**Градиентный Спуск Такое название не пойдёт**

***Камкин Илья Дмитриевич***

*студент, Липецкий государственный технический университет,*

*Россия, г. Липецк*

***Седых Ирина Александровна***

*научный руководитель, доктор тех. Наук,*

*профессор кафедры высшей математики*

*Липецкий государственный технический университет,*

*Россия, г. Липецк*

**аннотация**

Целью статьи является изучение градиентного спуска, а так же анализ зависимости параметров градиентного спуска на размер среднеквадратичной ошибки.

**Ключевые слова:** градиентный спуск, среднеквадратичная ошибка, машинное обучение.

Градиентный спуск — простой метод оптимизации, позволяющий найти минимум целевой функции. Это жадный алгоритм, который делает шаг в сторону максимальной скорости убывания функции. Рассмотрим функцию нескольких переменных f(w), где . Чтобы найти w, при котором функция достигает своего локального минимума, градиентный спуск делает следующие шаги:

1. Выбирается случайное значение w.

2. Выбирается максимальное количество итераций T.

3. Выбирается значение скорости обучения (шаг сходимости)

4. Повторять следующие 2 шага пока f не изменится, или число итераций не превысит T.

a. Вычислить:

b. Перезаписать w:

Здесь

Рассмотрим пример с функцией параболоида в качестве функции нескольких переменных. На каждой итерации обновляются так:

Градиент можно рассматривать как стрелку, указывающую направление, в котором функция возрастает быстрее всего. Следование отрицательному направлению градиента приведёт к точкам, в которых значение функции уменьшается с максимальной скоростью. Скорость обучения, также называемая размером шага, определяет, насколько быстро мы движемся в направлении от градиента.

При использовании градиентного спуска мы сталкиваемся со следующими проблемами:

1. Попадание в ловушку локального минимума, что является прямым следствием жадности этого алгоритма.
2. Превышение и отсутствие глобального оптимума — это прямой результат слишком быстрого движения по направлению градиента.
3. Осцилляция - это явление, которое возникает, когда значение функции не изменяется существенно независимо от направления, в котором оно движется (так называемое плато).

Чтобы решить эти проблемы, к выражению добавляется момент α. Ниже формула для определённой итерации:

