Национальный исследовательский университет ИТМО

Факультет информационных технологий и программирования

Прикладная математика и информатика

Методы оптимизации

Отчёт по лабораторной работе №2

Работу выполнили:

Ивченков Д. А., M32341

Султанов М. М., M32341

Трещёв А. С., M32341

Преподаватель:

Ким С. Е.

Санкт-Петербург

2023

Цель работы: изучение алгоритма стохастического градиентного спуска и его модификаций, исследование его работы на примере решения линейной регрессии

Задачи:

1. Реализовать стохастический градиентный спуск для решения линейной регрессии. Исследовать сходимость с разным размером батча (1 - SGD, 2, …, n-1 – Minibatch GD, n – GD из предыдущей работы).
2. Подобрать функцию изменения шага (learning rate scheduling), чтобы улучшить сходимость.
3. Исследовать модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).
4. Исследовать сходимость алгоритмов. Сравнить различные методы по скорости сходимости, надёжности, требуемым машинным ресурсам (объём оперативной памяти, кол-ву арифметических операций, времени выполнения).
5. Построить траекторию спуска различных алгоритмов из одной и той же исходной точки с одинаковой точностью.
6. Реализовать полиномиальную регрессию. Построить графики восстановленной регрессии для полиномов разной степени.
7. Модифицировать полиномиальную регрессию добавлением регуляризации в модель (L1, L2, Elastic регуляризации).
8. Исследовать влияние регуляризации на восстановление регрессии.

Использованные библиотеки:

- Numpy
- Matplotlib.pyplot

Ссылка на реализацию:

https://github.com/yoptimizationCt/lab2

Стохастический градиентный спуск для решения задачи линейной регрессии

В некоторых задачах оптимизации модель не может быть полностью определена, поскольку она зависит от величин, неизвестных в момент составления задачи. Такая характеристика присуща многим экономическим моделям, которые могут зависеть, например, от будущих процентных ставок, будущего спроса, будущего курса валют и прочих величин, чаще всего прогнозируемых и угадываемых. Вместо того, чтобы просто использовать наилучшее предположение, разработчики данных моделей могут включить в неё дополнительные знания о неизвестных величинах, например, если они знают ряд возможных сценариев и оценки вероятностей каждого сценария. Алгоритмы стохастической оптимизации используют эти количественные оценки неопределённости для получения решений, оптимизирующих ожидаемую производительность модели.

Связанные парадигмы для работы с неопределенными данными в модели включают оптимизацию с ограничением вероятности, при которой мы гарантируем, что переменные удовлетворяют заданным ограничениям с некоторой заданной вероятностью, и робастную оптимизацию (robust optimization), при которой определенные ограничения требуются для сохранения всех возможных значений неопределенных данных. Однако многие алгоритмы стохастической оптимизации исходят из формулирования одной или нескольких детерминированных подзадач, каждая из которых может быть решена с помощью методов оптимизации, изученных ранее.

В данной лабораторной работе нам предстоит решить задачу линейной регрессии. В общем случае она формулируется следующим образом:

Есть n числовых признаков $f_1(x), \dots, f_n(x)$, модель многомерной регрессии $f(x, \alpha) = \sum_{j=1}^{n} \alpha_j f_j(x)$, обучающая выборка из пар $(x_i, y_i)_{i=1\dots n}$, где $x_i \in R^n, y_i \in R$

Необходимо найти вектор α, такой, что $\sum_{i=1}^{n} (f(x_i, \alpha) - y_i)^2 \to min$

Например, при простом двумерном случае, рассматриваемом в этой лабораторной работе: у нас есть n точек на плоскости xOy с координатами (x_i, y_i), и мы ищем такую линейную функцию $f(x) = a \cdot x + b$, чтобы её график ближе всего находился к этим точкам, то есть минимизируем сумму расстояний от точек до прямой, или более формально: $g(a, b) = \sum_{i=1}^{n} (a \cdot x_i + b - y_i)^2 \to min$

Нетрудно увидеть, что на самом деле функция $g(a, b)$ является ничем иным, как квадратичной функцией, для которых в первой лабораторной работе реализовывался метод градиентного спуска с постоянным шагом и одномерным поиском. Логично предположить, что и здесь тоже можно использовать градиентный спуск, однако функция содержит n слагаемых, и при большом числе n вычисления будут производиться достаточно медленно.

Здесь нас и выручает стохастический градиентный спуск (далее SGD): дело в том, что на каждой эпохе вычисления, мы считаем градиент лишь у нескольких слагаемых функции, выбираемых случайным или закономерным образом (количество таких слагаемых называется **размером батча**)

