

Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации
Отчет по лабораторной работе №2

Работу выполнили:
Володько А.А., М32321
Ярунина К.А., М32371
Преподаватель:
Казанков В.К.



Санкт-Петербург
2023

Постановка задачи:

1. Реализуйте стохастический градиентный спуск для решения линейной регрессии. Исследуйте сходимость с разным размером батча (1 - SGD, 2, ..., $n - 1$ - Minibatch GD, n - GD из предыдущей работы).
 2. Подберите функцию изменения шага (learning rate scheduling), чтобы улучшить сходимость, например экспоненциальную или ступенчатую.
 3. Исследуйте модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).
 4. Исследуйте сходимость алгоритмов. Сравнить различные методы по скорости сходимости, надежности, требуемым машинным ресурсам (объем оперативной памяти, количеству арифметических операций, времени выполнения)
 5. Постройте траекторию спуска различных алгоритмов из одной и той же исходной точки с одинаковой точностью. В отчете наложить эту траекторию на рисунок с линиями равного уровня заданной функции.
- *.
- (a) Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.
 - (b) Модифицируйте полиномиальную регрессию добавлением регуляризации в модель ($L1$, $L2$, Elastic регуляризации).
 - (c) Исследуйте влияние регуляризации на восстановление регрессии.

1. Стохастический градиентный спуск для решения линейной регрессии

Что такое линейная регрессия...

Предположим, у нас есть некоторые данные $x = (x_1 \dots x_n)$, $y = (y_1 \dots y_n)$, такие, что y_i зависят от x_i ($y_i = \phi(x_i)$), и мы задаемся вопросом, как связаны x и y . Есть предположение, что они связаны плюс минус линейно, и наша задача найти эту линейную зависимость.

Итак, мы хотим приблизить прямую $\tilde{y} = w_1 + w_2 x$ к начальным данным, то есть найти такое w , при котором достигается минимум функции $f(w) := \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (w_1 + w_2 x_i - y_i)^2$.

Очевидно, это просто задача на поиск аргумента, где достигается минимум функции, которую мы пока помним как решать благодаря первой лабе (например, самый обычный градиентный спуск с постоянным шагом). Однако есть более интересный подход. В 1 лабораторной мы рассматривали разные способы оптимизации градиентного спуска, позволяющие существенно сократить количество итераций алгоритма (например, метод дихотомии). А что, если мы будем гнаться не за уменьшением количества итераций, а за скорость работы каждой из них? В этом заключается прикол стохастического градиентного спуска: тут мы на каждой итерации алгоритма для получения следующего приближения w вместо того, чтобы вычислять градиент на полном наборе данных в текущем приближении, будем считать его на некотором подмножестве этих данных. Что это значит: заметим, что наша функция $f(w) = \frac{1}{n} \sum_{i=1}^n f_i(w)$, тогда вместо $w = w - \alpha \nabla f(w)$ предлагается использовать $w = w - \alpha \nabla f_i(w)$, выбирая рандомные i на каждой итерации алгоритма (здесь в обеих формулах α – константа, постоянный шаг).

Во-первых, все-таки полагаться совсем на рандом идея сомнительная, поэтому в нашей реализации мы просто перемешаем f_i и будем их чередовать (иначе с небольшой, но все же существующей вероятностью может случиться так, что мы на каждой итерации будем выбирать один и тот же i , а это нехорошо, у нас и другие есть).

Во-вторых, возникает вопрос, когда остановиться. Останавливаться из-за небольшого последнего шага, как мы делали ранее, не прокатит – мы могли прийти к минимуму по некоторым координатам функции, а по остальным все еще находиться достаточно далеко от искомого значения. Тогда можно воспользоваться методом скользящего среднего – давайте вычислять среднее арифметическое квадратов последних k шагов, где k – размерность пространства. Тогда мы посмотрим на все предшествующие координаты и точно сможем сказать, что градиент по всем координатам достаточно уменьшился, то есть мы достаточно близко к минимуму.

Но и это еще не все: что такое **Minibatch**. Хороший вопрос, сначала введем определение. Batch – реализация градиентного спуска, когда на каждой итерации обучающая выборка просматривается целиком, и только после этого изменяются веса модели. Тогда в minibatch мы можем использовать только некоторую часть этой выборки, значит, описанный выше стохастический градиентный спуск использовал части размера 1. Но можно же и что-то между. Ну давайте, тогда $w = w - \alpha \nabla g_i(w)$, где $g_i(w) = \frac{1}{n} \sum_{j=1}^k f_{(ik+j)\%n}(w)$, где k – размер minibatch'a, и g_i рандомно перемешаем и будем чередовать.

И реализация:

```
def stochastic_gd(w: np.array, x: np.array, y: np.array, batch=1, lr=0.009, eps=1e-5, epoch=20000):
    dim = x.size
    i, iters, step, g_size, history = 0, 0, 1, ceil(dim / batch), [w],
    steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
    steps_history[0] = w

    def get_dgi(u):
        indexes = [(u * batch + j) % dim for j in range(batch)]
        return np.array([sum([2 * (ww[0] + ww[1] * x[j] - y[j]) for j in indexes]) / dim,
                        sum([2 * (ww[0] + ww[1] * x[j] - y[j]) * x[j] for j in indexes]) / dim])

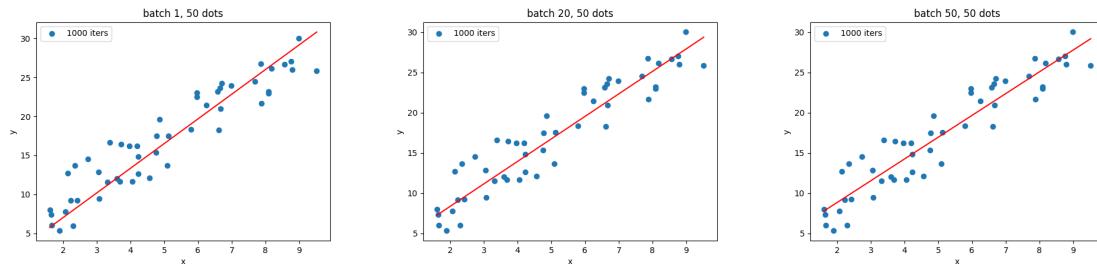
    def dgi(ww):
        return np.array([sum([2 * (ww[0] + ww[1] * x[j] - y[j]) for j in indexes]) / dim,
                        sum([2 * (ww[0] + ww[1] * x[j] - y[j]) * x[j] for j in indexes]) / dim])

    dg = [get_dgi(k) for k in range(g_size)]
    random.shuffle(dg)
    for i in range(epoch):
        i = (i + 1) % g_size
        step = dg[i](w) * lr
        steps_history[i] = np.abs(step)
        if np.sum(steps_history) / g_size < brk:
            break
        w = w - step
        iters += 1
        history.append(w)
    return np.array(history), iters
```

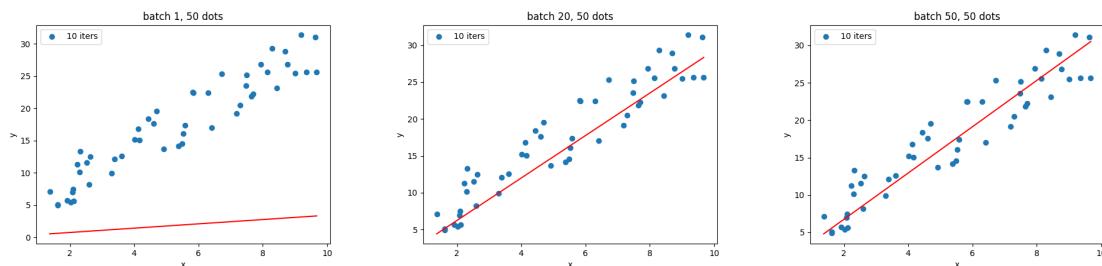
Это сокращает задействованные вычислительные ресурсы и позволяет достичь более высокой скорости итераций в обмен на более низкую скорость сходимости.