**Реализация Градиентного спуска на python**

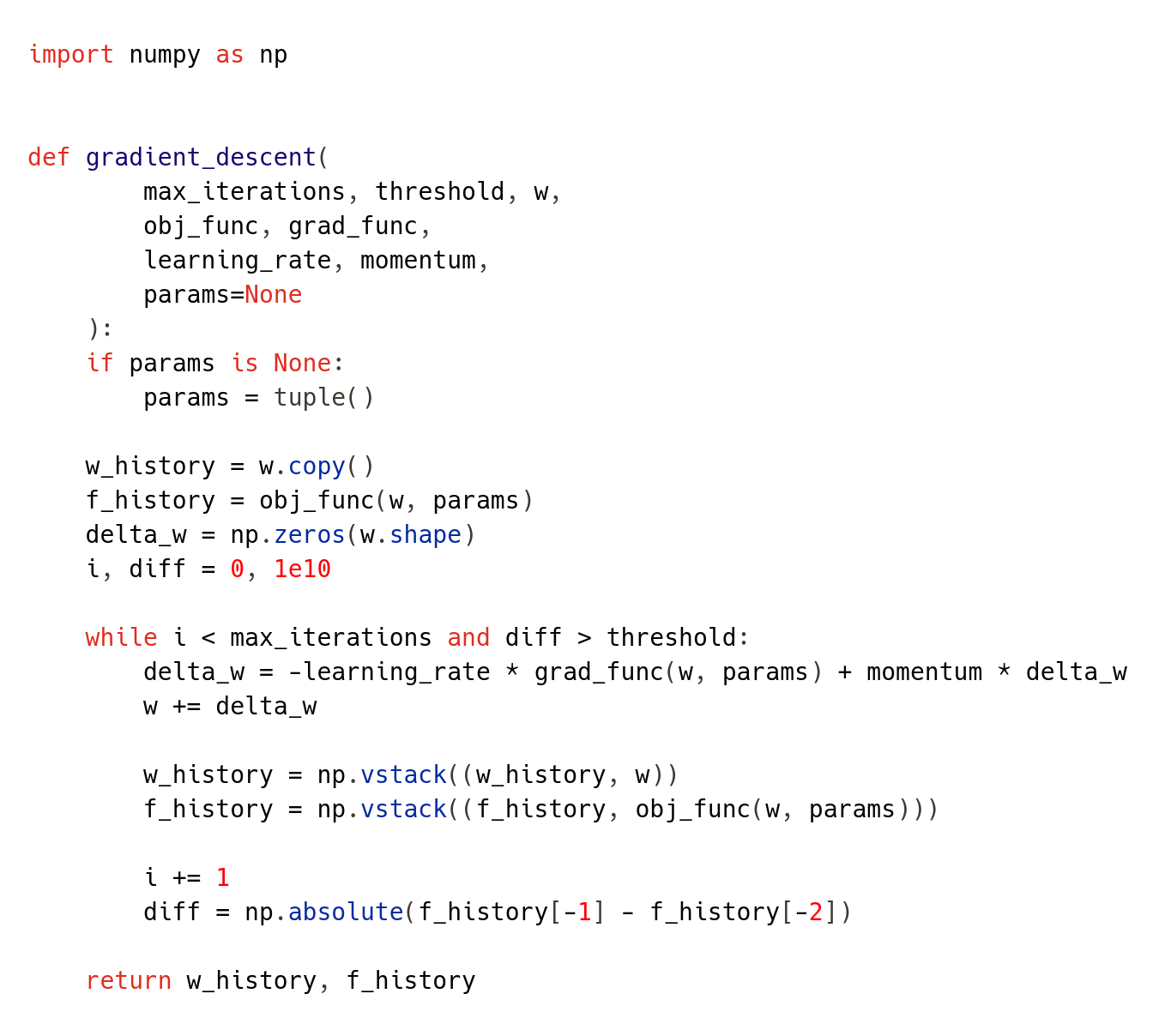
Для математических вычислений используется библиотека numpy.

Для графического представления используется библиотека matplotlib.

Реализуем функцию градиентного спуска, которая будет принимать функцию, её градиент и возвращать посещённые точки со значениями в этих точках. (см. рис. 1)

Функция будет принимать следующие параметры:

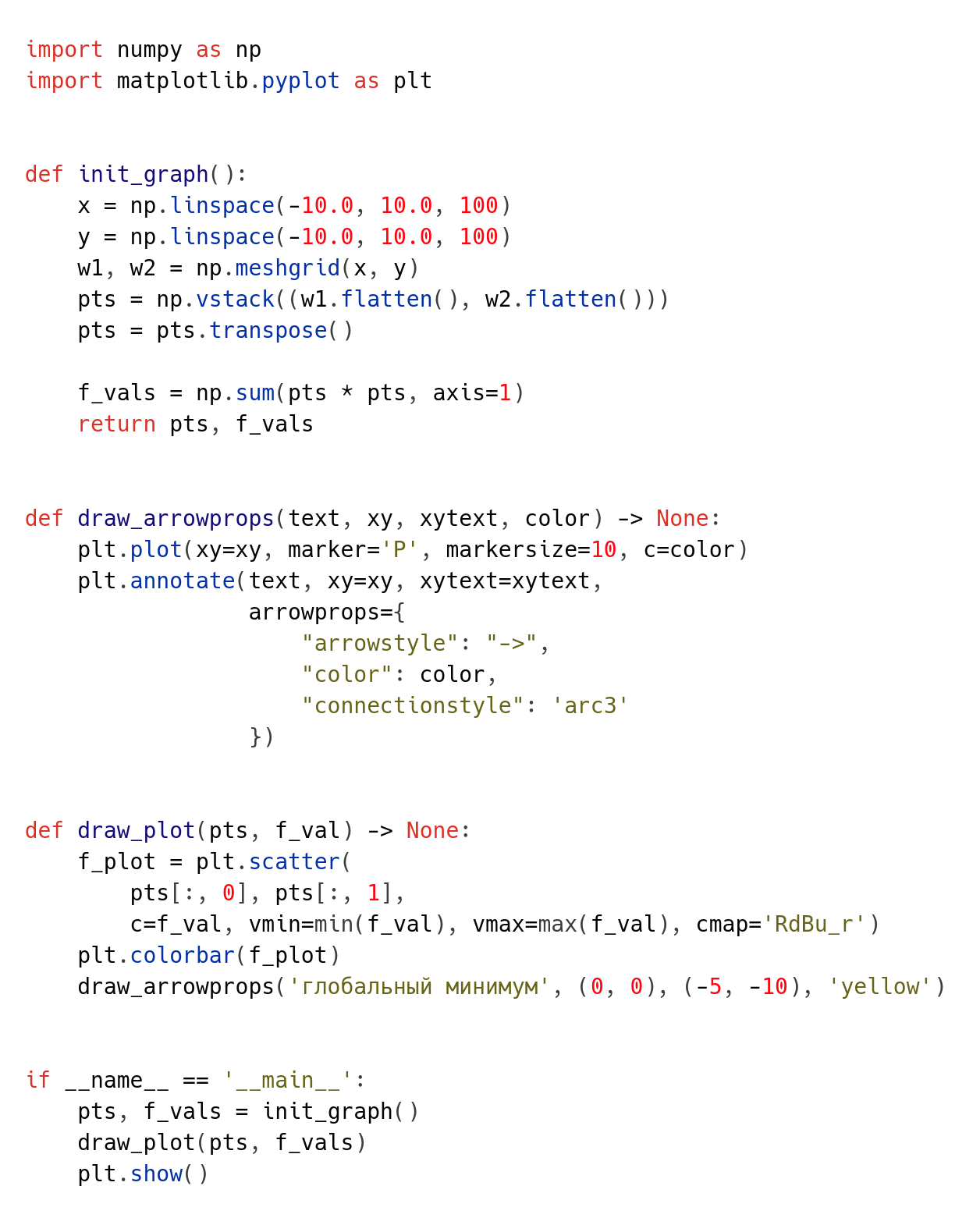
* max\_iterations — максимальное количество итераций.
* threshold — если разница между значениями функции падает ниже этого значения остановить вычисление.
* w - точка, с которой начинается вычисление градиентного спуска.
* obj\_func — функция, которая вычисляет значение.
* grad\_func — функция, которое вычисляет градиент.
* learning\_rate — значение скорости обучения (шаг сходимости).
* momentum — значение момента.
* params — дополнительные параметры для функций вычисляющих градиент и значению (не является необходимым).

Рисунок 1. Функция градиентного спуска

Функция вернёт:

* w\_history — все точки, которые были посещены во время градиентного спуска
* f\_history — значения для всех точек, которые были посещены во время градиентного спуска

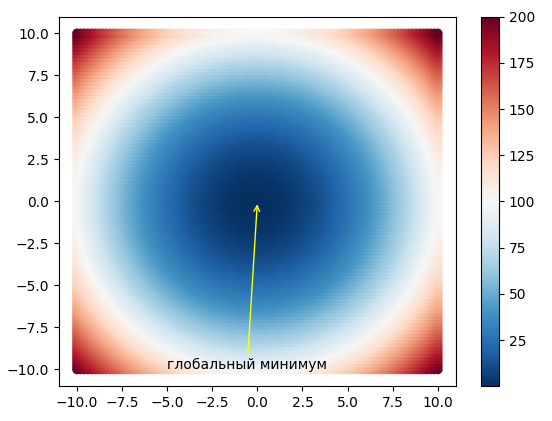
Рассмотрим пример с параболоидом, для начала изобразим график, с которым будем в дальнейшем работать (см. рис.2).