Реализация стохастического градиентного спуска с разным размером батча:

```
# Считает значение заданного слагаемого в заданной точке
def calc_summand(x, y, point):
    value = -y
    for i in range(len(point)):
        value += point[i] * x ** i
    return value * value


def get_gradient(x, y, point, h=10e-6):
    gradient = np.zeros(len(point))
    for i in range(len(point)):
        delta = point.copy()
        delta[i] += h
        gradient[i] = (calc_summand(x, y, delta) - calc_summand(x, y, point))
/ h
    return gradient

# Считает градиент суммы заданных слагаемых (батча)
def sum_gradient(X, Y, point, summand_numbers):
    gradient = np.zeros(len(point))
    for i in summand_numbers:
        gradient += get_gradient(X[i], Y[i], point)
    return gradient

def gradient_descent(X, Y, start_point, learning_rate, epochs, batch_size):
    points = np.zeros((epochs, len(start_point)))
    points[0] = start_point
    lr = learning_rate
    for epoch in range(1, epochs):
        summand_numbers = [(epoch * batch_size + j) % len(X) for j in
range(batch_size)]
        points[epoch] = points[epoch - 1] - lr * sum_gradient(X, Y,
points[epoch - 1], summand_numbers)
        # Step decay
        if epoch % 100 == 0:
            lr *= 0.5
    return points
```

Также, как видно из следующих строк кода:

```
if epoch % 100 == 0:
    lr *= 0.5
```

мы реализовали функцию изменения шага (learning rate scheduling), чтобы улучшить сходимость градиентного спуска, а именно, каждые 100 шагов шага уменьшается в 2 раза.

Ниже представлены траектории градиентного спуска для разных размеров батчей, а также с использованием и без использования learning_rate scheduling. Было взято изначально 50 точек и learning_rate = 0.009, а также 1000 эпох. Начальной точкой была принята точка (0; 0)

