Отчёт по лабораторной работе №4
по дисциплине «Методы Оптимизации»

Выполнили студенты группы M32341

Ивченков Дмитрий

Султанов Мирзомансурхон

Трещёв Артём

Санкт-Петербург, 2023

Использование вариантов SGD (torch.optim) из PyTorch для реализации SGD, Momentum, Nesterov, Adam, RMSProp, AdaGrad

Всё, что мы делали до этого во второй лабе, можно сделать проще и более эффективно, используя библиотеку torch.optim, так как её писали умные люди и реализации оптимизированы лучше чем то, что писали мы, но вот вопрос в том, насколько лучше. Перейдём сначала к реализации.

Для начала поприветствуем рукописный батч, так как в torch.optim, к сожалению, нет поддержки батчей, приходится явно указывать, какие точки мы хотим учитывать для вычисления градиента на следующем шаге, для этого написана функция mini_batch, которая из набора всех точек отбирает batch_size точек для дальнейшей работы.

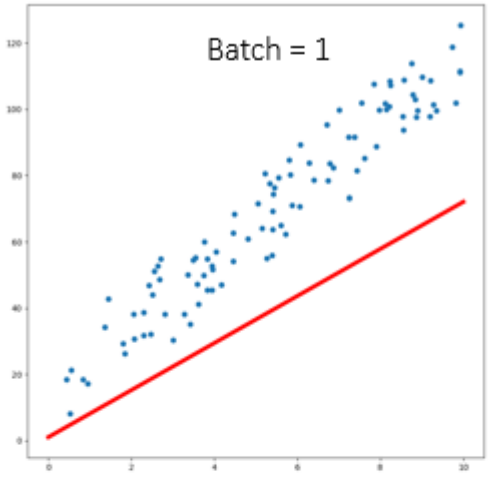
```
def get_mini_batch(X, Y, i, batch_size):
    X_mini = []
    Y_mini = []
    for j in range(batch_size):
        X_mini.append(X[(i * batch_size + j) % X.size()[0]].item())
        Y_mini.append(Y[(i * batch_size + j) % Y.size()[0]].item())
    return array_to_torch(X_mini), array_to_torch(Y_mini)
```

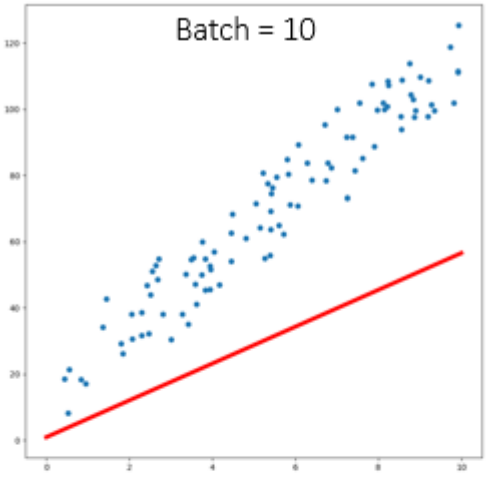
А теперь собственно саму реализацию градиентного спуска через библиотеку PyTorch. Легко заметить, что единственная отличающаяся часть каждой реализации — это то, какой оптимизатор выбираем, а затем уже моделируем ситуацию на батче, находим градиент согласно MSELoss и делаем шаг оптимизации.

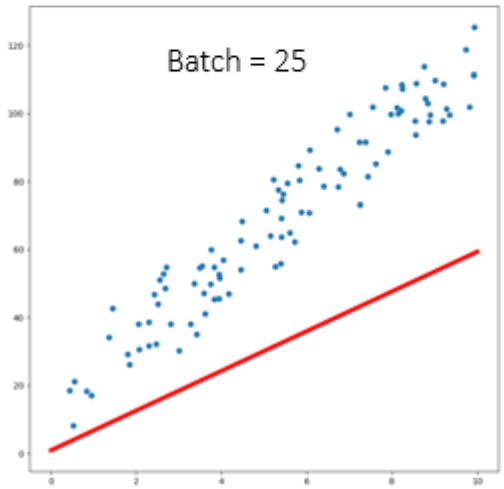
```
def SGD_PT(X, Y, lr, mode='SGD', epochs=100, log=False, initial_guess=None, batch_size = 1):
    # Модель линейной регрессии
    # Если взять, что у нас Y = wX + b, то w = weight в нашей модели, а b - bias
    if initial_guess is None:
        initial_guess = [0, 0]
    points = np.zeros((epochs + 1, 2))
    model = nn.Linear(1, 1)
    model.weight.data = torch.tensor([[float(initial_guess[0])]])
    model.bias.data = torch.tensor([[float(initial_guess[1])]])
    # Функция потерь
    criterion = nn.MSELoss()
    optimizer = ''
    # Оптимизатор и скорость обучения
    # Ordinary SGD
    if mode == 'SGD':
        optimizer = optim.SGD(model.parameters(), lr=lr)
    # SGD with momentum
    elif mode == 'Momentum':
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    # SGD Nesterov
    elif mode == 'Nesterov':
        optimizer = optim.SGD(model.parameters(), lr=lr, nesterov=True, momentum=0.9)
    # AdaGrad
    elif mode == 'AdaGrad':
        optimizer = optim.Adagrad(model.parameters(), lr=lr)
    # RMSProp
    elif mode == 'RMSProp':
        optimizer = optim.RMSprop(model.parameters(), lr=lr)
    # Adam
    elif mode == 'Adam':
        optimizer = optim.Adam(model.parameters(), lr=lr)
```

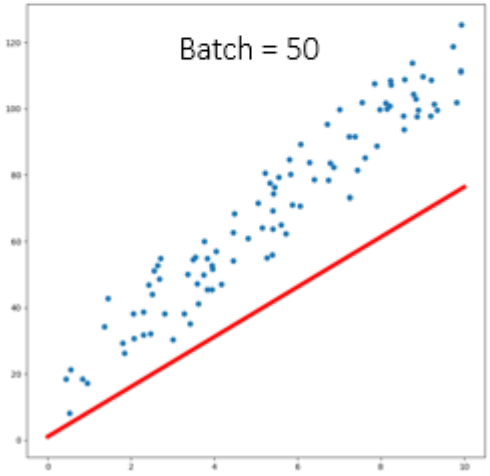
```
    num_epoch = epochs
    for epoch in range(epochs):
        # Подсчитываем ошибку модели

        X_mini, Y_mini = get_mini_batch(X, Y, np.random.randint(0, X.size()[0]), batch_size)
        outputs = model(X_mini)
        loss = criterion(outputs, Y_mini)

        # Обнуляем градиент, если мы не хотим накапливать градиент
        optimizer.zero_grad()
        # Обновляем градиент
        loss.backward()

        gradient_values = [param.grad.item() for param in model.parameters()]
        if abs(gradient_values[0]) < eps_point[0] and abs(gradient_values[1]) < eps_point[1]:
            print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch + 1, epochs, loss.item()))
            print('Predicted:', model.weight.item(), model.bias.item())
            num_epoch = epoch
            break
        # Делаем шаг градиентного спуска
        optimizer.step()
        points[epoch + 1] = np.array([model.weight.item(), model.bias.item()])
        if log:
            # Выводим промежуточные результаты
            if (epoch + 1) % 10 == 0:
                print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch + 1, epochs, loss.item()))
                print('Predicted:', model.weight.item(), model.bias.item())
    return points, num_epoch
```