Рисунок 2. Отображение графика параболоида

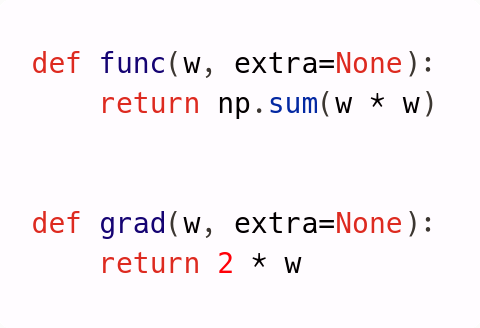
Функция init\_graph генерирует поверхность с 10.000 точками, распределёнными по поверхности.

Функция draw\_arrowprops отображает окрашенную стрелку.

Функция draw\_plot отображает график параболоида, где цвет — значение функции в определённой точке.

Рисунок 3. График параболоида

Далее определим нашу функцию параболоида, а так же её градиент. (см. рис. 4)

Рисунок 4. Функция параболоида и её градиент

Прежде, чем перейти к визуализации зависимости найденного локального минимума от значений изначальных параметров градиентного спуска рассмотрим ещё один метод поиска локального минимума — метод Ньютона.

**метода ньютона**

Прежде, чем перейдём к визуализации зависимости найденного локального минимума от значений изначальных параметров градиентного спуска рассмотрим ещё один метод поиска локального минимума — метод Ньютона. Сам метод Ньютона ищет такой *x,* что *f(x)=0* это не совсем то, что нужно, ведь функция не обязательно имеет экстремум в нуле. Поэтому применяется метод Ньютона для оптимизации [1]. Это два разных метода, для последнего необходимо считать вторые производные (матрицу Гессе). Далее под методом Ньютона будет подразумеваться именно метод Ньютона для оптимизации.

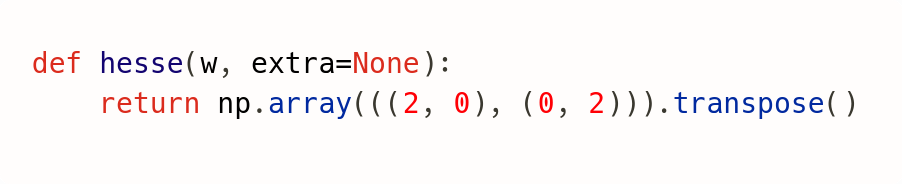
Метод Ньютона итеративный алгоритм, в одномерном случае выглядит следующим образом:

В многомерном случае первая производная заменяется на градиент, а вторая — на матрицу Гессе. Делить матрицы нельзя, поэтому происходит умножение на обратную матрицу.