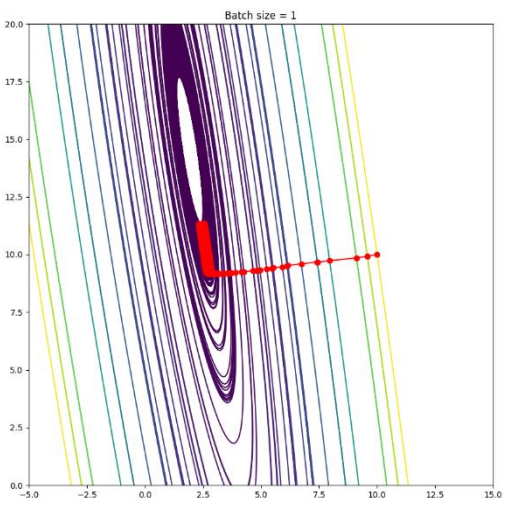

Рисунок 1 Размер батча = 1, неизменяемый шаг

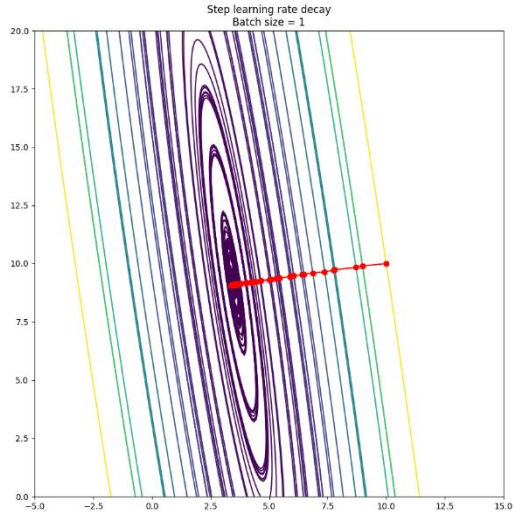

Рисунок 1 Размер батча = 1, изменяемый шаг

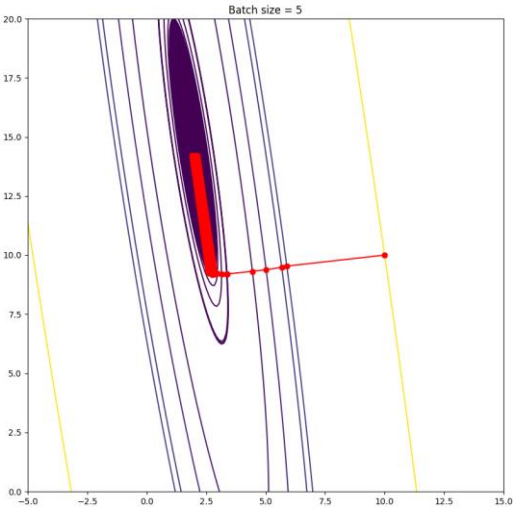

Рисунок 3 Размер батча = 5, неизменяемый шаг

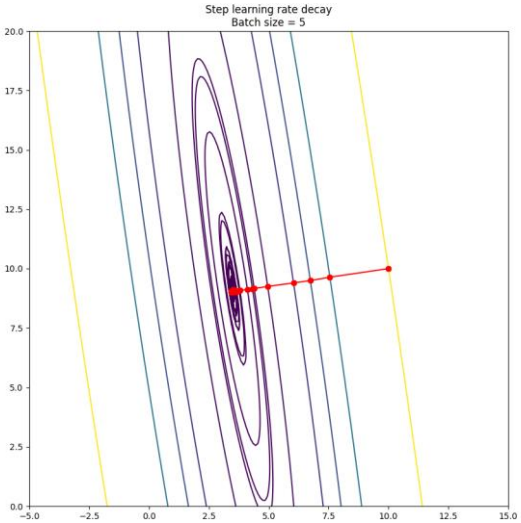

Рисунок 4 Размер батча = 5, изменяемый шаг

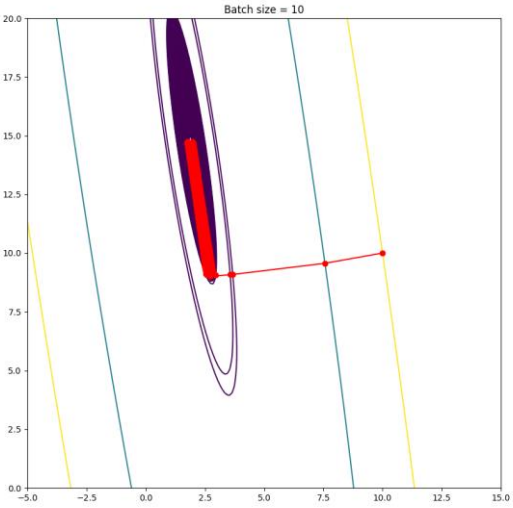

Рисунок 5 Размер батча = 10, неизменяемый шаг

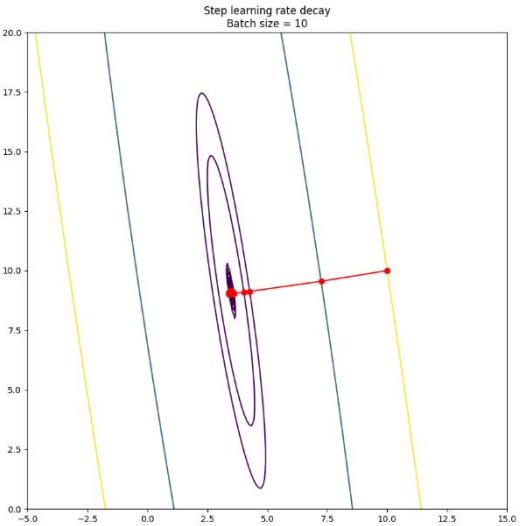

Рисунок 6 Размер батча = 10, изменяемый шаг

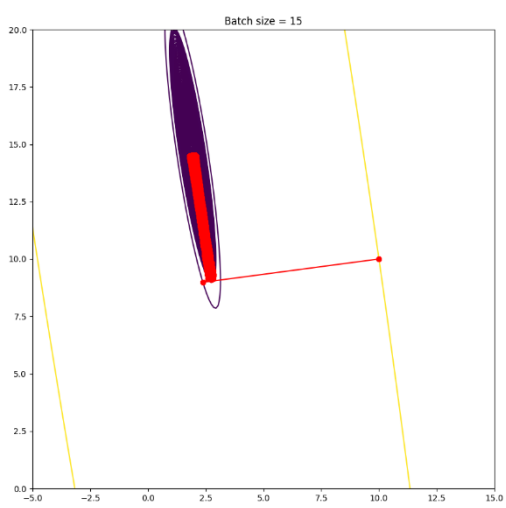

Рисунок 7 Размер батча = 15, неизменяемый шаг

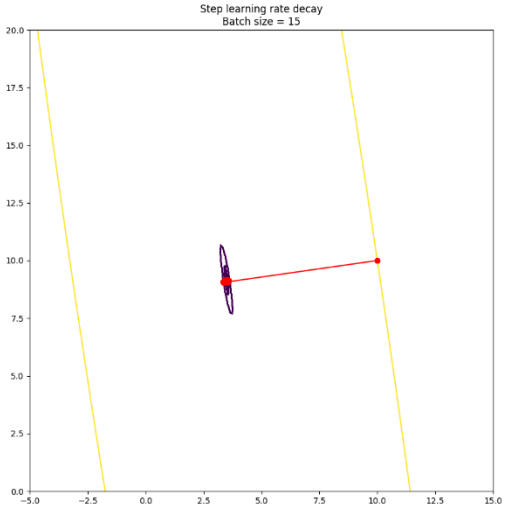

Рисунок 8 Размер батча = 15, изменяемый шаг

Видно, что при увеличении размера батча сходимость стохастического градиентного спуска ускоряется, а различные неточности и отклонения уменьшаются. Ступенчатое уменьшение шага также заметно влияет на скорость сходимости.

Модификации градиентного спуска

1. Momentum (метод инерции)

Лучше всего начать с физической аналогии: представим, что у нас есть мячик, который катится с горы. Мячик не застревает перед небольшой кочкой, так как у него есть некоторая масса и импульс, более того, под действием инерционных сил он способен некоторое время двигаться вверх горы против силы тяготения. Аналогичный приём может быть использован и в градиентной оптимизации.