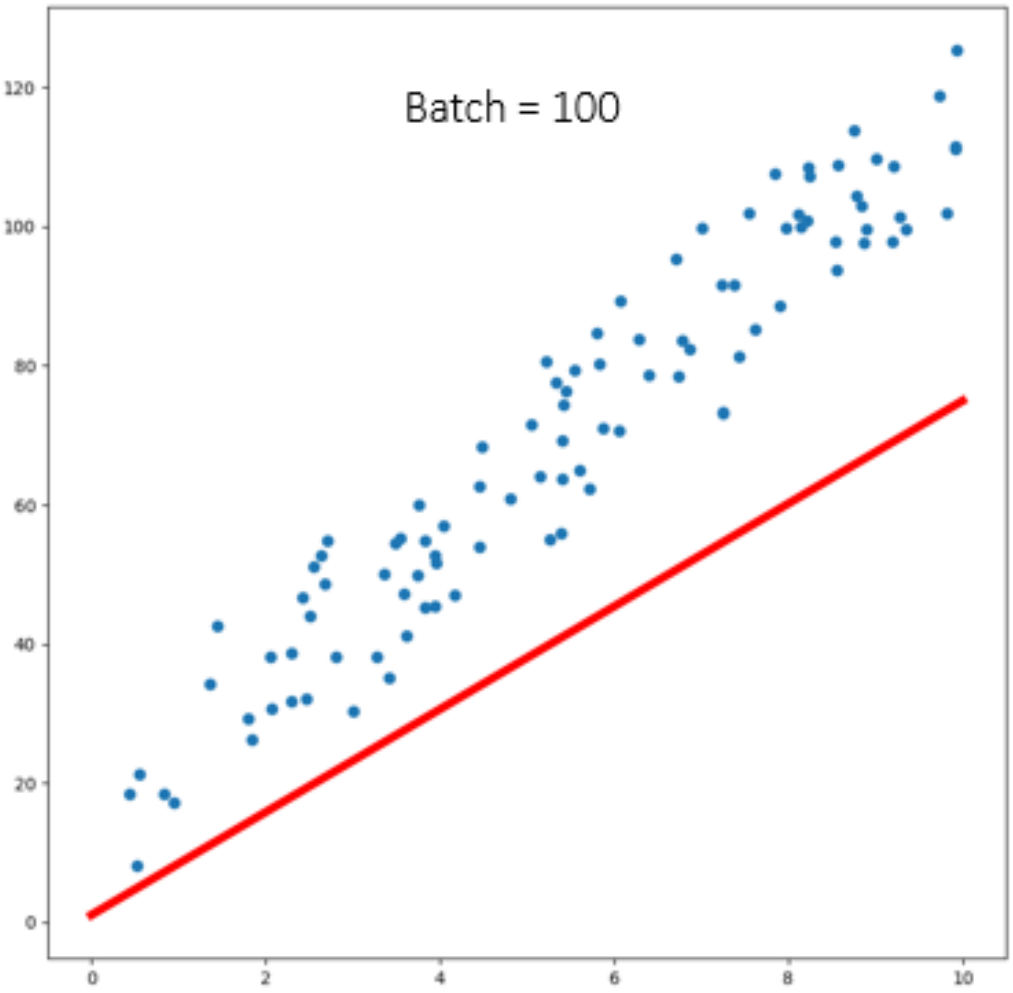
Как же оно работает на практике? Вполне прекрасно, результаты линейной регрессии можно наглядно увидеть на рисунках ниже. Для начала рассмотрим то, как хорошо справляется настройка батча на примере функции y = 10x с шумом равным 30 и 10 эпохами на 100 точках.


Batch = 1


Batch = 10


Batch = 25


Batch = 50

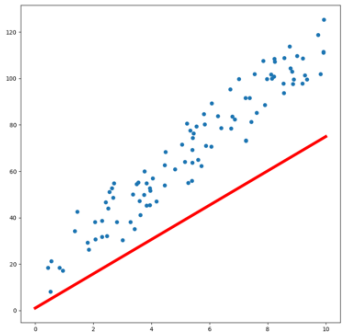


Как можно видеть, чем больше батч, тем точнее результат и тем быстрее мы добираемся до оптимального ответа за 10 эпох, однако, уже начиная с 25 разницы особой нет в результате, а результат с одним батчем тоже показывает неплохие результаты на данном примере, однако здесь нам просто повезло, ведь при работе алгоритм может всё

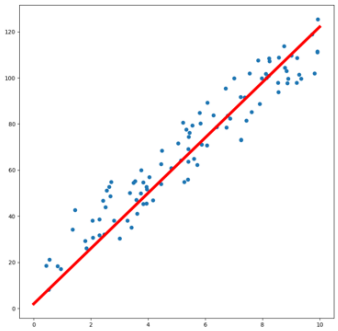
время выбирать случайно одни и те же точки и не учитывать все точки достоверно верно. Чем больше батч, тем меньше вероятность подобного явления.

Теперь посмотрим, как же у нас меняется результат в зависимости от эпох, на этот раз зафиксируем всё кроме эпох на примере функции y = 10x с шумом равным 30 и для примера полным батчем равным количеству точек на 100 точках.