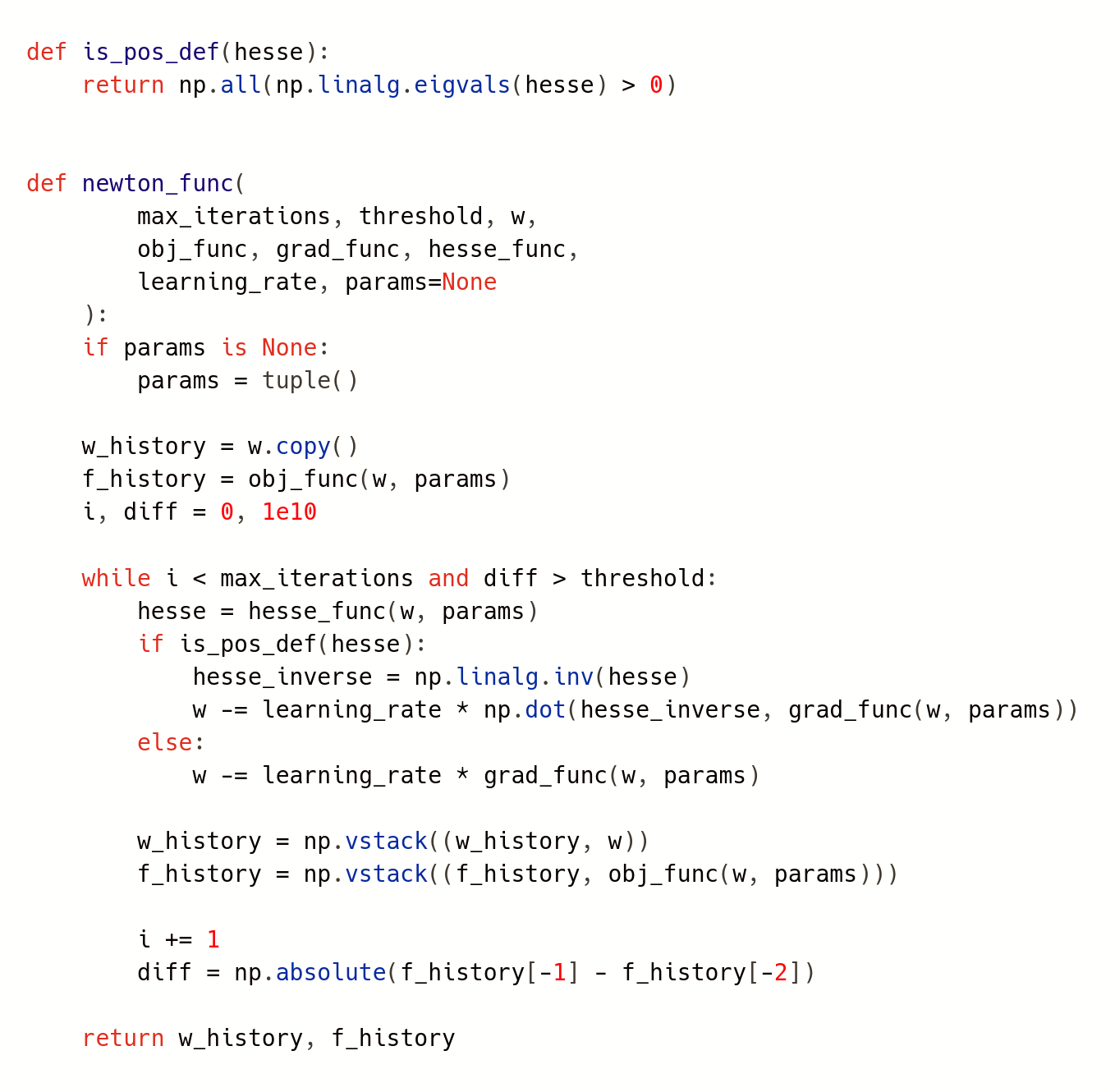
Так же нам необходима проверка в том ли мы направлении движемся. Поэтому будем проверять матрицу Гессе, если она положительно определена [2] — то мы движемся в правильном направлении, если нет — то будем использовать градиент для продвижения далее.

**Реализация Метода ньютона на python**

Для ранее рассмотренной функции параболоида напишем матрицу Гессе. (см рис. 5)

Рисунок 5. Матрица Гессе для функции параболоида

После напишем сам метода Ньютона. (см. рис. 6)

Рисунок 6. Метод Ньютона

is\_pos\_def – функция проверки матрицы Гессе.

newton\_func отличается от функции градиентного спуска тем, что принимается дополнительный параметр:

* hesse\_func – функция матрицы Гессе

Возвращаемые значения такие же, как и у градиентного спуска.

def draw\_one\_graph(w\_history, pts, f\_vals):

draw\_plot(pts, f\_vals)

plt.plot(w\_history[:, 0], w\_history[:, 1], marker='o', c='magenta')

draw\_arrowprops('минимум', w\_history[-1], (-1, 7), 'green')

for i, w in enumerate(w\_history[:-1]):

plt.annotate(

"",

xy=w, xycoords='data', xytext=w\_history[i+1, :], textcoords='data',

arrowprops={

"arrowstyle": '<-',

"connectionstyle": 'angle3'