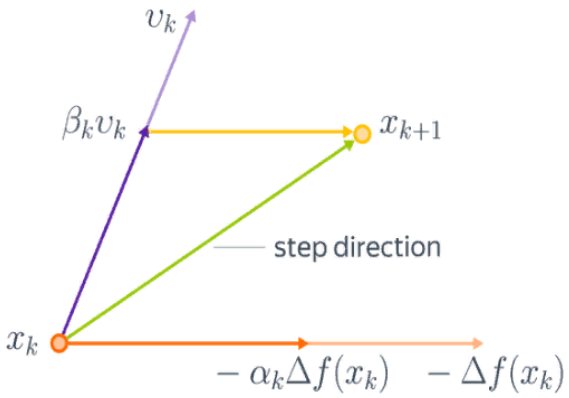

Рисунок 9 – визуализация работы Momentum

С математической точки зрения мы добавляем к изменению текущей точки ещё одно слагаемое: $x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k(x_k - x_k - 1)$ (рис. 19)

Заметим, что мы немного усугубили ситуацию тем, что теперь нам надо подбирать не только α_k но и β_k. Тогда для SGD получим небольшую выгоду тем, что будем сглаживать отступы по координатным осям от нужного нам значения и быстрее доберёмся до нужного ответа (рис.20)

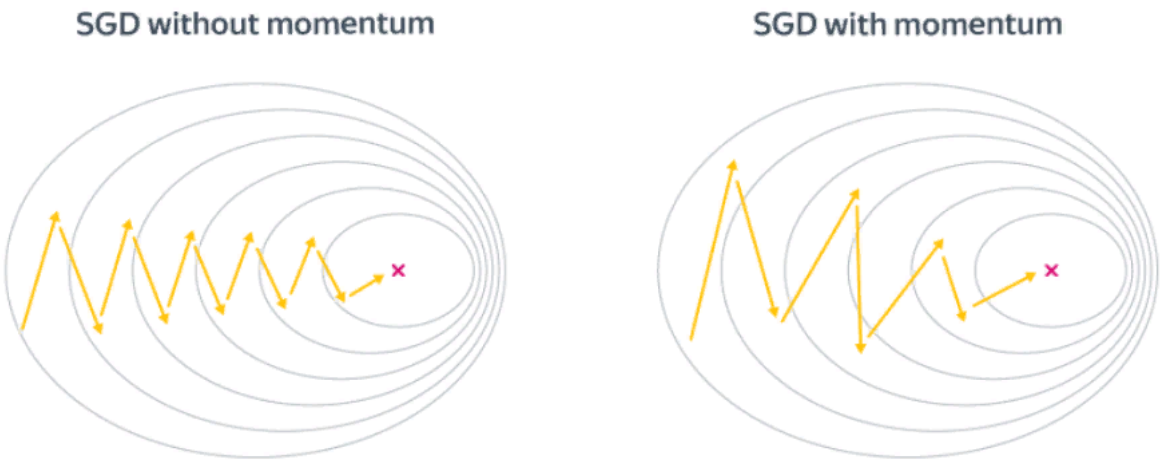

Рисунок 10 – траектории градиентного спуска с использованием Momentum и без

Также иногда удобно бывает представить метод моментума в виде двух параллельных итерационных процессов:

$$v_{k+1} = \beta_k v_k - \alpha_k \nabla f(x_k)$$

$$x_{k+1} = x_k + v_{k+1}$$

Программная реализация Momentum представлена ниже:

```
for epoch in range(1, 100):
    j = np.random.randint(0, n)
    v = gamma * v + (1 - gamma) * get_gradient(X[j], Y[j], point)
    point = point - lr * v
```

2. Nesterov Momentum

В 1983 году Ю. Нестеровым был предложен следующий алгоритм: модифицируем немного Momentum и будем считать градиент не в текущей точке, а в той точке, в которую мы бы пошли, следуя импульсу:

$$v_{k+1} = \beta_k v_k - \alpha_k \nabla f(x_k + \beta_k v_k)$$

$$x_{k+1} = x_k + v_{k+1}$$

Сравним с Momentum (рис. 21)

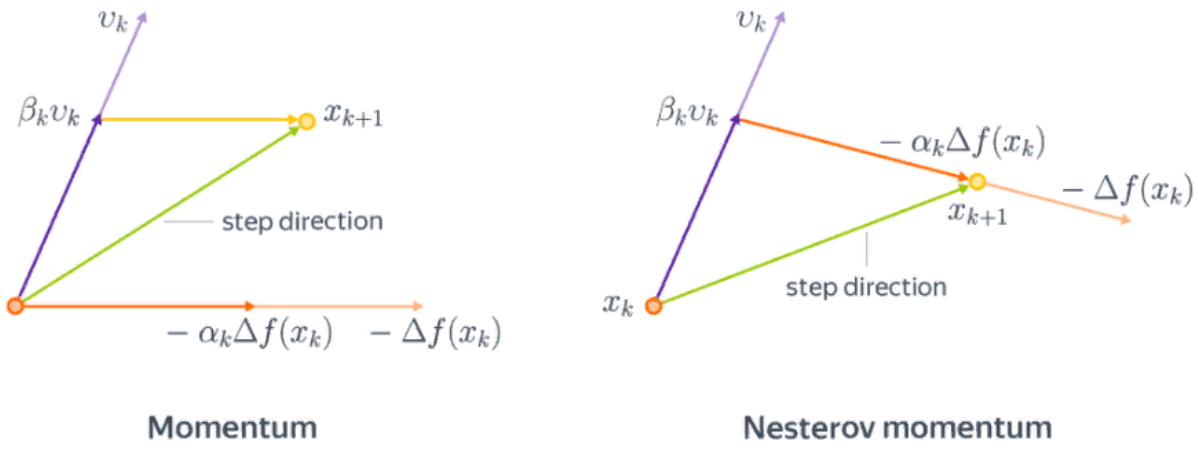

Рисунок 11 – сравнение методов Momentum и Nesterov momentum

Nesterov Momentum позволяет значительно повысить устойчивость и скорость сходимости в некоторых случаях, как бы «заглядывая в будущее» и исправляя ошибки на данном шаге оптимизации.