Чтобы протестировать, насколько корректно работает алгоритм, будем генерировать точки, соответствующие линейной регрессии некоторой функции (в данном примере $y = 2.6 \cdot x + 3.9$)

```
def generate_points(fun, num, disp=4):
    X = np.random.uniform(low=1, high=10, size=num)
    X.sort()
    Y = fun(X)
    for i in range(len(Y)):
        Y[i] = Y[i] + random.uniform(-disp, disp)
    return X, Y
```



Выше представлены графики найденной линейной регрессии за предельное количество итераций, в целом, функция достаточно простая и стохастический градиентный спуск справляется быстро, посмотрим на результаты за 10 итераций:



Здесь уже четко видно, что каждая итерация при максимальном размере батча (тогда стохастический спуск эквивалентен обычному градиентному) отрабатывает намного эффективнее по приближению, уже на 10 итерациях мы сильно приблизились к искомым значениям, однако итерации и работают медленнее.

2. Learning rate scheduling

Learning rate, или темп обучения, это параметр, определяющий размер шага алгоритма на каждой его итерации. До сих пор мы пользовались константным темпом, и нередко проигрывали: если темп обучения слишком большой, то можно перепрыгнуть через минимум и алгоритм разойдется, если слишком маленький, то работать он будет слишком медленно.

Оказывается, темп обучения можно изменять в зависимости от текущей итерации. Рассмотрим один из популярных методов изменения темпа обучения в соответствии с графиком – метод экспоненциального затухания, или **Exponential decay**. По названию несложно догадаться, что использовать мы будем убывающую экспоненциальную функцию. Формула для вычисления шага lr на итерации n тогда выглядит так: $lr = lr_0 \cdot e^{-nd}$, где lr_0 – изначальный темп обучения, а d – параметр затухания.

```
def constant_lr(initial_lr: float, _) -> float:
    return initial_lr

def exp_lr(initial_lr: float, iterations: int, rate=0.09) -> float:
    return initial_lr * exp(-rate * iterations)

def gd(f, x: np.array, lr, lr0=0.009, brk=1e-5):
    df = nd.Gradient(f)
    iterations, grad, history, steps = 0, 1, [np.hstack((x, f(x)))], []
    while np.linalg.norm(grad) > brk:
        iterations += 1
        grad = df(x)
```

```

step = lr(lr0, iterations)
steps.append(step)
x = x - grad * step
history.append(np.hstack((x, f(x))))
return np.array(history), iterations, steps

```

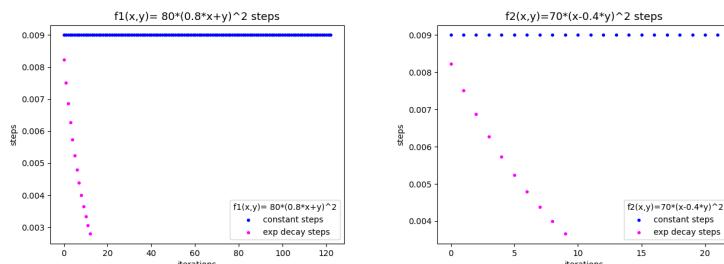
Сравним работу **exponential decay** с методом с константным шагом на примере следующих функций:

$$(a) f_1(x, y) = 80 \cdot (0.8 \cdot x + y)^2$$

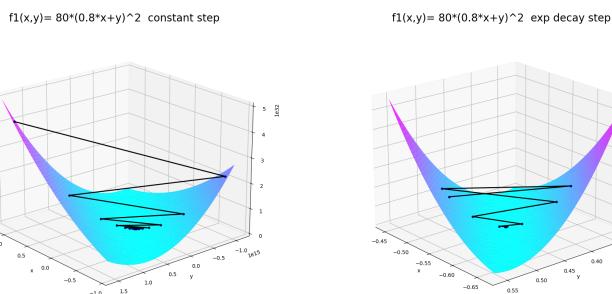
$$(b) f_2(x, y) = 70 \cdot (x - 0.4 \cdot y)^2$$

Начальная точка тут $(x, y) = (-0.5, 0.5)$, $lr_0 = 0.009$, $d = 0.09$.

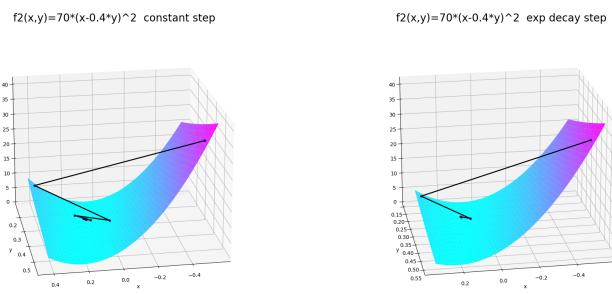
На следующих графиках показано, как менялись значения темпа обучения в зависимости от числа итераций в обоих случаях (действительно, экспоненциально убывает):



В случае с f_1 градиентный спуск с постоянным шагом разошелся, но с экспоненциальным затуханием смог сойтись.



При рассмотрении функции f_2 градиентный спуск сорвался в обоих случаях, но здесь мы можем наблюдать разницу в количестве итераций: с константным шагом их было 22, а с убывающим всего 10.



Ясно, что в случае, когда мы возьмем функцию и начальные параметры, подходящие для данного метода, мы сможем существенно сократить количество итераций алгоритма, избежать колебаний - зиг-загов или завершения в локальных минимумах, не соответствующих реальному значению минимума функции.

А теперь то же самое, на на стохастическом))))))

Реализация:

```

def gd(w: np.array, x: np.array, y: np.array, batch=1, lr=constant_lr, lr0=0.3, brk=1e-5, epoch=100):
    dim = x.size

