1-\beta_1)\nabla F(x_i)$

$v_{i+1} \leftarrow \beta_2 v_i + (1-\beta_2)\nabla F(x_i)^2$

$\hat{m}_i = \dfrac{m_{i+1}}{1-\beta_1^{i+1}}$

$\hat{v}_i = \dfrac{v_{i+1}}{1-\beta_2^{i+1}}$

$x_{i+1} = x_i - \eta \dfrac{\hat{m}_i}{\sqrt{\hat{v}_i}}$

Сюда так же можно добавить в деление малый ε, чтобы избежать деления на ноль.

Данный алгоритм является наиболее используемым в задачах машинного обучения.

Приведем код на Python для этого алгоритмов:

```
def Adam(f, gradf, x0, epsilon, max_iter, rho=0.7, beta1=0.3, b1, b2):
    x = x0.copy()
    iteration = 0
    s = 0
    m = 0
    v = 0
    while True:
        #                    ,
        beta2 = 1. - beta1
        alpha = backtracking (x, f, gradf, rho=rho, alpha0=1., beta1=beta1, beta2=beta2)

        gradient = gradf(x)
        m = b1 * m + (1 - b1) * gradient
        v = b2 * v + (1 - b2) * gradient ** 2

        m_hat = m / (1 - b1 ** (iteration + 1))
        v_hat = v / (1 - b2 ** (iteration + 1))

        x = x - alpha * m_hat / np.sqrt(v_hat)
        iteration += 1

        #
        if np.linalg.norm(gradf(x)) < epsilon:
            break
        if iteration >= max_iter:
            break
```

```
return x
```

3 Выбор шага для спуска

Для этих методов открытым остается вопрос - как выбрать шаг?

Существует несколько методов выбора шага:

- Константный шаг
- Постепенно убывающий шаг
- Метод наискорейшего спуска
- Правило Голдштейна-Армийо

Разберем каждый из вариантов выбора шага:

3.1 Константный шаг

Данный вариант прост в исполнении. Можно использовать при "несложных"функциях. При использовании в нейросетях - можно провести подбор шага на валидационных данных и использовать оптимальный в обучении. Конечно, данный метод нежелателен , потому что при больших значениях шага он может не сойтись, а при малых - сходиться очень долго.

Приведем две теоремы о сходимости градиентного спуска с константным шагом

Теорема 1 *Пусть* $\lambda^{[k]} = \lambda = const$*, функция* f *дифференцируема, ограничена снизу. Пусть выполняется условие Липшица для градиента* $f'(x) : ||f'(x) - f'(y)|| \leq L||x - y||$*. Пусть* $0 < \lambda < 2/L$*.*

Тогда $\lim_{k\to\infty} f'(x^{[k]}) = 0,\ f(x^{[k+1]}) < f(x^{[k]})$ *при любом выборе начального приближения.*

Определение

Дифференцируемая функция f называется сильно выпуклой (с константой $\Lambda > 0$), если для любых x и y из R^n справедливо

$f(x + y) \geq f(x) + \langle f'(x), y\rangle + \Lambda||y||^2/2$.

Теорема 2 *Пусть функция* f *дифференцируема, сильно выпукла с константой* Λ*. Пусть выполняется условие Липшица для градиента* $f'(x) : ||f'(x) - f'(y)|| \leq L||x - y||.0 < \lambda < 2/L$*.*

Тогда $\lim_{k\to\infty} x^{[k]} = x*,\ ||x^{[k]} - x*|| \leq q^k||x^{[0]} - x*||,\ q = \max\{|1 - \lambda\Lambda|, |1 - \lambda L|\}$ *при любом выборе начального приближения.*

Из этих теорем следует, что мы можем подбирать шаг, исходя из констант Λ и L. На самом деле скорость сходимости характеризуется величиной $\frac{L-\Lambda}{L+\Lambda}$, которую можно получить из оценок собственных значений гессиана. Но даже так, не зная подробной информации о минимизируемой функции сложно подобрать шаг основываясь на этих теоремах.