Программная реализация Nesterov:

```
for epoch in range(1, 100):
    j = np.random.randint(0, n)
    v = gamma * v + (1 - gamma) * get_gradient(X[j], Y[j], point -
lr*gamma*v)
    point = point - lr * v
```

3. AdaGrad

В предыдущих пунктах мы пытались каким-либо образом эксплуатировать свойства градиентного спуска, теперь же мы попытаемся подойти к улучшению алгоритма с точки зрения выбора размера шага. Первый алгоритм называется Adagrad – он является адаптацией стохастического градиентного спуска. Идея следующая: если мы «вышли на плато» по какой-то координате и соответствующая компонента градиента начала затухать, то нельзя уменьшать размер шага слишком сильно, так как мы рискуем не дойти до правильного ответа, лежащего вне «плато», но в то же время уменьшать шаг надо, потому что на этом «плато» может и лежать оптимальное значение. Если же градиент долгое время довольно большой, то это может быть знаком, что нам нужно уменьшить размер шага, чтобы не пропустить оптимум. Поэтому мы стараемся компенсировать слишком большие или слишком маленькие координаты градиента.

Более формально это выглядит так: зафиксируем α – исходный learning_rate. Напишем следующую формулу:

$$G_{k+1} = G_k + \left(\nabla f(x_k)\right)^2$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k)$$

Возведение в квадрат и деление векторов покомпонентные. На практике ε остаётся постоянным и равным 10^{-8}. По сути, мы начинаем динамически подбирать нужный нам learning_rate для каждого шага.

Программная реализация Adagrad:

```
state_sum = 0
for epoch in range(1, 100):
    j = np.random.randint(0, n)
    gr = get_gradient(X[j], Y[j], point)
    state_sum += gr*gr
    point = point - lr * gr / (np.sqrt(state_sum) + eps)
```

Но довольно часто так получается, что размер шага уменьшается слишком быстро и для решения этой проблемы придумали другой алгоритм.

4. RMSProp

Модифицируем предыдущую идею - будем не просто складывать нормы градиентов, а усреднять их:

$$G_{k+1} = \gamma G_k + (1-\gamma)\left(\nabla f(x_k)\right)^2$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k)$$

Такой выбор позволяет учитывать историю градиентов, но при этом размер шага уменьшается не так быстро

Программная реализация RMSProp:

```
s = 0
for epoch in range(1, 100):
    j = np.random.randint(0, n)
    gr = get_gradient(X[j], Y[j], point)
    s = alpha * s + (1 - alpha) * (gr * gr)
    point = point - lr * gr / (np.sqrt(s + eps))
```

Таким образом, адаптивный подбор шага позволяет выбрать лишь изначальное значение learning_rate, а всё остальное сделает сам алгоритм, но всё ещё стоит аккуратно выбирать, так как алгоритм может либо преждевременно выйти на плато, либо вовсе разойтись

5. Adam

А теперь объединим всё вышеизложенное в один алгоритм – Adam (Adaptive momentum), считающийся почти лучшим алгоритмом в решении задач SGD. Математически, это выглядит следующим образом:

$$v_{k+1} = \beta_1 v_k + (1-\beta_1)\nabla f(x_k)$$

$$G_{k+1} = \beta_2 G_k + (1-\beta_2)\left(\nabla f(x_k)\right)^2$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} v_{k+1}$$

Как правило, в данном алгоритме подбирают лишь один параметр α – learning_rate, а остальные $\beta_1, \beta_2, \varepsilon$ берут стандартными значениями $0.9, 0.99, 10^{-8}$ соответственно. Подбор learning_rate и является главной целью данного алгоритма.

Зачастую learning_rate берут равным $3e^{-4}$, однако иногда случаются и отступления от этого правила, в зависимости от поставленной задачи. Также следует помнить, что Adam требует хранения как параметров модели, так и градиентов, накопленного импульса и нормировочных констант, поэтому достижение более быстрой скорости решения задачи требует большего объёма памяти, что иногда может сильно сказаться на способностях модели к дообучению.