```

```

i, iters, step, g_size, history = 0, 0, 1, ceil(dim / batch), [w]
iterations = 0
steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
steps = []

def get_dgi(u):
    indexes = [(u * batch + j) % dim for j in range(batch)]

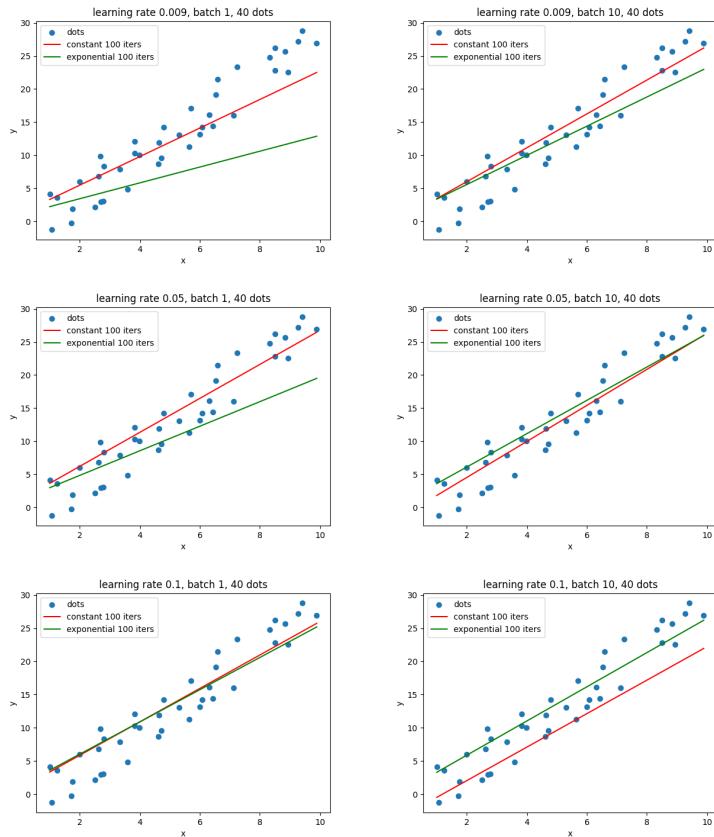
def dgi(ww):
    return np.array([sum([(2./dim) * (ww[0] + ww[1] * x[j] - y[j]) for j in indexes]),
                    sum([(2./dim) * (ww[0] + ww[1] * x[j] - y[j]) * x[j] for j in indexes])])

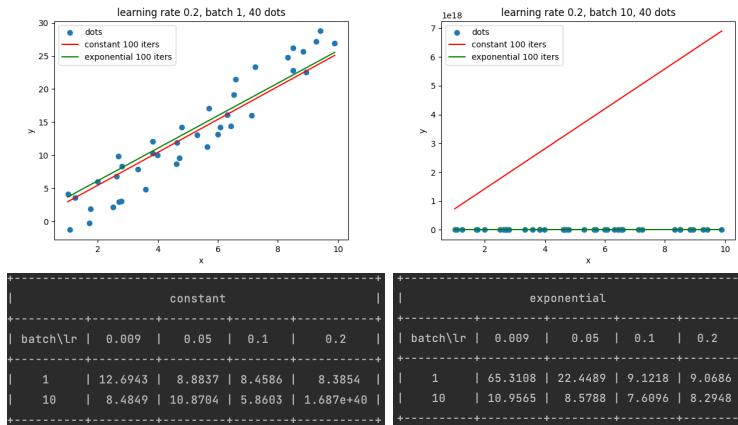
return dgi

dg = [get_dgi(k) for k in range(g_size)]
random.shuffle(dg)
for i in range(epoch):
    i = (i + 1) % g_size
    iterations += 1
    grad = dg[i](w)
    step = lr(lr0, iterations) * grad
    steps_history[i] = np.abs(step)
    if np.sum(steps_history) / g_size < brk:
        break
    steps.append(step)
    w = w - step
    history.append(w)
return np.array(history), iterations, steps

```

Посмотрим работу метода на примере линейной регрессии, заданной функцией $y = 3x + 2$, на 40 начальных точках, для батчей = 1 и 10 и констант для начального постоянного шага $\in \{0.009, 0.05, 0.1, 0.2\}$.





В целом прослеживается общая тенденция: для меньших значений начального **learning rate** константный шаг проявляет себя лучше, так как экспоненциальный «затухает» слишком быстро, поэтому до самого минимума дойти не успевает.

На больших же значениях экспоненциальный проявляется себя во всей красе: благодаря затуханию понижается вероятность перепрыгнуть через минимум, соответственно не появляется $e + 100$ в результатах. В таблицах, приведенных выше, представлены значения точки-результата метода в общей функции потерь, мы видим, что наименьшего результата нам удалось добиться при **batch=10** и **learning rate=0.2** – максимальных из рассматриваемых, на методе с экспоненциальным шагом, константный же шаг при этих же значениях вовсе разошелся.

3. Модификации градиентного спуска

Для начала рассмотрим такую штуку: **экспоненциальное скользящее среднее**. Она будет использоваться в следующих методах, есть смысл сразу ее пояснить.

Рассмотрим следующую формулу: $v_{i+1} = \gamma v_i + (1 - \gamma)g_i$, $\gamma \in (0, 1)$

Она используется для усреднения функции: если у нас была очень осциллирующая последовательность g , то в зависимости от γ мы получим менее осциллирующую последовательность v , то есть некое ее сглаживание. Чем больше γ , тем сильнее сглаживание, но, с другой стороны, особенно на первых итерациях, происходит некое смещение за счет того, что $v_0 = 0$. Чтобы с этим бороться, делаем коррекцию: $\tilde{v}_{i+1} = v_{i+1}/(1 - \gamma^i)$.

i. Momentum

Собственно, первая модификация: давайте воспользуемся экспоненциальным скользящим средним. Зачем это надо: если у нас есть функция, в которой градиент осциллирует, мы можем его сгладить, (ну например, это уберет скачки спуска). Получим градиентный спуск, работающий по следующей логике:

$$v_{i+1} = \gamma v_i + (1 - \gamma)\nabla f(x_i)$$

$$x_{i+1} = x_i - lr \cdot v_{i+1}$$

```

def get_dgi(u, batch, dim, x, y):
    indexes = [(u * batch + j) % dim for j in range(batch)]

    def dgi(ww):
        return np.array([sum([2 * (ww[0] + ww[1] * x[j] - y[j]) for j in indexes]) / dim,
                       sum([2 * (ww[0] + ww[1] * x[j] - y[j]) * x[j] for j in indexes]) / dim])

    return dgi

def momentum(w: np.array, x: np.array, y: np.array, batch=100, gamma=0.6, lr=0.009, brk=1e-5, epoch=1000): # 5
    ↵    arithmetics * x.size * iter
    dim = x.size
    i, step, g_size, history = 0, 1, ceil(dim / batch), [w]
    iters = 0
    steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
    v = np.zeros(w.shape)
    dg = [get_dgi(k, batch, dim, x, y) for k in range(g_size)]
    random.shuffle(dg)
    for _ in range(epoch):
        i = (i + 1) % g_size
        v = gamma * v + dg[i](w) * (1 - gamma)
        step = lr * v
        steps_history[i] = np.abs(step)

```

```

    if np.sum(steps_history) / g_size < brk:
        break
    w = w - step
    iters += 1
    history.append(w)
return np.array(history), iters

```

ii. Nesterov

Модификация предыдущего метода, теперь мы будем считать градиент не на текущих весах, а в некотором приближении того, куда хотим шагнуть.

$$v_{i+1} = \gamma v_i + lr \nabla f(x_i - \gamma v_i)$$

$$x_{i+1} = x_i - v_{i+1}$$

```

def nesterov(w: np.array, x: np.array, y: np.array, batch=100, gamma=0.9, lr=0.0009, brk=1e-5, epoch=1000): # 6
    ↪ arithmetics * x.size * iter
    dim = x.size
    i, step, g_size, history = 0, 1, ceil(dim / batch), [w]
    iters = 0
    steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
    v = np.ones(w.shape)
    dg = [get_dgi(k, batch, dim, x, y) for k in range(g_size)]
    random.shuffle(dg)
    for _ in range(epoch):
        i = (i + 1) % g_size
        v = gamma * v + dg[i](w - gamma * v) * lr
        steps_history[i] = np.abs(v)
        if np.sum(steps_history) / g_size < brk:
            break
        w = w - v
        iters += 1
        history.append(w)
    return np.array(history), iters

```

iii. AdaGrad

Стандартная формула из градиентного спуска с постоянным шагом: $x_{i+1} = x_i - lr \nabla f(x_i)$. Зачастую (особенно с функциями с высоким числом обусловленности) выходит, что по какой-то координате скачки большие, а по какой-то нет. Чтобы от этого избавиться, вводим матрицу квадратов градиентов и делим на нее шаг. Однако тут же возникает проблема: получился learning rate scheduling, причем мы не можем никак влиять на его скорость.