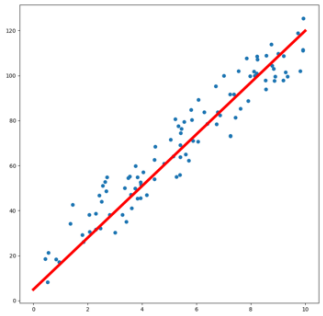
10 эпох



100 эпох

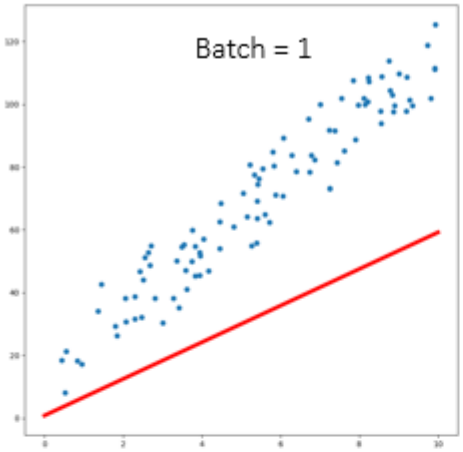


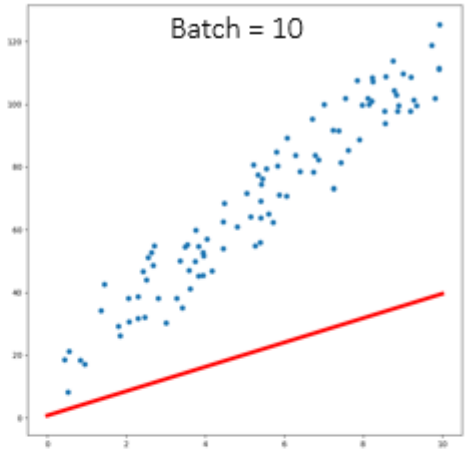
1000 эпох

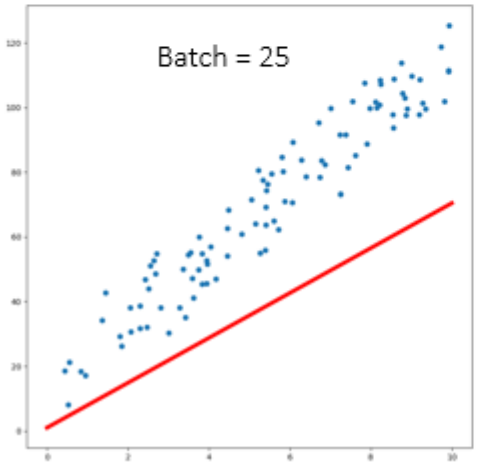


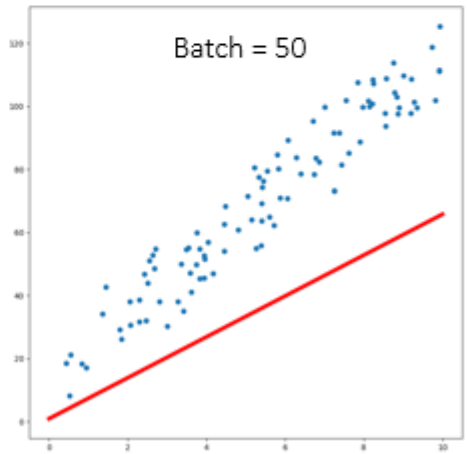
Так, ну хорошо, убедились, что всё работает. А что с другими модификациями? Так вот же они сверху вниз!

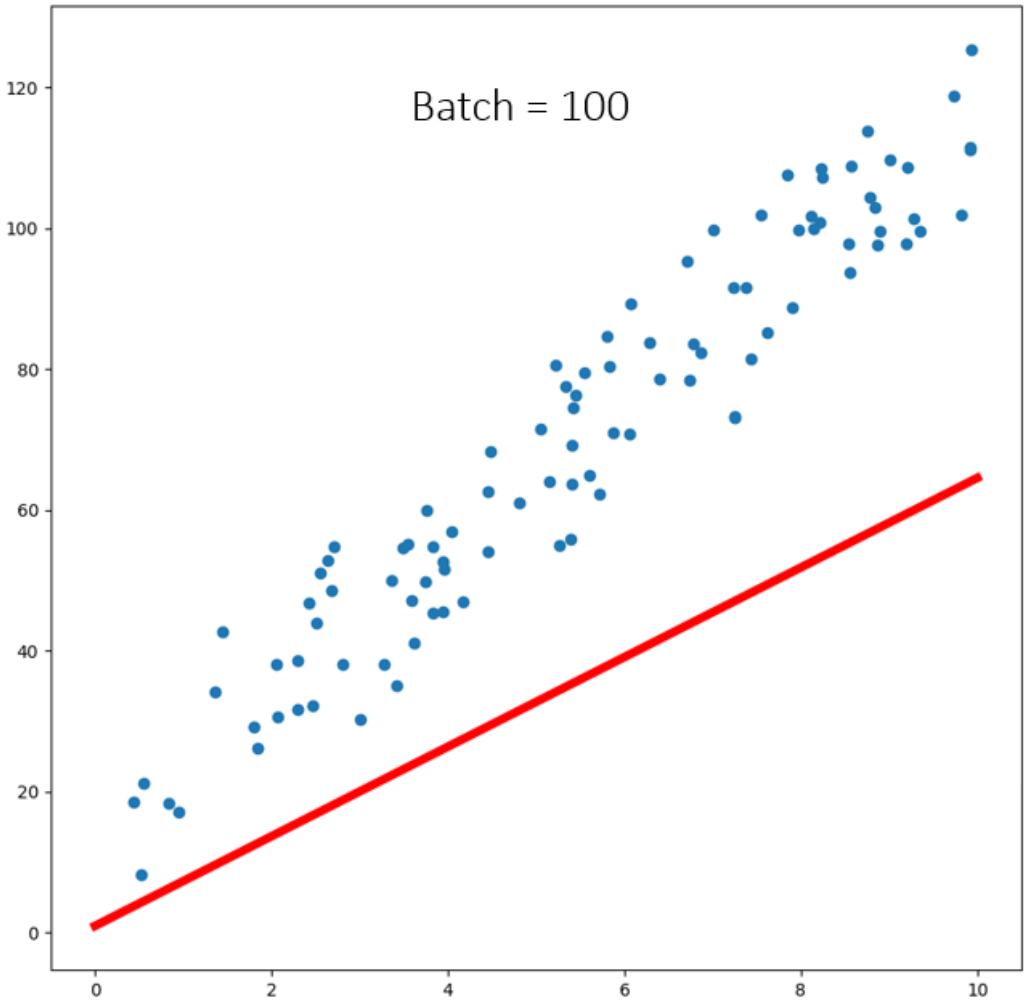
Momentum:


Batch = 1

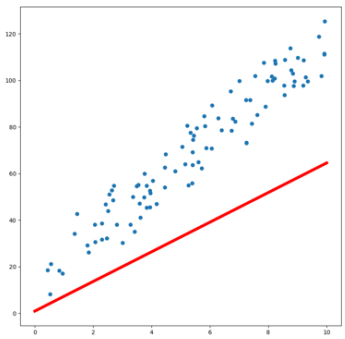

Batch = 10


Batch = 25

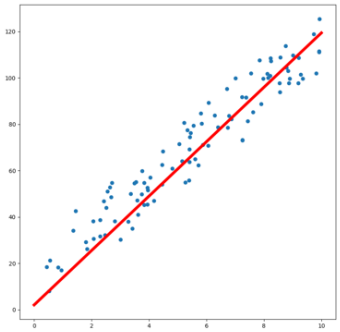

Batch = 50



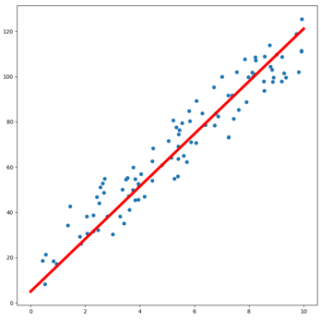
10 эпох



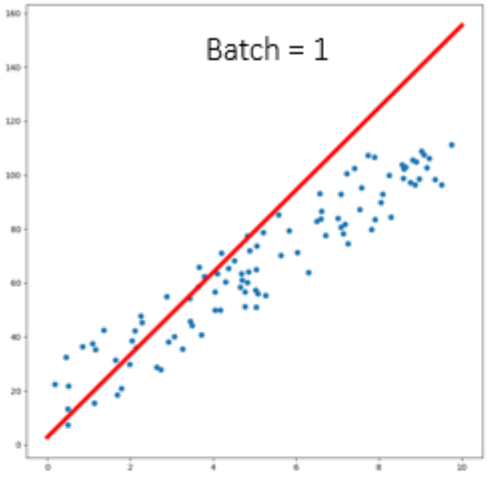
100 эпох

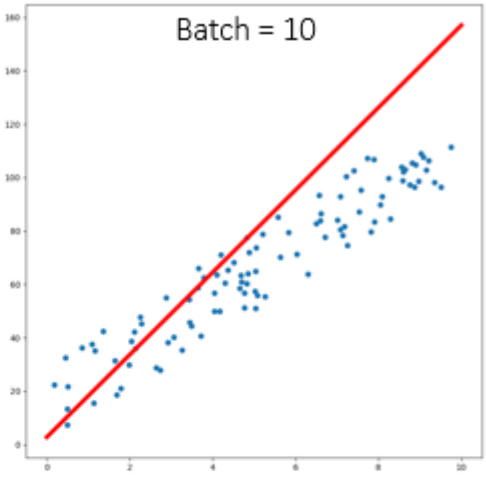


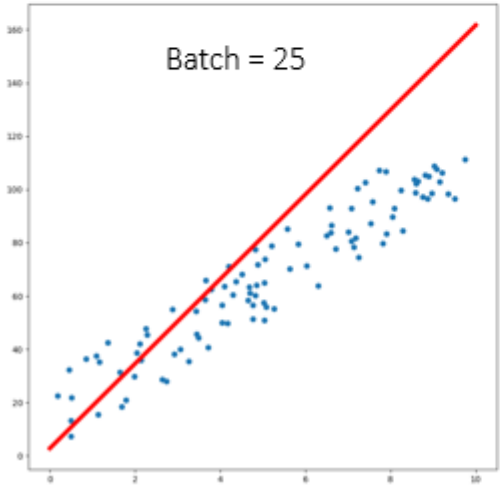
1000 эпох

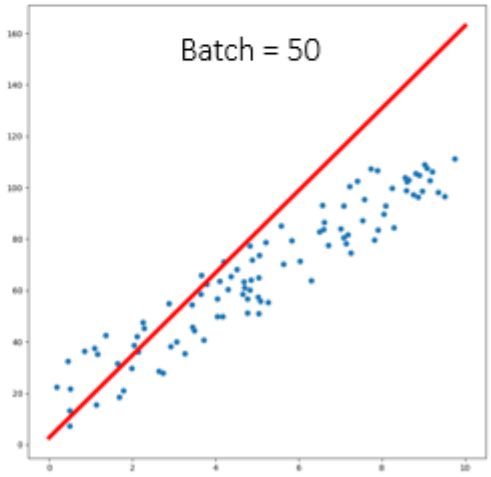


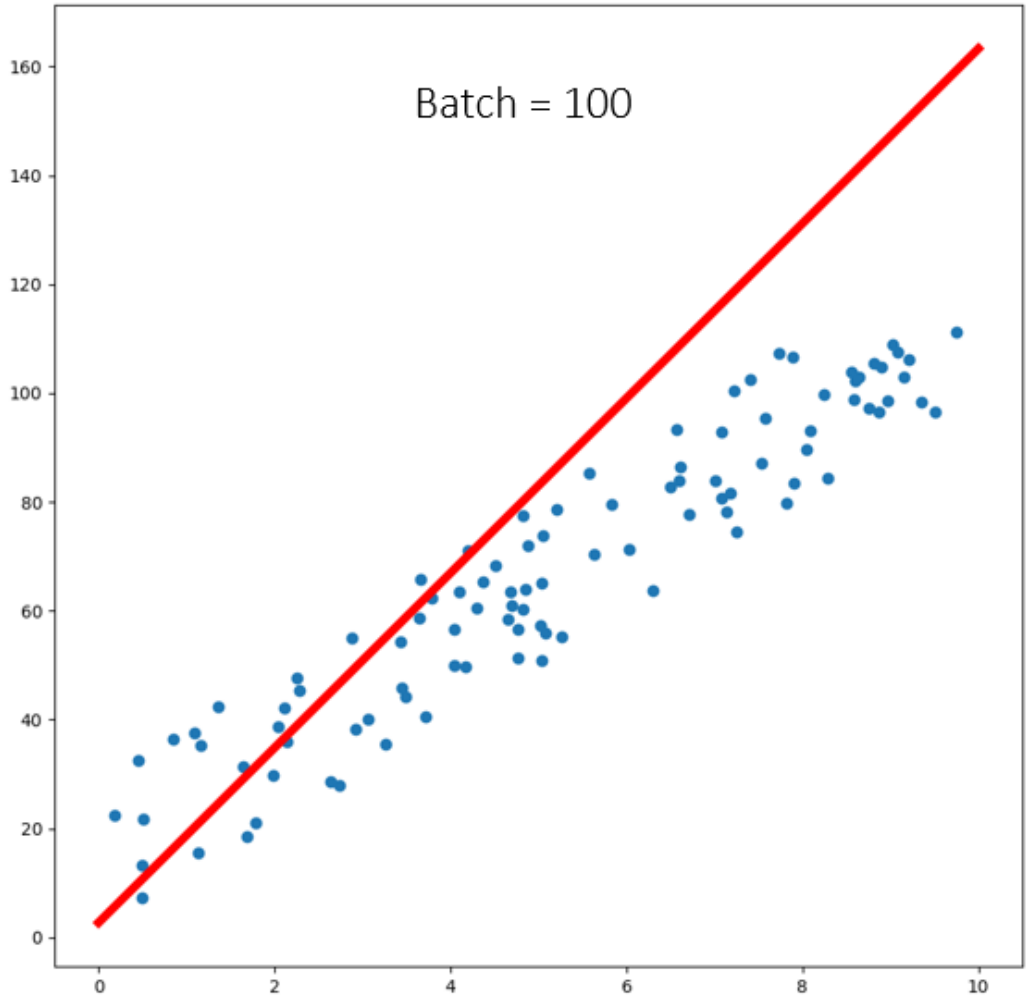
Nesterov:


Batch = 1

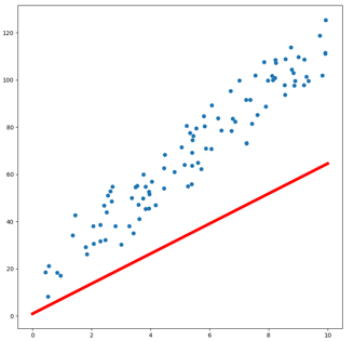

Batch = 10


Batch = 25

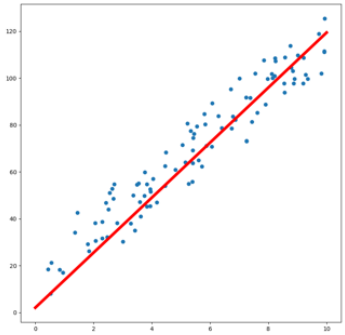

Batch = 50



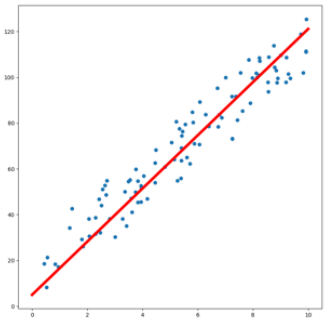
10 эпох



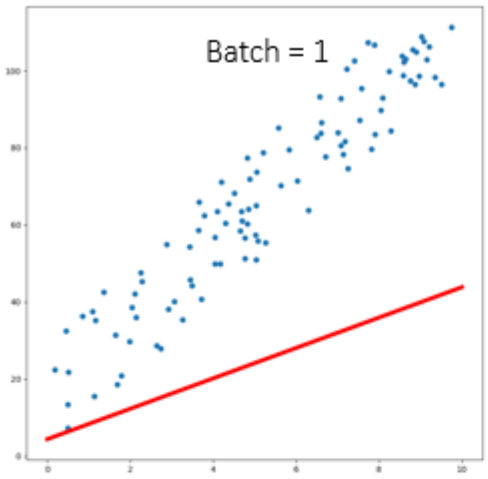
100 эпох

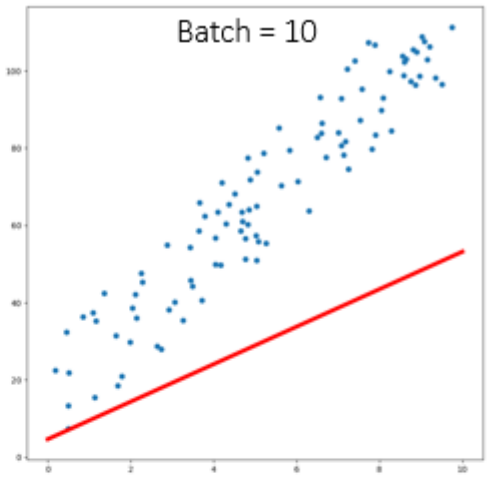


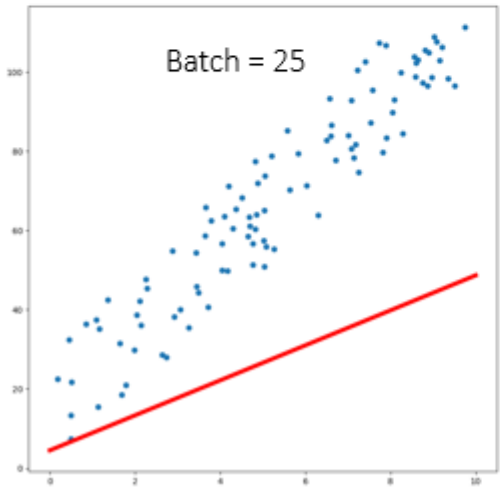
1000 эпох

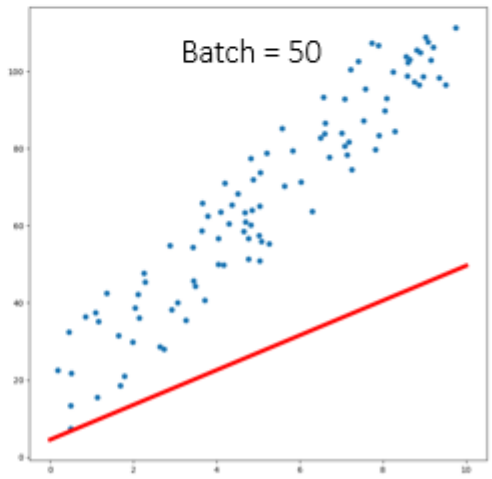


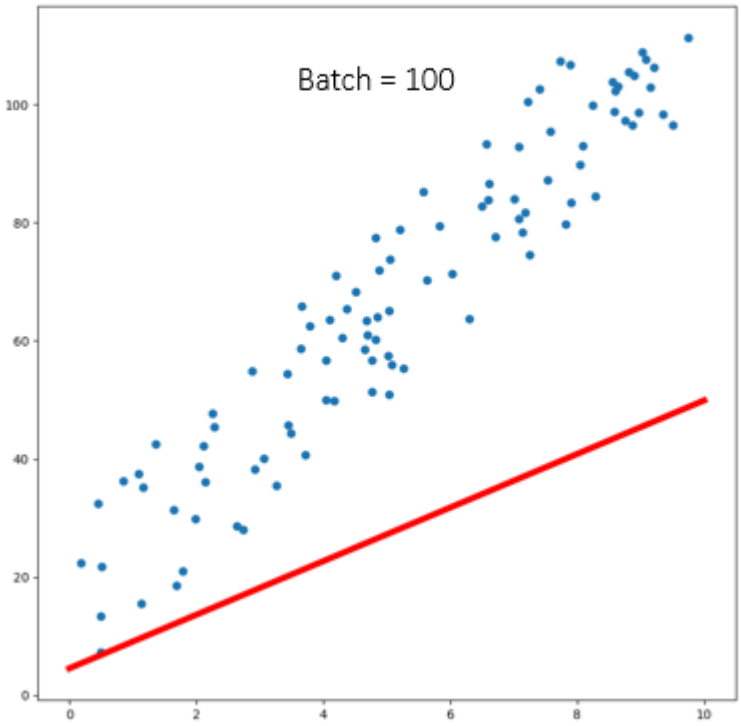
RMSProp:


Batch = 1

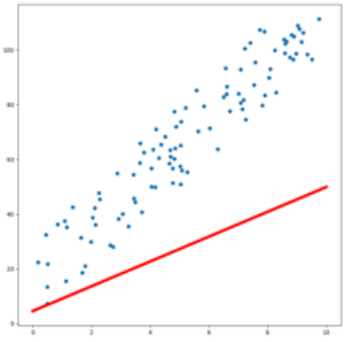

Batch = 10


Batch = 25

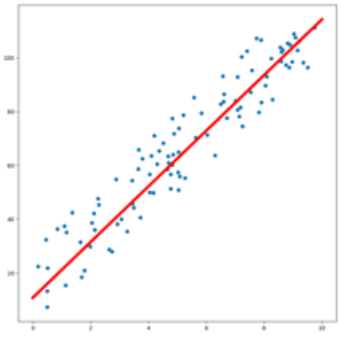

Batch = 50



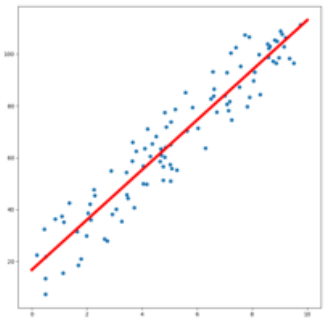
10 эпох



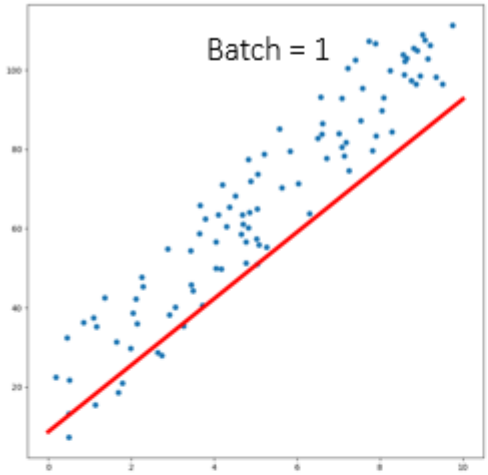
100 эпох

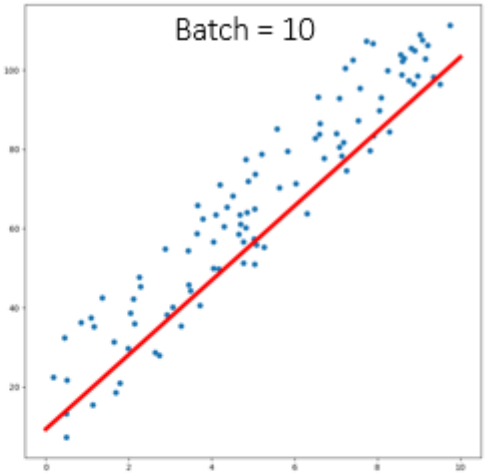


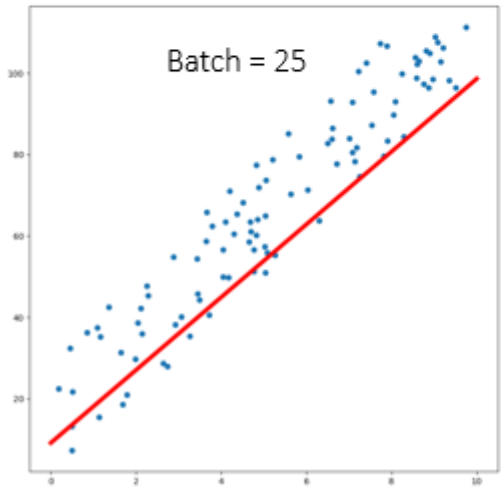
1000 эпох

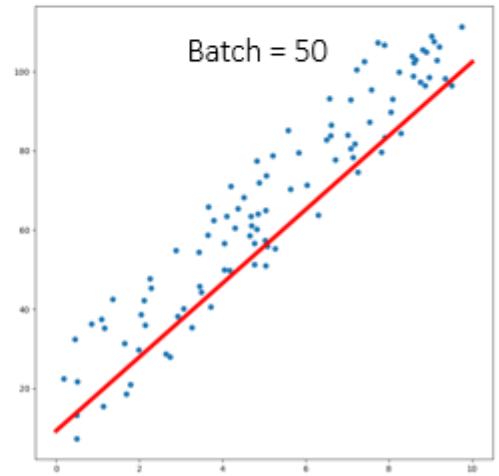


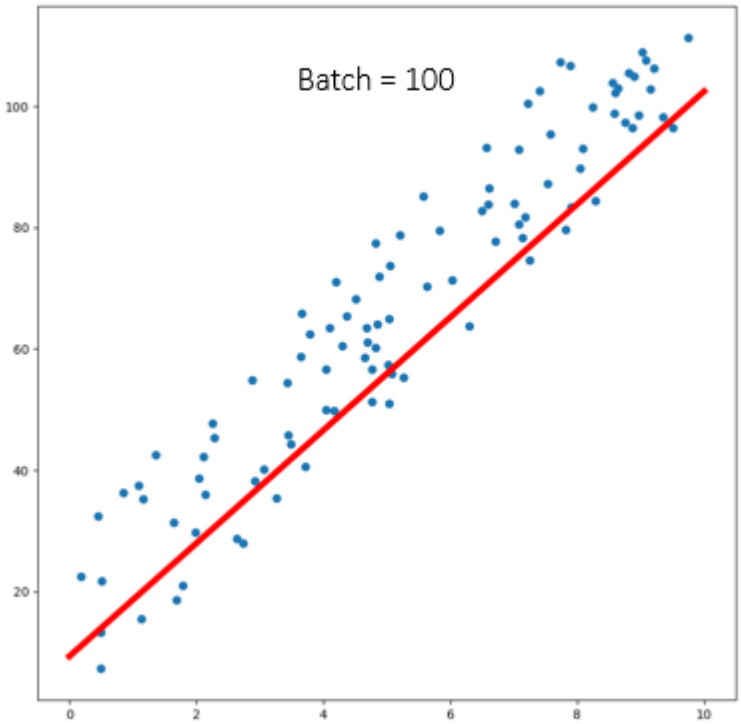
Adam:


Batch = 1

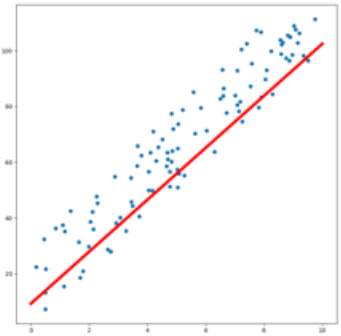

Batch = 10


Batch = 25

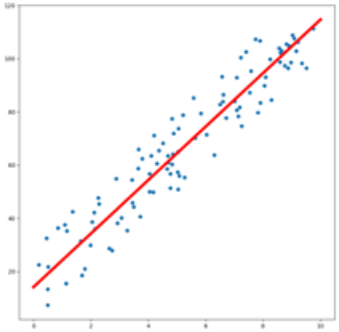

Batch = 50



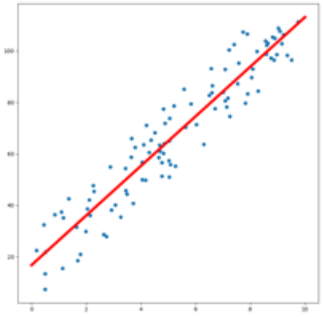
10 эпох



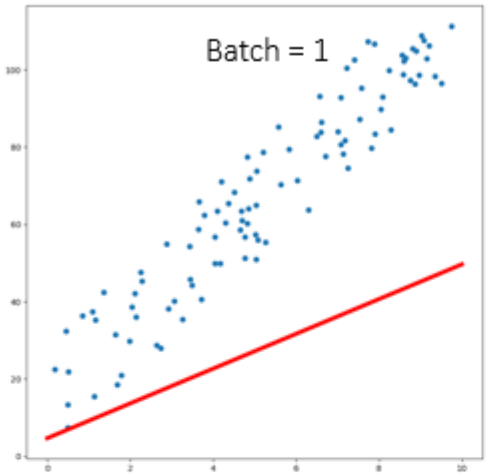
100 эпох

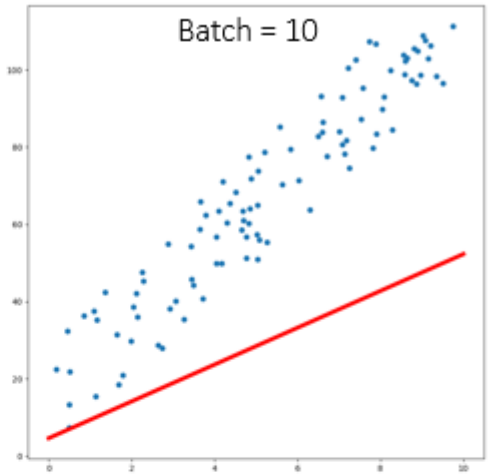


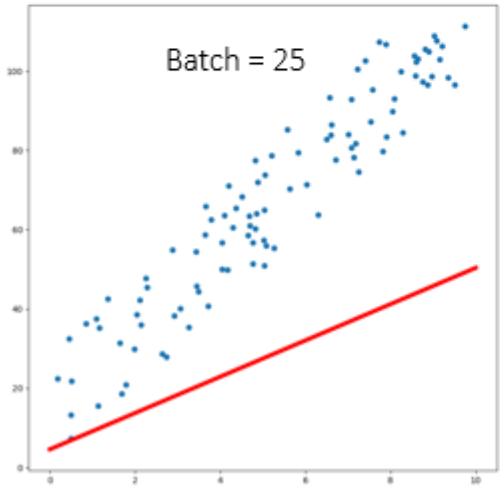
1000 эпох

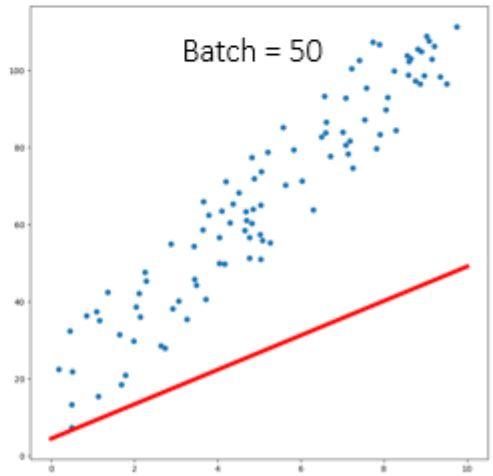


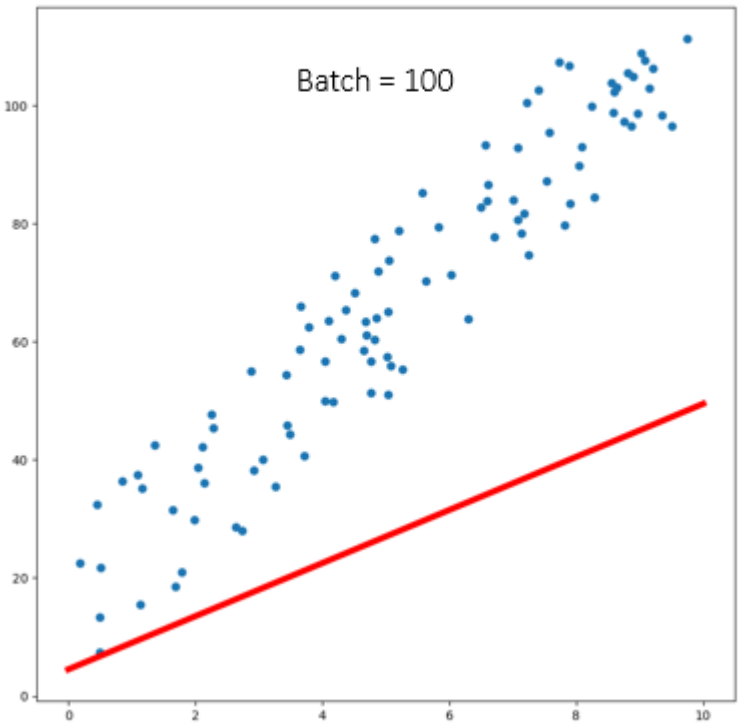
AdaGrad:


Batch = 1

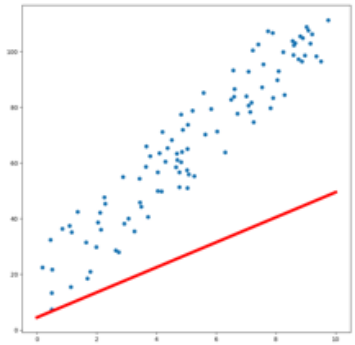
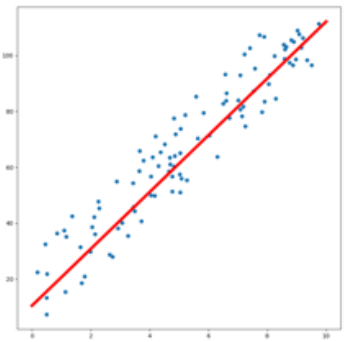
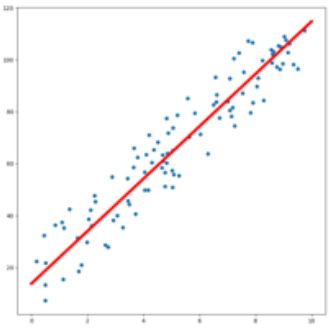

Batch = 10


Batch = 25


Batch = 50



10 эпох 100 эпох 1000 эпох

В целом, каждая из реализаций работает на ура. Какие-то сразу дают хорошие результаты, какие-то после. А насколько быстрее это по сравнению с нашими реализациями из 2 лабы?

Сравним работу различных методов наших самописных методов, используя разные learning_rate, результаты вычислений представлены в таблице ниже, для улучшения оценки все алгоритмы запускались на функции для линейной регрессии, использовавшей 100 точек, на выполнение давалось 10000 эпох, начальная точка (0; 0) и работало оно до тех пор, пока градиент не становился меньше эпсилона:

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

А теперь результаты с помощью PyTorch:

Критерий сравнения	SGD	Momentum	Nesterov	Adagrad	RMSProp	Adam
Среднее количество итераций	4073	400	401	537	892	632
Среднее время работы алгоритма, мс	48.432	24.264	26.684	14.465	25.455	23.894

Как можно заметить, в SGD методы из библиотеки работают в 2 раза быстрее и эффективнее, а вот со всеми остальными методами скорость возрастает почти в 4 раза, что поражает. Отсюда легко заключить вывод, что PyTorch умные люди придумали и набросали несколько оптимизаций на каждый из этих методов, чтобы он работал как можно быстрее!