Программная реализация Adam:

```
s = 0
v = 0
for epoch in range(1, 100):
    j = np.random.randint(0, n)
    gr = get_gradient(X[j], Y[j], point)
    v = beta1 * v + (1 - beta1) * gr
    s = beta2 * s + (1 - beta2) * (gr * gr)
    vv = v / (1 - beta1 ** (epoch + 1))
    ss = s / (1 - beta2 ** (epoch + 1))
    point = point - lr * vv / (np.sqrt(ss + eps))
```

Сходимость алгоритмов

Сравним работу различных методов, используя разные learning_rate, результаты вычислений представлены в таблицах 1-3, для улучшения оценки все алгоритмы запускались на функции для линейной регрессии, использовавшей 100 точек, на выполнение давалось 10000 эпох, начальная точка (0; 0):

Таблица 1 – learning rate = 0.009

Критерий сравнения	SGD	Momentum	Nesterov	Adagrad	RMSProp	Adam
Среднее количество итераций	8925	3280	5242	3067	3535	2129
Среднее количество вычислений	2495	49209	94368	46014	58739	66008
Среднее время работы алгоритма, мс	85.956	41.739	76.694	43.401	51.285	45.443

Таблица 2 – learning rate = 0.09

Критерий сравнения	SGD	Momentum	Nesterov	Adagrad	RMSProp	Adam
Среднее количество итераций	9825	8228	7434	8858	7368	6452
Среднее количество вычислений	36141	93429	180184	45528	92311	257216
Среднее время работы алгоритма, мс	107.962	83.113	98.187	63.353	76.164	123.544

Таблица 3 – learning rate 0.03

Критерий сравнения	SGD	Momentum	Nesterov	Adagrad	RMSProp	Adam
Среднее количество итераций	7265	5829	2598	4272	1243	648
Среднее количество вычислений	30482	38979	104934	64089	42473	20148
Среднее время работы алгоритма, мс	113.040	33.884	84.473	61.768	15.599	13.287

Стоит понимать, что данные значения лишь приблизительно дают представления о реальной работе методов, однако хорошо видно, что в среднем методам Nesterov, RMSProp и Adam требуется меньше итераций чтобы дойти до оптимума, чем обычному SGD. При этом заметно, что методам Nesterov и Adam требуется значительно больше вычислений, что может заметно повлиять на их производительность. Также результаты работы методов сильно зависят от learning rate, а также от начальной точки и оптимизируемой функции. Таким образом, на практике, возникает множество различных условий, влияющих на скорость и работоспособность того или иного метода.

Траектория спуска алгоритмов из одной и той же исходной точкой и с одинаковой точностью.

Теперь протестируем то, каким же образом себя ведёт градиентный спуск на каждой из реализаций. Для первого примера возьмём исходную функцию y = 10x и сгенерируем 1000 точек с максимальным отклонением от этой функции в 1. И попробуем каждый из 6 алгоритмов на данной функции и посмотрим на то, как оно подбирает ответ для f (a, b) = ax + b, которая была бы ближе всего к сгенерированным точкам из исходной точки (0,0), lr для SGD, SGD with momentum, Nesterov = 0,0009, а для остальных модификаций = 1, батч использовался размером 1; gamma для Nesterov = 0.5; alpha для RMS_prop = 0.5; для Adam beta1 = 0.9, beta2 = 0.999, eps = 10e-8. Тогда результат будет следующим:

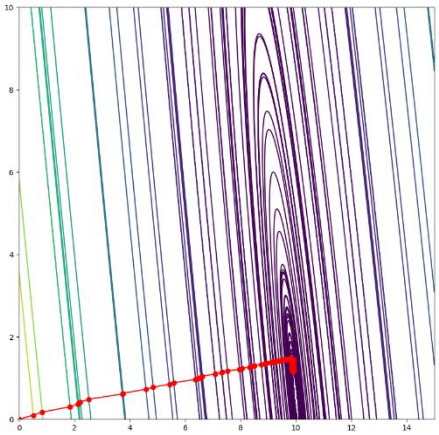

Рисунок 12 - SGD

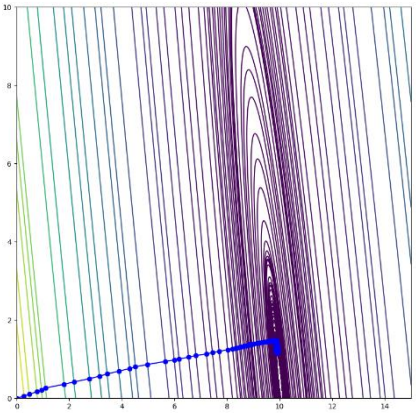

Рисунок 13 – SGD with momentum

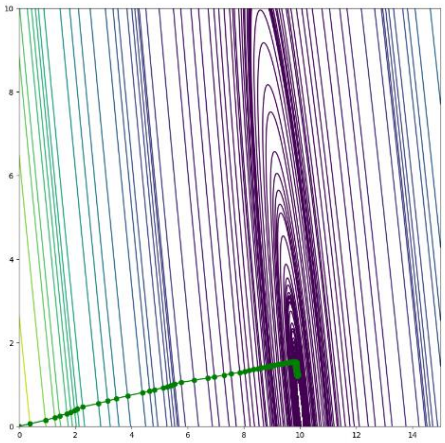

Рисунок 14 - Nesterov

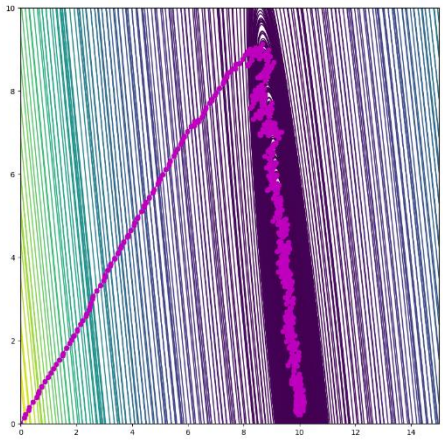

Рисунок 15 – RMSprop

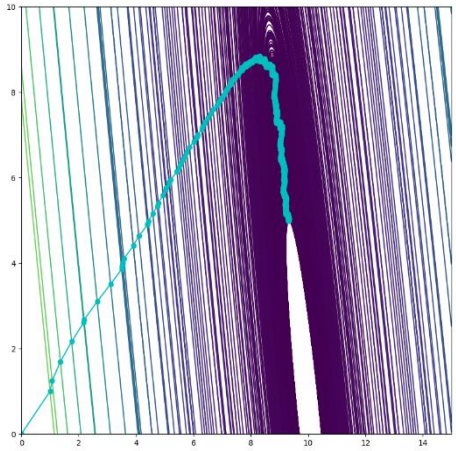

Рисунок 16 - AdaGrad

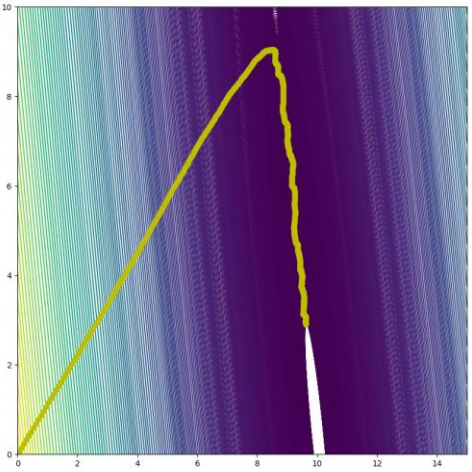

Рисунок 17 - Adam

Замечу, что спуск с моментумом действительно проходит быстрее, хоть это не сильно заметно вблизи ответа, но если посмотреть путь от исходной точки (0,0) до области, где ответ находится, то можно заметить, что с моментумом плотность пройденных линий уровней меньше. При этом заметим, что из-за большего lr модификации RMSprop, AdaGrad, Adam пошли немного по другой траектории, но благодаря этому легче заметить различия между ними, когда они рядом с ответом. RMSprop перепрыгивает ответ всё время в то время, как AdaGrad и Adam плавно к нему подходят. Это происходит из-за того, что в RMS_prop у нас альфа = 0.5, что уменьшает шаг на каждой эпохе по сравнению с AdaGrad и Adam.

Полиномиальная регрессия

Задача полиномиальной регрессии заключается в приближении какой-либо зависимости многочленом произвольной степени. В этом случае всё так же минимизируется сумма квадратов расстояний, но теперь вместо двух коэффициентов линейной зависимости a и b ищутся n + 1 коэффициентов многочлена n-ой степени $a_0, a_1 \ldots a_n$.