$$\begin{aligned} g_{i+1} &= \nabla f(x_i) \\ G_{i+1} &= G_i + g_{i+1}^2 \\ x_{i+1} &= x_i - \frac{lr \cdot g_{i+1}}{\sqrt{G_{i+1} + \varepsilon}} \end{aligned}$$

```

def adagrad(w: np.array, x: np.array, y: np.array, batch=100, lr=1, epsilon=1e-6, brk=1e-5, epoch=1000): # 6
    ↪ arithmetics * x.size * iter
    dim = x.size
    i, step, g_size, history = 0, 1, ceil(dim / batch), [w]
    iters = 0
    steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
    g = np.zeros(w.shape)
    dg = [get_dgi(k, batch, dim, x, y) for k in range(g_size)]
    random.shuffle(dg)
    for _ in range(epoch):
        i = (i + 1) % g_size
        grad = dg[i](w)
        g += grad ** 2
        step = grad * (lr / np.sqrt(g + epsilon)) # чтобы не было деления на ноль, epsilon - сглаживающий
        ↪ параметр
        steps_history[i] = np.abs(step)
        if np.sum(steps_history) / g_size < brk:
            break
        w = w - step
        iters += 1
        history.append(w)
    return np.array(history), iters

```

iv. RMSProp

Здесь будем пытаться регулировать скорость получившегося learning rate'а на основе квадратов градиентов, но уже с применением экспоненциального производящего среднего

$$g_{i+1} = \nabla f(x_i)$$

$$\begin{aligned} E_{i+1} &= \gamma E_i + (1 - \gamma) g_{i+1}^2 \\ x_{i+1} &= x_i - \frac{lr \cdot g_{i+1}}{\sqrt{E_{i+1} + \epsilon}} \end{aligned}$$

```
def rms_prop(w: np.array, x: np.array, y: np.array, batch=100, gamma=0.975, lr=0.01, epsilon=1e-6, brk=1e-5,
← epoch=1000): # 6 arithmetics * x.size * iter
dim = x.size
i, step, g_size, history = 0, 1, ceil(dim / batch), [w]
iters = 0
steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
moving_avg = np.zeros(w.shape)
dg = [get_dgi(k, batch, dim, x, y) for k in range(g_size)]
random.shuffle(dg)
for _ in range(epoch):
    i = (i + 1) % g_size
    grad = dg[i](w)
    moving_avg = gamma * moving_avg + (1 - gamma) * grad ** 2
    step = grad * (lr / np.sqrt(moving_avg + epsilon))
    steps_history[i] = np.abs(step)
    if np.sum(steps_history) / g_size < brk:
        break
    w = w - step
    iters += 1
    history.append(w)
return np.array(history), iters
```

v. Adam

Фактически просто объединяя несколько описанных ранее модификаций

$$g_{i+1} = \nabla f(x_i)$$

$$\begin{aligned} m_{i+1} &= \gamma_m m_i + (1 - \gamma_m) g_{i+1}, v_{i+1} = \gamma_v v_i + (1 - \gamma_v) g_{i+1}^2 \\ \tilde{m}_{i+1} &= \frac{m_{i+1}}{1 - \gamma_m}, \tilde{v}_{i+1} = \frac{v_{i+1}}{1 - \gamma_v} \\ x_{i+1} &= x_i - \frac{lr \cdot \tilde{m}_{i+1}}{\sqrt{\tilde{v}_{i+1} + \epsilon}} \end{aligned}$$

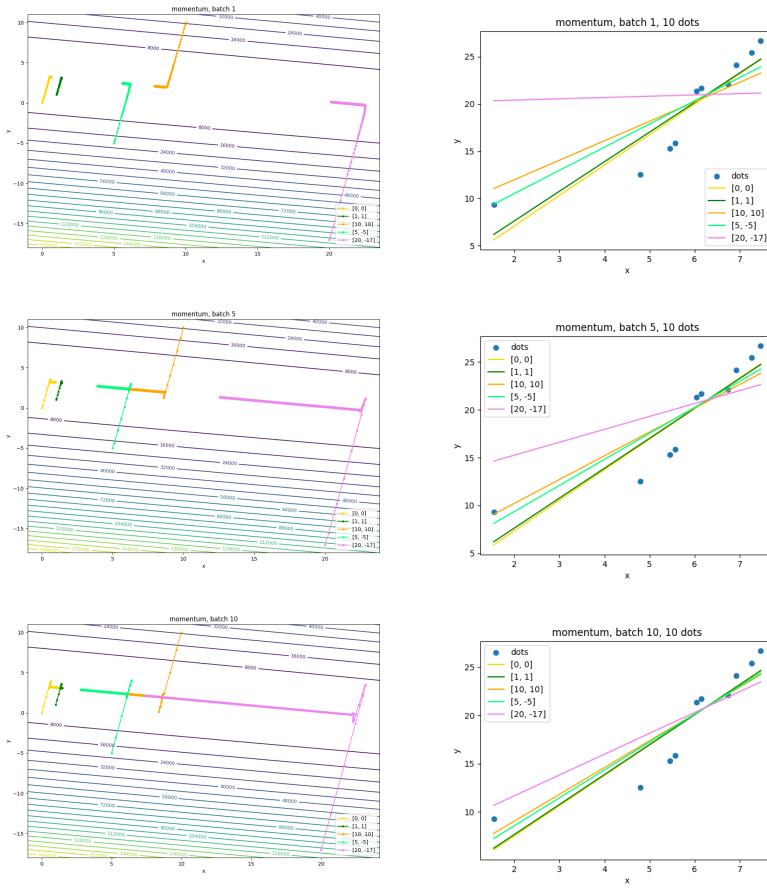
```
def adam(w: np.array, x: np.array, y: np.array, batch=100, gamma_m=0.9, gamma_v=0.999, lr=0.001, epsilon=1e-8,
← brk=1e-5, epoch=10000):
dim = x.size
i, step, g_size, history = 0, 1, ceil(dim / batch), [w]
iters = 0
steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
m = np.zeros(w.shape)
v = np.zeros(w.shape)
dg = [get_dgi(k, batch, dim, x, y) for k in range(g_size)]
random.shuffle(dg)
for ep in range(1, epoch + 1):
    i = (i + 1) % g_size
    grad = dg[i](w)
    m = gamma_m * m + (1 - gamma_m) * grad
    v = gamma_v * v + (1 - gamma_v) * grad ** 2
    mm = m / (1 - gamma_m ** ep)
    vv = v / (1 - gamma_v ** ep)
    step = lr * mm / (np.sqrt(vv) + epsilon)
    steps_history[i] = np.abs(step)
    if np.sum(steps_history) / g_size < brk:
        break
    w = w - step
    iters += 1
    history.append(w)
return np.array(history), iters
```

4. Сходимость алгоритмов и траектории спуска

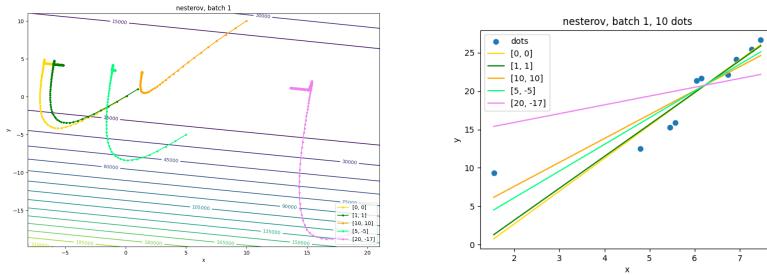
Для исследования алгоритмов построим 10 точек вокруг прямой $y = 3x + 2$ для получения соответствующей линейной регрессии и посмотрим, как ведут себя вышеописанные алгоритмы в работе с ней на разных начальных точках.