})

def solve\_fw():

rand = np.random.RandomState(19)

w = rand.uniform(-10, 10, 2)

fig, \_ = plt.subplots(nrows=4, ncols=4, figsize=(54, 54))

learning\_rates = [.05, .3, .7, .9]

momentum = [0, .2, .7]

ind = 1

pts, f\_vals = init\_graph()

for alpha in momentum:

for col, rate in enumerate(learning\_rates):

plt.subplot(3, 4, ind)

w\_history, \_ = gradient\_descent(

max\_iterations=10,

threshold=-1,

w=w.copy(),

obj\_func=func,

grad\_func=grad,

learning\_rate=rate,

momentum=alpha

)

draw\_one\_graph(w\_history, pts, f\_vals)

ind += 1

plt.text(-9, 12, f'Скорость = {rate}', fontsize=13)

if col == 1:

plt.text(10, 15, f'Импульс = {alpha}', fontsize=20)

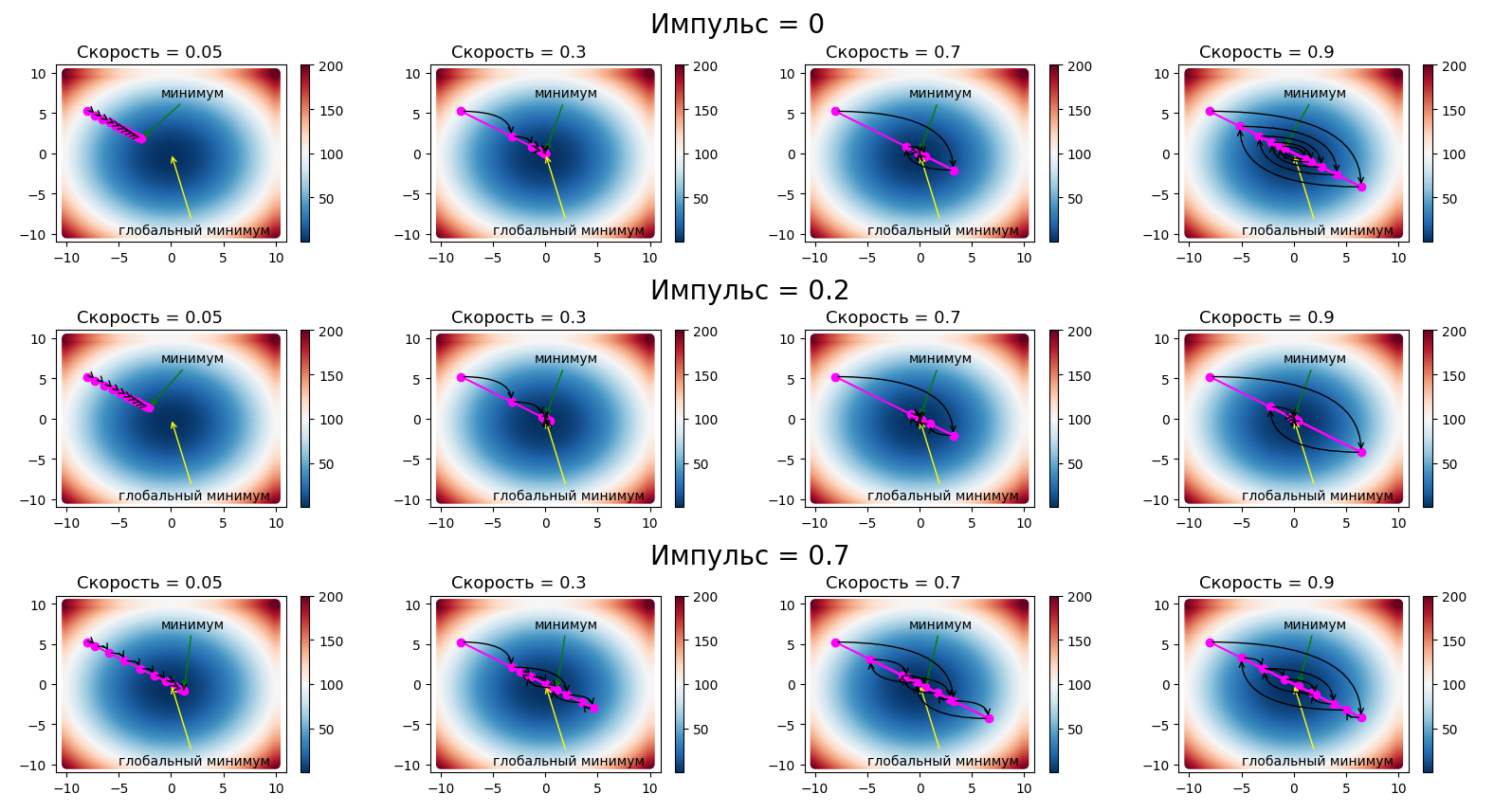
fig.subplots\_adjust(hspace=.5, wspace=.3)

plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

solve\_fw()