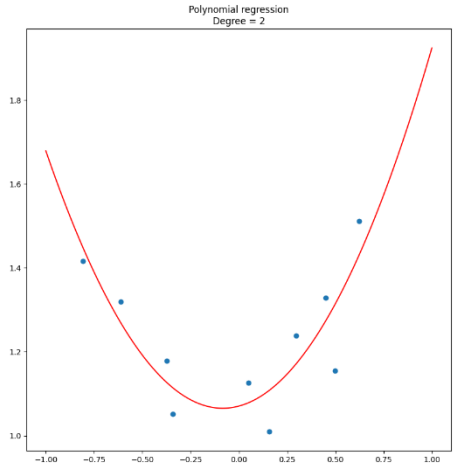

Рисунок 18 – Полиномиальная регрессия степени 2

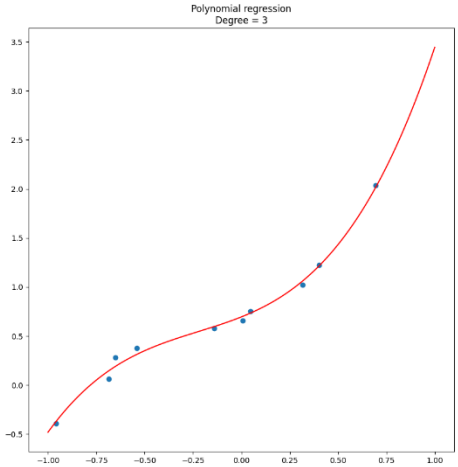

Рисунок 19 – Полиномиальная регрессия степени 3

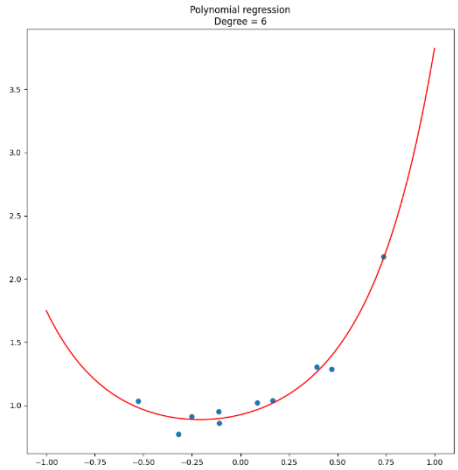

Рисунок 20 – Полиномиальная регрессия степени 6

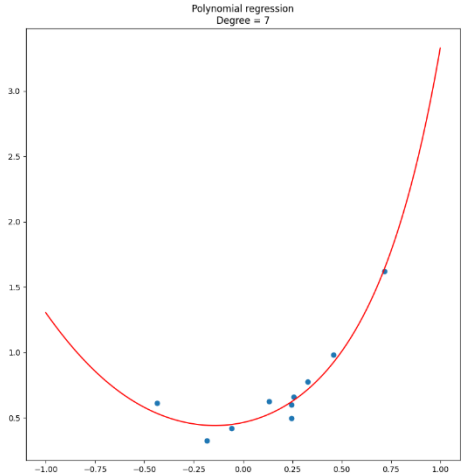

Рисунок 21 – Полиномиальная регрессия степени 7

Регуляризация

Регуляризация используется для борьбы с переобучением, т.е. для упрощения получаемой функции. Восстановленная полиномиальная регрессия может иметь большие по модулю веса, тогда функция будет более извилистой и её поведение будет сложнее. Чтобы получить упрощённую и сглаженную картину, соответствующую более общему случаю, можно ограничивать веса искомой функции, накладывая штраф за излишнюю сложность.

L1-регуляризация добавляет к минимизируемой функции октаэдрическую норму вектора весов, т.е. градиентный спуск одновременно с функцией минимизирует ещё и сумму модулей получаемых весов:

$$\|Xw - y\|_2^2 + \lambda\|w\|_1 \to min$$

w – искомые веса (коэффициенты) регресии,

$\|w\|_1 = |a_0| + \cdots + |a_n|$ – октаэдрическая норма,

λ – коэффициент регуляризации.

L2-регуляризация использует евклидову норму вектора весов:

$$\|Xw - y\|_2^2 + \lambda\|w\|_2^2 \to min$$

$\|w\|_2^2 = a_0^2 + \cdots + a_n^2$ – евклидова норма.

Elastic регуляризация является объединением L1 и L2 регуляризаций:

$$\|Xw - y\|_2^2 + \lambda_1\|w\|_1 + \lambda_2\|w\|_2^2 \to min$$

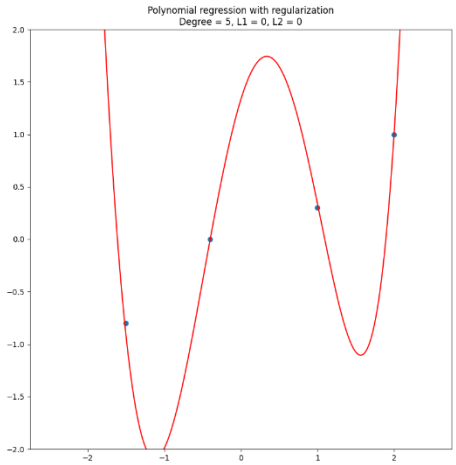

Рисунок 22 – Регрессия без регуляризации

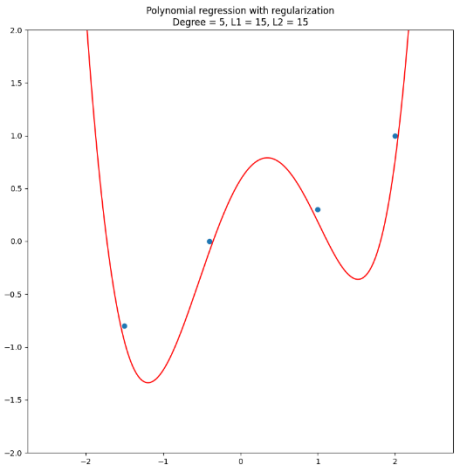

Рисунок 23 – Регрессия с регуляризацией l1 = 15, l2 = 15

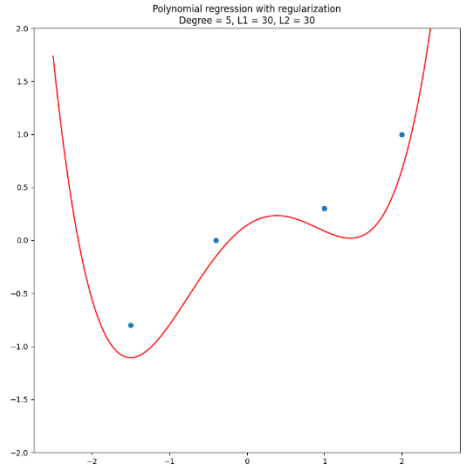

Рисунок 24 – Регрессия с регуляризацией l1 = 30, l2 = 30

Чем больше коэффициент регуляризации, т.е. штраф за увеличение веса высокий, тем меньшие веса могут быть выбраны, и наоборот, чем меньше коэффициент, т.е. штраф незначительный, тем больше могут быть расхождения.

L1-регуляризация может обнулять некоторые веса, если влияние соответствующих им параметров на функцию незначительно, поэтому данный вид регуляризации используется для отбора параметров. L2-регуляризация позволяет гладко и плавно

уменьшать веса. Elastic регуляризация сочетает в себе выгоду от обоих видов регуляризации – предотвращает переобучение и производит отбор параметров.