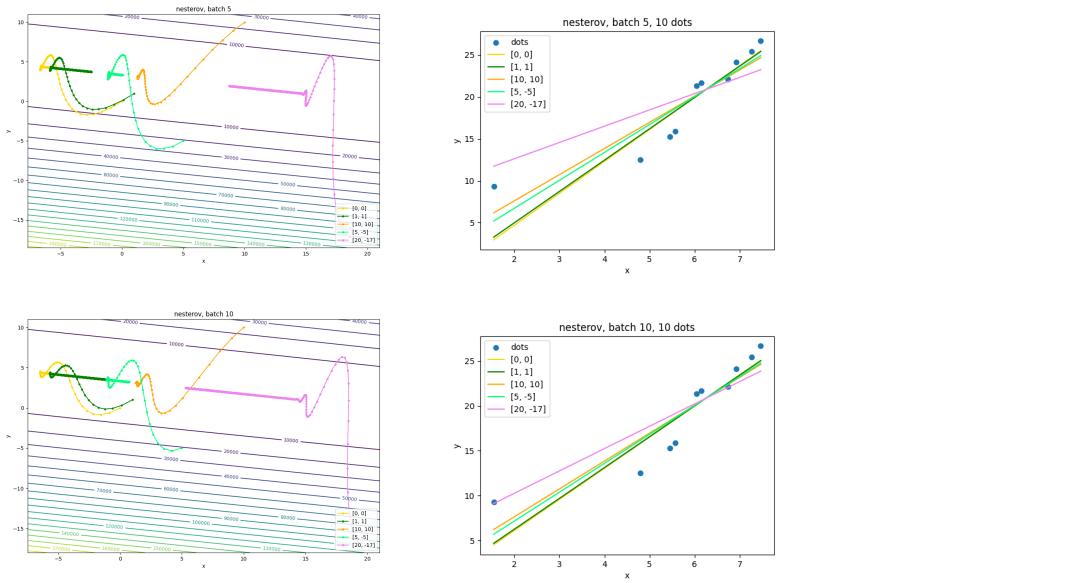
Здесь мы взяли 3 размера батча: 1, 5, 10, и 5 разных начальных точек: $(x, y) \in \{(0, 0), (1, 1), (10, 10), (5, -5), (20, -17)\}$, в таблицах посмотрим на наиболее близкий к искомым значениям результат, наиболее далеких от них, и средний.