Запустим данный код и увидим следующее:



Рассмотрим график с импульсом = 0, и скоростью обучения = 0.05, видно, что алгоритм не достиг глобального минимума, увеличивая импульс (первый столбец) видно, что ситуация улучшается.

Так же рассмотрим график с импульсом = 0.7, и скоростью обучения = 0.9, видно, что алгоритм совершает колебания, это не является оптимальным вариантом.

Оптимально же использовать небольшое значение скорости и среднее значение импульса.

**Градиентный спуск для минимизации среднеквадратичной ошибки**

Градиентный спуск — это хороший и простой метод минимизации среднеквадратичной ошибки в контролируемой классификации или задаче регрессии.

Предположим, что нам дано m обучающих примеров , где каждый пример имеет n признаков. Если соответствующие целевое и выходное значения для каждого примера равны и соответственно, то функция среднеквадратичной ошибки определяется как:

Где выход определяется взвешенной линейной комбинацией входов, определяемой как:

Неизвестным параметром в приведенном выше уравнении является весовой вектор

Целевая функция в этом случае представляет собой среднеквадратичную ошибку с градиентом, определяемым выражением:

Где – i-й пример, или массив признаков размера n .

Функция и её градиент в коде будут выглядеть так:

def grad\_mse(w, xy):

x, y = xy

rows, cols = x.shape

o = np.sum(x\*w, axis=1)

diff = y-o

diff = np.tile(diff.reshape((rows, 1)), (1, cols))

grad = -np.sum(diff\*x, axis=0)

return grad

def mse(w, xy):

x, y = xy

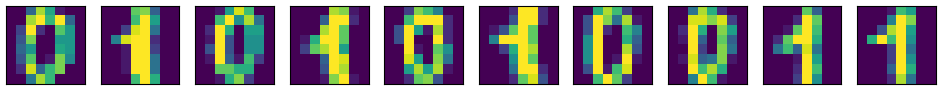
o = np.sum(x\*w, axis=1)

mse = np.sum((y-o)\*\*2)

return mse/2

Чтобы проиллюстрировать градиентный спуск в задаче классификации, воспользуемся набором рукописных цифр включённых в sklearn.datasets.

Чтобы не усложнять задачу используем блок с 10 цифрами нулей и единиц.



digits, target = dt.load\_digits(n\_class=2, return\_X\_y=True)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(

digits, target, test\_size=.2, random\_state=10

)

x\_train = np.hstack((np.ones((len(y\_train), 1)), x\_train))

x\_test = np.hstack((np.ones((len(y\_test), 1)), x\_test))

rand = np.random.RandomState(19)

w = rand.uniform(-1, 1, x\_train.shape[1]) \* 1e-6

if flag\_show\_data:

fig, ax = plt.subplots(

nrows=1, ncols=10, figsize=(12, 4),

subplot\_kw={"xticks": [], "yticks": [] }

)

for i in np.arange(10):

ax[i].imshow(digits[i, :].reshape(8, 8))

plt.show()

Необходимо разделить данные на обучающие и тестовые. В приведенном ниже коде выполняется градиентный спуск на тренировочном наборе и строится среднеквадратичная ошибка с разными параметрами градиентного спуска.