3.2 Убывающий шаг

Немного измененный прошлый вариант с поправкой на текущую итерацию. В зависимости от задачи можем брать такие варианты, как $\lambda_i = \frac{\lambda}{\sqrt{i+1}}$ или $\lambda_i = \frac{\lambda}{i+1}$. В данном случае мы так или иначе сойдемся, но при достаточно далекой начальной точке, мы можем застрять, не дойдя до экстремума.

3.3 Метод наискорейшего спуска

Данный метод основа на следующей идее - мы будем двигаться в сторону антиградиента, пока значение функции уменьшается. Мы выбираем шаг из соотношения $\lambda^{[k]} = \arg\min_{\lambda\in[0,\infty)} f(x^{[k]} - \lambda f'(x^{[k]})$.

Очевиден существенный недостаток метода - задача поиска шага может зачастую быть нелегче самой задачи минимизации. Однако данный момент можно применить в случае регрессии, в виду простого для дифференцирования и поиска оптимального значения данного функционала.

3.4 Правило Голдштейна-Армийо

Наиболее эффективный и полулярный выбор шага. Используется почти во всех солверах. Нужно найти такое $x_{k+1} = x_k - \lambda f'(x_k)$,что:

$\alpha\langle f'(x_k), x_k - x_{k+1}\rangle \leq f(x_k) - f(x_{k+1})$

$\beta\langle f'(x_k), x_k - x_{k+1}\rangle \geq f(x_k) - f(x_{k+1})$

Где $0 < \alpha < \beta < 1$ - некоторые константы.

Опишем данный метод геометрически. Рассмотрим функцию одной переменной : $\varphi(\lambda) = f(x - \lambda f'(x))$. Тогда нужное λ принадлежит той части графика функции φ, которая лежит между функциями $\phi_2(\lambda_k) = f(x_k) - \beta\lambda_k\|f'(x_k)\|^2$ и $\varphi_1(\lambda_k) = f(x_k) - \alpha\lambda_k\|f'(x_k)\|^2$. Таким образом подбираем шаг, пока не начнет выполняться условие: $\varphi 2(\lambda) \leq f(x_{k+1}) \leq \varphi 1(\lambda)$ Рассмотрим алгоритм, который позволяет находить эту самую λ.

```
def backtracking (x, f, grad_f, rho, alpha0, beta1, beta2):
    # 0 < rho < 1
    # alpha0 - initial guess of step size
    # beta1 and beta2 - constants from conditions
    alpha = alpha0
    phi1 = f(x) -  beta1 * alpha * grad_f(x).dot(grad_f(x))
    phi2 = f(x) -  beta2 * alpha * grad_f(x).dot(grad_f(x))
    f_k = f(x - alpha * grad_f(x))

    while not ((f_k <=phi1) and
               (f_k >=phi2)):
        alpha *= rho

        phi1 = f(x) -  beta1 * alpha * grad_f(x).dot(grad_f(x))
        phi2 = f(x) -  beta2 * alpha * grad_f(x).dot(grad_f(x))
        f_k = f(x - alpha * grad_f(x))

    return alpha
```

4 Заключение

В работе были рассмотрены методы первого порядка, начиная с самого простого, заканчивая наиболее актуальным. Было проведено сравнение методов различного порядка. Разобран выбор шага. В заключении хотелось бы сказать, что методы первого порядка в настоящее время очень важны, так как находят широкое применение в нейросетях для минимизации функции ошибок. В последнее время с ростом возможности вычислительной техники изучении данных методов продвинулось вперед. Поэтому важно развивать и улучшать уже имеющиеся техники и искать новые. Многие важные моменты были получены в последнее десятилетие. Удачи вам в ваших исследованиях!

Список литературы

[1] Методы выпуклой оптимизации. Ю. Е. Нестеров

[2] http://www.machinelearning.ru/wiki/index.php

[3] https://mipt.ru/online/diskretnaya-matematika/metody-optimizatsii-katrutsa-a-m-.php

[4] https://towardsdatascience.com

[5] https://ru.wikipedia.org/wiki