i. Momentum

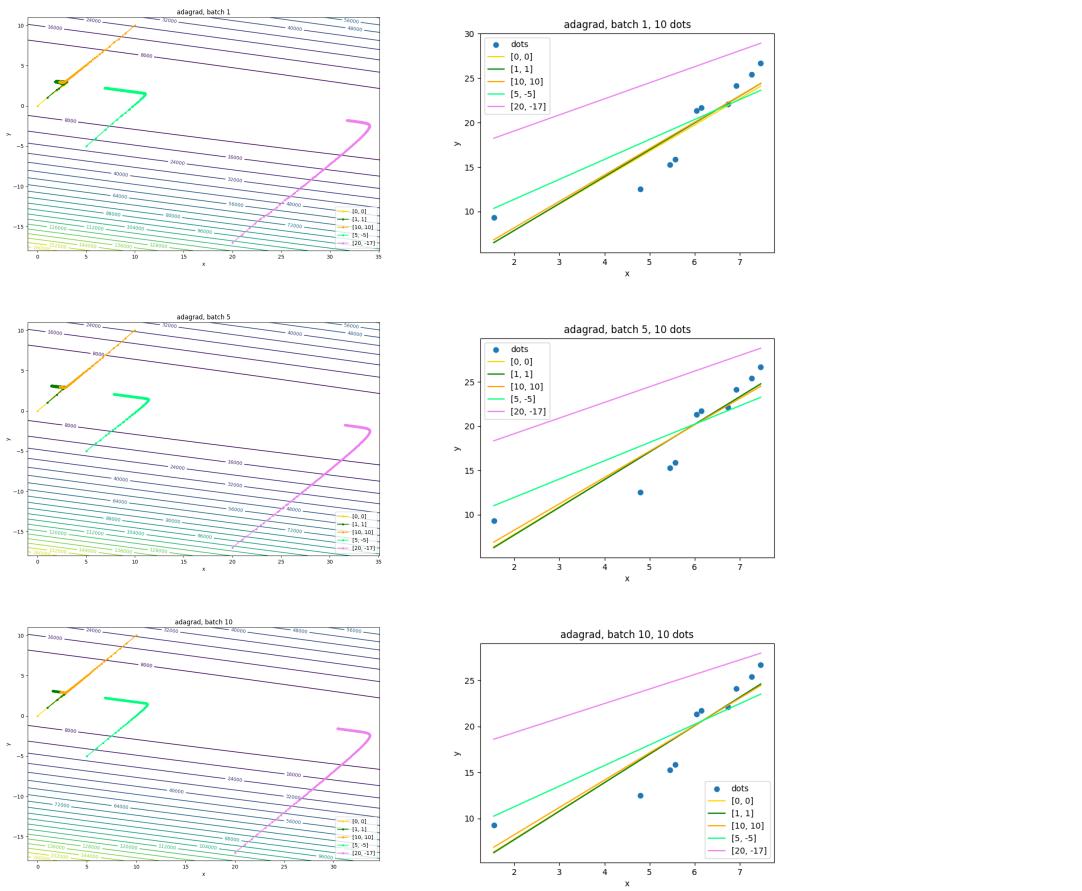


ii. Nesterov

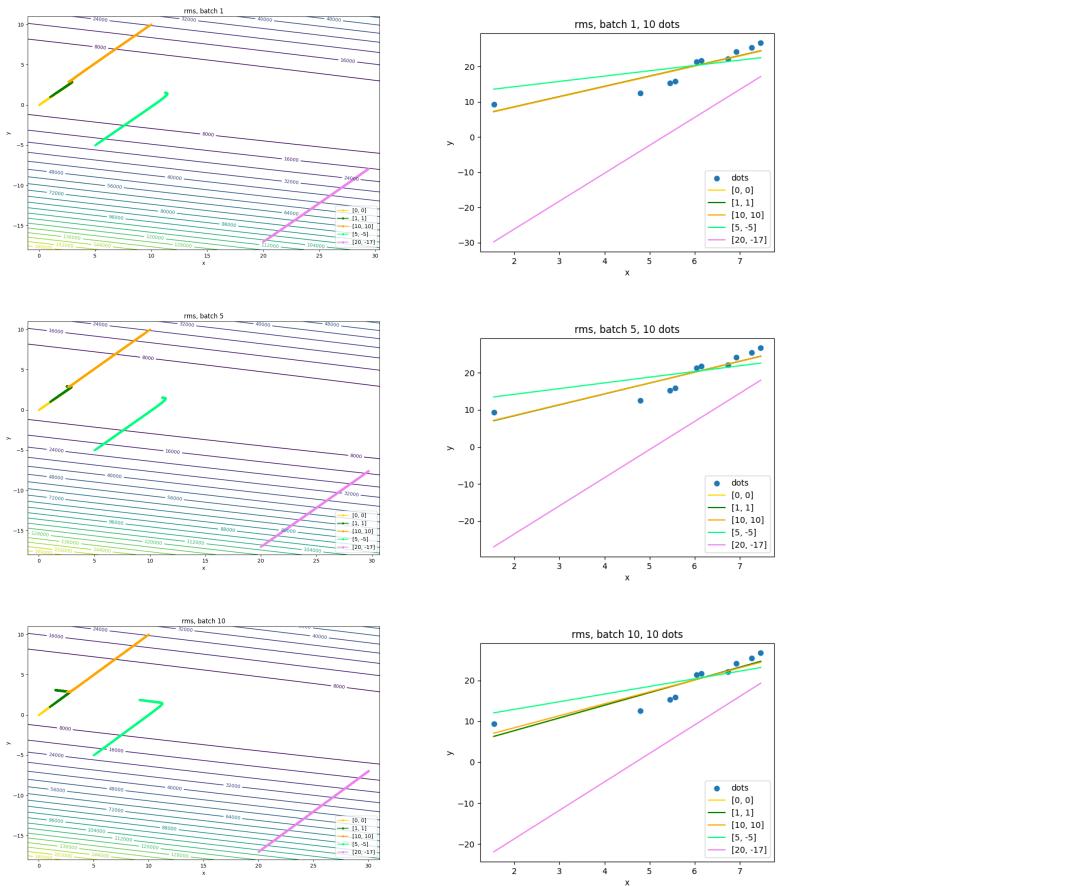




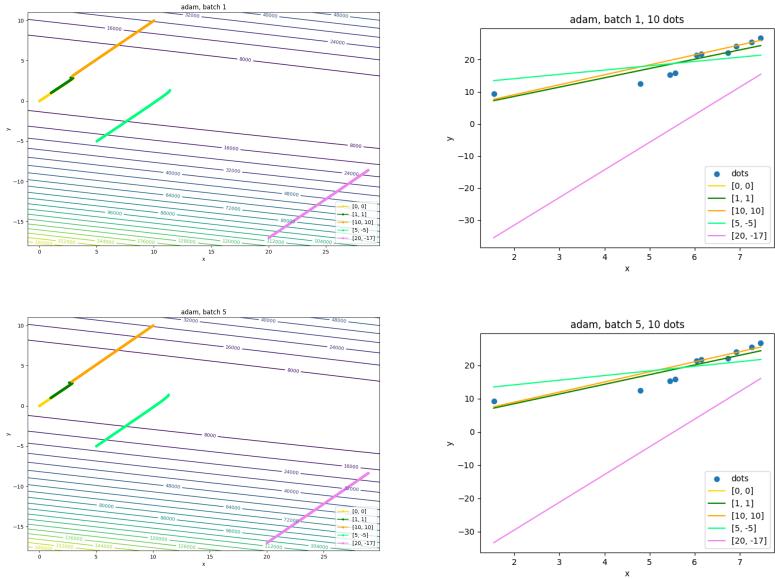
iii. AdaGrad

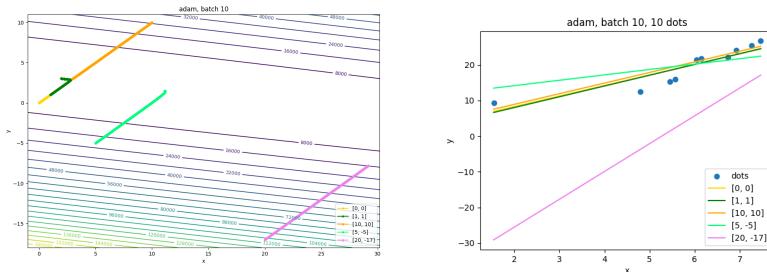


iv. RMSProp



V. Adam





batch 1												
algorithm	best result	worst result	avg result	min iters	max iters	avg iters	min memory	max memory	avg memory	min time	max time	avg time
nesterov	[1.55324 3.00135]	[12.06963 1.7034]	[0.55712 3.12754]	1000	1000	1000	3840	6592	4858	0.024	0.026	0.0248
momentum	[1.58198 2.99889]	[18.24899 0.95109]	[6.76185 2.36712]	1000	1000	1000	3776	4640	3949	0.024	0.025	0.0246
adagrad	[2.67759 2.88028]	[33.22381 -0.96822]	[9.79054 1.97599]	1000	1000	1000	3776	4640	3949	0.025	0.027	0.0258
rms	[2.35173 2.91471]	[29.11806 -8.7053]	[9.8612 0.31024]	1000	1000	1000	3776	4608	3942	0.028	0.029	0.0284
adam	[1.83424 1.746]	[20.98607 -16.17842]	[7.7201 -1.73308]	1000	1000	1000	3936	4768	4182	0.034	0.035	0.0344

batch 5												
algorithm	best result	worst result	avg result	min iters	max iters	avg iters	min memory	max memory	avg memory	min time	max time	avg time
nesterov	[2.33883 2.90863]	[6.57316 2.39802]	[1.96258 2.9547]	1000	1000	1000	3808	3808	3808	0.043	0.044	0.0436
momentum	[2.08617 2.93494]	[8.95063 2.11955]	[4.40464 2.65409]	1000	1000	1000	3776	3776	3776	0.043	0.046	0.044
adagrad	[2.48509 2.88767]	[33.10125 -0.96102]	[9.83933 1.9715]	1000	1000	1000	3776	3776	3776	0.044	0.045	0.0446
rms	[2.31111 2.91579]	[29.20847 -8.61928]	[9.84843 0.33127]	1000	1000	1000	3744	3776	3750	0.046	0.048	0.0464
adam	[1.83797 1.7502]	[20.91359 -16.17035]	[7.7217 -1.73034]	1000	1000	1000	3984	3936	3910	0.052	0.053	0.0526

batch 10												
algorithm	best result	worst result	avg result	min iters	max iters	avg iters	min memory	max memory	avg memory	min time	max time	avg time
nesterov	[1.97686 2.95038]	[4.12534 2.68463]	[2.60561 2.87261]	697	1000	939	3776	3776	3776	0.046	0.067	0.0622
momentum	[2.57495 2.8764]	[4.97487 2.57956]	[3.41758 2.77218]	653	1000	876	3744	3744	3744	0.043	0.065	0.056
adagrad	[2.8612 2.85168]	[30.10618 -0.57846]	[9.042 2.06725]	13	1000	429	3680	3744	3786	0.001	0.067	0.0286
rms	[2.84182 2.8351]	[30.06683 -6.91087]	[9.68805 0.72787]	261	1000	700	3712	3744	3718	0.018	0.069	0.048
adam	[1.86973 1.86956]	[20.9858 -16.01373]	[7.75432 -1.64549]	1000	1000	1000	3872	3984	3878	0.074	0.076	0.0744

По графикам траекторий можно сразу отметить, что в целом все алгоритмы ведут себя похожим образом вне зависимости от выбора начальных точек – их траектории движения очень схожи между собой. **Momentum** и **Nesterov** в целом справляются с задачей нахождения регрессии одинаково хорошо: если не смотреть на случай **batch = 1**, они пришли практически к идентичным результатам вне зависимости от выбора начальной точки. При большем размере батча уменьшается дисперсия результата функции в зависимости от выбора начальной точки. Такие модификации как **Nesterov** и **Momentum**, которые в своей работе используют «инерцию» стабильнее относительно взятой начальной точки, так как если мы начали вблизи минимума, то значение градиента будет маленьким и мы уйдем не далеко, а в противном случае градиент будет задавать больший импульс, в следствии чего мы быстрее приблизимся к нужным нам значениям. А вот если мы посмотрим на **AdaGrad**, то по понятным причинам начальная точка сильно влияет на результат работы алгоритма. Почему это так? Очевидно, что чем дальше мы от минимума, тем больше мы на начальных итерациях накопим градиентов, которые в свою очередь уменьшат **learning rate** до крохотных значений и мы просто застрянем на одном месте.