Так же напишем функцию для вычисления частоты ошибок классификации.

def error(w, xy):

x, y = xy

o = np.sum(x \* w, axis=1)

ind\_0, ind\_1 = np.where(o <= .5), np.where(o > .5)

o[ind\_0] = 0

o[ind\_1] = 1

return np.sum((o - y) \* (o - y)) / y.size \* 100

Шаг выбран тестовым путём и равен импульс изменяется по следующему набору: [0, 0.7, 0.9].

fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(12, 4))

for ind, alpha in enumerate((0, .7, .9)):

w\_history, y\_history = gradient\_descent(

max\_iterations=100,

threshold=1e-4,

w=w,

obj\_func=mse,

grad\_func=grad\_mse,

learning\_rate=1e-6,

momentum=alpha,

params=(x\_train, y\_train)

)

plt.subplot(131 + ind)

plt.plot(np.arange(y\_history.size), y\_history, color='green')

if ind == 1:

plt.xlabel('Итерация')

if ind == 0:

plt.ylabel('Среднеквадратичная ошибка')

plt.title(f'Импульс = {alpha}\n')

train\_error = error(w\_history[-1], (x\_train, y\_train))

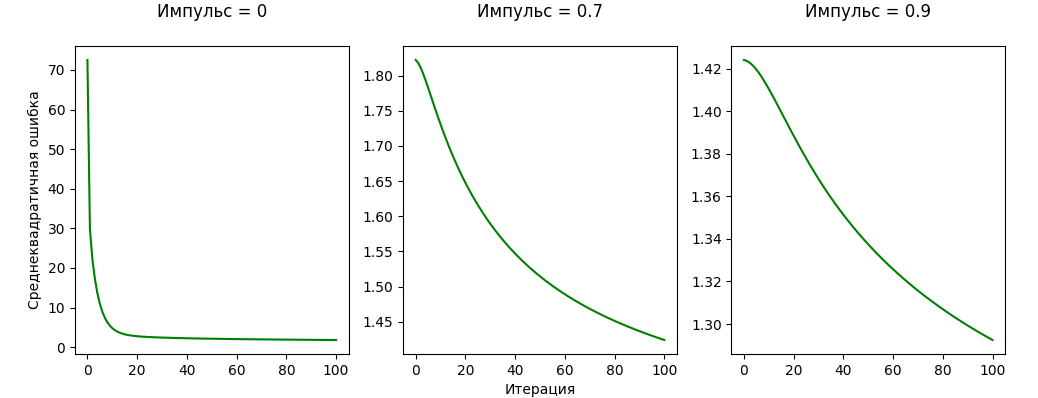
test\_error = error(w\_history[-1], (x\_test, y\_test))

print(f"Ошибка по отношению к обучающим данным: {train\_error}")

print(f"Ошибка по отношению к тестовым данным: {test\_error}\n")

plt.show()

Запустим код и увидим:

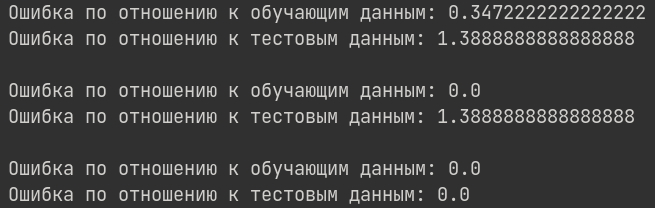


При 0 импульсе на 100 итерации достигается значение среднеквадратичной ошибки в 1.8.

При импульсе 0.7 достигается значение ошибки в 1.45.

При импульсе в 0.9 достигается значение ошибки в 1.30.

Однако, стоит рассмотреть вычисленную ошибку по отношению к тестовым и обучающим данным. Видно, что при импульсе 0.9 ошибка крайне мала. Она не равна фактическому нулю, но является маленькой.



В ходе анализа выяснилось, что незначительное изменение параметров градиентного спуска значительно влияет на результат. Нет какого-то универсального значения, и оно зависит от функции. Однако, зачастую маленькое значение шага и среднее значение импульса даёт хороший результат, что и было подтверждено в ходе исследования на функции окружности, а так же в минимизации среднеквадратичной ошибки в задаче классификации.

**Список литературы:**

1. https://en.wikipedia.org/wiki/Newton's\_method\_in\_optimization
2. https://ru.wikipedia.org/wiki/Положительно\_определённая\_матрица