Для остальных алгоритмов – **AdaGrad**, **RMSProp** и **Adam**, можем наблюдать незначительные изменения результатов при разных батчах, однако дисперсия результатов относительно больше.

Если смотреть на показатели количества итераций и затраченного времени, то можно сделать вывод, что модификациям, которые используют адаптивный шаг (**RMSProp**, **AdaGrad**, **Adam**) необходимо меньшее кол-во итераций, и как результат они работают быстрее по времени.

Что касается количества арифметических операций, используемых алгоритмами, поскольку алгоритмы работают заданное количество итераций (в нашем случае количество эпох 1000), то в случае размеров **batch=1** и **batch=5** количество арифметических операций будет одинаково для одного и того же алгоритма на всех запусках. Приведем таблицу количества арифметических операций, используемых алгоритмами на 1000 итерациях при вычислении линейной функции:

nesterov	momentum	adagrad	rms	adam
19000	18000	21000	24000	42000

При импользовании `batch=10` некоторым алгоритмам в лучшем случае потребовалось меньшее количество итераций для сходимости. Число арифметических операций будет пропорционально количеству итераций, поскольку на каждой итерации количество операций для каждого алгоритма константно.

*. (a) Полиномиальная регрессия

По аналогии с линейной регрессии, мы предполагаем, что у нас есть некоторые данные $x = (x_1 \dots x_n)$, $y = (y_1 \dots y_n)$, такие, что y_i зависит от x_i ($y_i = p(x_i)$), где $p(x)$ - какой-то полином вида $w_0 + w_1x + w_2x^2 + \dots + w_kx^k$. Наша задача - найти эту зависимость.

Мы хотим найти приблизить некую полиномиальную функцию $\tilde{y} = p(x)$ к начальным данным, то есть найти такой вектор весов w , при котором достигается минимум функции

$$f(w) := \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (p(x_i) - y_i)^2$$

Как и в случае с линейной регрессией, найдем градиент функции $f(w)$:

$$\frac{df}{w_i} = \frac{2}{n} \sum_{j=0}^n x_i^j * (p(x_i) - y_j)$$

(b) Модификации полиномиальной регрессии добавлением регуляризации в модель

Говоря о регуляризации, чаще всего она используется для предотвращения переобучения модели. То есть, когда степень полинома функции, которую стремимся приблизить, становится слишком велика, что влечет очень хороший результат на обучающей выборке, но на практике не имеет смысла.

По сути, регуляризация - это добавление некоторого штрафа в минимизируемую функцию за большие значения коэффициентов у линейной модели. Тем самым запрещаются слишком "резкие" изгибы, и предотвращается переобучение.

Среди основных видов регуляризаций можно выделить следующие: L1, L2-регуляризации, а также **Elastic regularization**. Рассмотрим каждую из них подробнее.

В случае L1 регуляризации (также известной как **lasso regularization**) наша минимизируемая функция будет выглядеть следующим образом:

$$f(w) := \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2 + \lambda * \sum_{i=1}^n |w_i|$$

Данный вид регуляризации также позволяет ограничить значения вектора w . Однако, к тому же он обладает интересным и полезным на практике свойством — обнуляет значения некоторых параметров, что в случае с линейными моделями приводит к отбору признаков.

В случае L2-регуляризации (или **Ridge regularization**) минимизируемая функция будет выглядеть следующим образом:

$$f(w) := \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2 + \lambda * \sum_{i=1}^n w_i^2$$

Минимизация регуляризованного соответствующим образом эмпирического риска приводит к выбору такого вектора параметров w , которое не слишком сильно отклоняется от нуля. В линейных классификаторах это позволяет избежать проблем мультиколлинеарности и переобучения.

И третий вид регуляризации, который мы рассмотрим - **Elastic regularization**. В данном случае минимизируемая функция выглядит следующим образом:

$$f(w) := \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2 + \lambda_1 * \sum_{i=1}^n |w_i| + \lambda_2 * \sum_{i=1}^n w_i^2$$

Приведенная регуляризация использует как L1, так и L2-регуляризации, учитывая эффективность обоих методов. Ее полезной особенностью является то, что она создает условия для группового эффекта при

высокой корреляции переменных, а не обнуляет некоторые из них, как в случае с L1-регуляризацией.

Реализация (с использованием методов стохастического градиентного спуска на случай если захотите много точек и модификации Adam, так как без нее работает объективно плохо, совсем без регуляризации шагов «особенно стохастический градиент» очень часто перепрыгивает минимум функции):

```

l1_reg = lambda w, L: L * np.sum(np.abs(w))
l2_reg = lambda w, L: L * np.sum(w ** 2)
elastic_reg = lambda w, l1, l2: l1_reg(w, l1) + l2_reg(w, l2)

def stochastic_gd(w: np.array, x: np.array, y: np.array,
                  l1, regularization=None, batch=100,
                  lr=0.5, gamma_m=0.9, gamma_v=0.999, epsilon=1e-8, brk=1e-3, epoch=100):
    dim = x.size
    i, step, g_size, history, iters = 0, 1, ceil(dim / batch), [w], 0
    steps_history = np.array([np.ones(w.shape) for _ in range(g_size)])
    m = np.zeros(w.shape)
    v = np.zeros(w.shape)

    def get_dgi(u):
        indexes = [(u * batch + j) % dim for j in range(batch)]

        def dgi(ww):
            return np.array([
                (2.0 / dim)
                * sum([
                    (sum([w[q] * x[index] ** q for q in range(len(ww))]) - y[index])
                    * (x[index] ** der) for index in indexes
                ])
                + (0 if regularization is None else regularization(w, *l1) / dim)
                for der in range(len(ww))
            ])

        return dgi

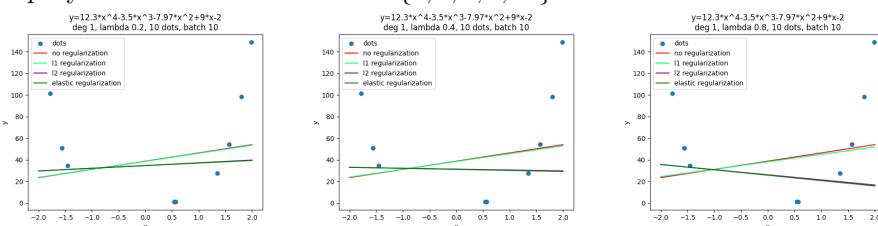
    dg = [get_dgi(k) for k in range(g_size)]
    random.shuffle(dg)
    for ep in range(1, epoch + 1):
        i = (i + 1) % g_size
        grad = dg[i](w)
        m = gamma_m * m + (1 - gamma_m) * grad
        v = gamma_v * v + (1 - gamma_v) * grad ** 2
        mm = m / (1 - gamma_m ** ep)
        vv = v / (1 - gamma_v ** ep)
        step = lr * mm / (np.sqrt(vv) + epsilon)
        steps_history[i] = np.abs(step)
        if np.sum(steps_history) / g_size < brk:
            break
        w = w - step
        iters += 1
        history.append(w)
    return np.array(history), iters

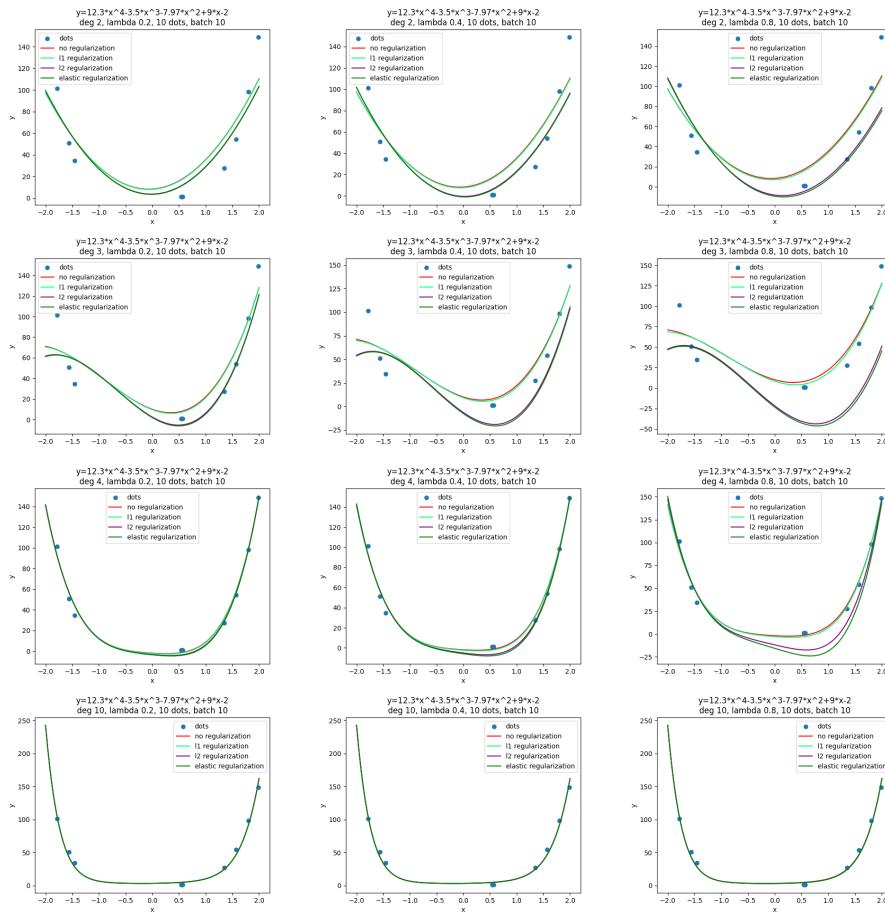
```

(c) Влияние регуляризации на восстановление регрессии

Рассмотрим такой случай: мы выбрали 10 точек, принадлежащих некоторой уже существующей функции, в данном случае 4 степени.

Ниже представлены графики работы вышеприведенного алгоритма для `batch=10`, (максимальный, соответствует обычному градиентному спуску – точек мало, смысла делить на минибатчи особого нет, это результат только ухудшит), для разных параметров $\lambda \in \{0.2\}$ (первый столбец), 0.4 (второй), 0.8 (третий) и требуемой степени полинома $\in \{1, 2, 3, 4, 10\}$





В целом, только глядя на графики, можно отметить, что $\lambda = 0.2$ проявляет себя лучше всего на данном наборе данных. При больших λ лучше заметно непосредственно влияние регуляризации. Но смотреть только на графики не особо надежно, тут во многих случаях неочевидно, какая регуляризация приблизилась лучше всего, давайте посмотрим на значение функции потерь в полученной в каждом алгоритме точке, а также величину суммы модулей коэффициентов.

[функция потерь, сумма модулей коэффициентов, мин по модулю коэффициентов]						
degree	lambda	no reg	L1	L2	elastic	
1	0.2	[26883.206, 37.577, 10.312]	[26899.816, 37.332, 10.122]	[27436.356, 33.473, 7.284]	[27474.658, 33.267, 7.129]	
1	0.4	[26883.206, 37.577, 10.312]	[26918.078, 37.089, 9.934]	[28255.871, 30.682, 4.884]	[28354.965, 29.73, 4.627]	
1	0.8	[26883.206, 37.577, 10.312]	[26959.363, 36.689, 9.564]	[30189.303, 24.696, 1.292]	[30419.932, 24.152, 0.916]	
2	0.2	[5961.327, 39.563, 3.143]	[5940.029, 36.779, 2.945]	[5938.97, 33.094, 0.504]	[5950.617, 32.178, 0.351]	
2	0.4	[5961.327, 39.563, 3.143]	[5921.567, 36.4, 2.75]	[6297.322, 31.86, 1.55]	[6357.355, 31.22, 1.801]	
2	0.8	[5961.327, 39.563, 3.143]	[5892.641, 35.657, 2.372]	[7585.894, 31.157, 4.719]	[7807.738, 30.459, 5.143]	
3	0.2	[4272.654, 53.457, 5.557]	[4226.095, 51.206, 5.631]	[3918.953, 53.085, 6.243]	[3906.576, 52.537, 6.299]	
3	0.4	[4272.654, 53.457, 5.557]	[4181.88, 51.347, 5.784]	[4011.984, 52.888, 6.693]	[4049.221, 52.248, 6.774]	
3	0.8	[4272.654, 53.457, 5.557]	[4100.695, 51.603, 5.846]	[5199.092, 50.854, 5.566]	[5466.596, 49.894, 4.737]	
4	0.2	[424.84, 38.106, 0.931]	[415.564, 15.734, 0.736]	[386.831, 17.17, 0.891]	[382.519, 16.975, 0.283]	
4	0.4	[424.84, 38.106, 0.931]	[407.211, 15.761, 0.541]	[375.835, 19.229, 1.101]	[378.902, 19.432, 1.489]	
4	0.8	[424.84, 38.106, 0.931]	[393.247, 15.812, 0.153]	[449.06, 23.347, 2.046]	[520.991, 24.427, 2.181]	
10	0.2	[4088.162, 7.155, 0.027]	[4011.228, 7.133, 0.027]	[4011.213, 7.133, 0.027]	[4014.265, 7.109, 0.027]	[4020.316, 7.063, 0.027]
10	0.4	[4088.162, 7.155, 0.027]	[4014.289, 7.106, 0.027]	[4014.246, 7.111, 0.027]	[4020.258, 7.069, 0.027]	[4032.262, 6.973, 0.026]
10	0.8	[4088.162, 7.155, 0.027]	[4020.394, 7.057, 0.027]	[4020.258, 7.069, 0.027]	[4032.262, 6.973, 0.026]	

Здесь видно, что на степенях $=2, 3, 4$ L1-регуляризация, как и должна была, работает на то, чтобы коэффициенты результата алгоритма – вектора w были в среднем меньше по модулю, а также (на больших степенях это заметно сильнее) пытается приблизить к нулю некоторые из параметров. L2-регуляризация, пытается не сильно отклонять коэффициенты от нуля, то есть уменьшает их значения с средним, Elastic же работает как комбинация вышеупомянутых регуляризаций. На степени $degree=10$ мы, как и следовало ожидать, получили переобученную модель, можно сказать, что на всех регуляризациях и параметрах нам выдали коэффициенты, соответствующие функции, едва ли вообще отклоняющейся от какой либо из точек, однако для предположения результата на других точках работать она будет плохо.