Исследование эффективности и сравнение с собственными реализациями методов из библиотеки SciPy

Библиотека scipy.optimize предоставляет предоставляет несколько часто используемых алгоритмов оптимизации. Здесь мы сравнили работы методов Powell's DogLeg, BFGS и L-BFGS из пакета с нашими реализациями из 3-ей лабораторной работы

Для сравнения была взята функция Розенброка $f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$ и стартовая точка (-2, 2). Напомним, что минимум данной функции находится в точке (1, 1)

Результаты сравнения представлены в таблицах 2.а.1 и 2.а.2

Таблица 2.а.1 – методы библиотеки SciPy

	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Кол-во итераций	275	26	35	38
Кол-во вычислений функции	296	27	126	150
Кол-во вычислений градиента	296	24	42	50
Кол-во вычислений гессиана	275	23	0	0
Кол-во вычислений функции на итерацию	1,08	1,04	3,60	3,95
Кол-во вычислений градиента на итерацию	1,08	0,92	1,20	1,32
Кол-во вычислений гессиана на итерацию	1,00	0,88	0,00	0,00

Таблица 2.а.2 – методы, реализованные в лабораторной работе №3

	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Кол-во итераций	343	45	23	95
Кол-во вычислений функции	367	45	942	384
Кол-во вычислений градиента	367	44	173	96

Кол-во вычислений гессиана	343	43	0	0
Кол-во вычислений функции на итерацию	1,07	1,00	40,96	4,04
Кол-во вычислений градиента на итерацию	1,07	0,98	7,52	1,01
Кол-во вычислений гессиана на итерацию	1,00	0,96	0,00	0,00

Как видим из таблиц, наши методы, реализованные в третьей лабораторной, хоть и местами не сильно, но всё же уступают по мощности методам библиотеки SciPy

Сравнение работы метода для вычисления градиента из библиотеки PyTorch с другими реализациями

Для вычисления градиента функции можно использовать различные методы. Можно реализовать какой-то свой собственный метод либо воспользоваться стандартными библиотеками. В данном пункте мы сравнили реализацию метода вычисления градиента из библиотеки `PyTorch` с реализациями методов из библиотек `Numpy` и `Numdifftools`, а также с нашей собственной реализацией вычисления градиента. В отличие от сложных методов вычисления градиентов, наш собственный метод является простым вычислением градиента функции по определению, а именно: $\nabla f(x) = \lim\limits_{||h|| \to 0} \frac{f(x+h)-f(x)}{||h||}$

Таким образом код для вычисления градиента выглядит следующим образом:

```
def gradient(f, x, h=10e-6):
    grad = np.zeros(len(x))
    delta_x = x.copy()
    for i in range(len(x)):
        delta_x[i] += h
        grad[i] = (f(delta_x) - f(x)) / h
        delta_x[i] -= h
    return grad
```

Однако, наша реализация сильно зависит от выбора константы h, что и отражено в таблицах 2.b.1, 2.b.2

Таблица 2.b.1 – сравнение вычисления градиента

Функция	PyTorch gradient	gradient, h=10e-6	gradient, h=1e-7
$f(x) = x^2$ [-4.0, -2.0, -1.0, 0.0, 1.0, 2.0, 4.0]	[-6.0000, -4.0000, -2.0000, 0.0000, 2.0000, 4.0000, 6.0000]	[-7.99999, -3.99999, -1.99999, 1.e-05, 2.00001, 4.00001, 8.00001]	[-7.99999988, -3.9999999, -1.9999999, 1.e-07, 2.0000001, 4.00000009, 8.00000013]
$f(x) = x_0^2 + x_1^2$ [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-4, 0], [0, 0], [4, 0], [-4, 4], [0, 4], [4, 4]]	[[-3.99999000e+00, 1.00000008e-05], [1.e-05, 1.e-05], [4.00001000e+00, 1.00000008e-05], [-3.99999, 4.00001], [1.00000008e-05, 4.00001000e+00], [4.00001, 4.00001]]	[-3.99999990e+00, 9.76996262e-08], [1.e-07, 1.e-07], [4.00000009e+00, 8.88178420e-08], [-3.9999999 , 4.00000008], [9.76996262e-08, 4.00000009e+00], [4.00000008, 4.00000008]
$f(x) = x_0^2 + 2x_0x_1 - 4x_1^2$ [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-4, -4], [0, 0], [4, 4], [0, -20], [4, -16], [8, -12]]	[[-3.99999,-4.00004], [1.e-05, -4.e-05], [4.00001, 3.99996], [1.00001785e-05, -2.00000400e+01], [4.00001, -16.00004], [8.00001, -12.00004]	[[-3.9999999, -4.0000004], [1.e-07, -4.e-07], [4.00000009, 3.99999959], [1.06581410e-07, -2.00000004e+01], [4.0000001 , -16.00000036], [8.00000009, -12.00000039]]
$f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$ Функция Розенброка [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-3206, -800], [-2, 0], [3202, -800], [-1606, -400], [-2, 400], [1602, -400]]	[-3205.97599011, -799.99900001], [-1.99999000e+00, 9.99999994e-04] [3202.0240101 , -799.99900001] [-1605.97999009, -399.999], [-2.00399001, 400.001] [1602.02001009, -399.999]	[-3205.99976021, -799.99998889], [-1.9999999e+00, 1.0000889e-05], [3202.00023225, -799.99999571], [-1605.99980006, -399.99998933], [-2.00003967, 400.00000922], [1602.00019593, -399.99999274]

Функция	Numpy gradient	Numdifftools gradient
$f(x) = x^2$ [-4.0, -2.0, -1.0, 0.0, 1.0, 2.0, 4.0]	[-12. -7.5 -2. 0. 2. 7.5 12.]	[-8, -4, -2, 0, 2, 4, 8]
$f(x) = x_0^2 + x_1^2$ [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-4, 0], [0, 0], [4, 0], [-4, 4], [0, 4], [4, 4]]	[[-4, 0], [0, 0], [4, 0], [-4, 4], [0, 4], [4, 4]]
$f(x) = x_0^2 + 2x_0x_1 - 4x_1^2$ [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-4, -4], [0, 0], [4, 4], [0, -20], [4, -16], [8, -12]]	[[-4, -4], [0, 0], [4, 4], [0, -20], [4, -16], [8, -12]]
$f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$ Функция Розенброка [[-2, 0], [0, 0], [2, 0], [-2, 2], [0, 2], [2, 2]]	[[-3206, -800], [-2, 0], [3202, -800], [-1606, -400], [-2, 400], [1602, -400]]	[[-3206, -800], [-2, 0], [3202, -800], [-1606, -400], [-2, 400], [1602, -400]]

Таблица 2.b.2 – сравнение времени работы вычисления градиента

Функция	PyTorch gradient	gradient, h=10e-6	gradient, h=1e-7	Numpy gradient	Numdifftools gradient
$f(x) = x^2$	1.0035 ms	0.05 ms	0.05 ms	0.061 ms	6.1145 ms
$f(x) = x_0^2 + x_1^2$	1.0829 ms	0.06 ms	0.06 ms	0.061 ms	7.005 ms
$f(x) = x_0^2 + 2x_0x_1 - 4x_1^2$	1.0753 ms	0.063 ms	0.064 ms	0.066 ms	7.0295 ms
$f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$ Функция Розенброка	1.0813 ms	1.003 ms	1.004 ms	1.0023 ms	6.1274 ms

Исследование работы методов из SciPy при задании границ изменения параметров

В данной части лабораторной работы мы сравнили работу методов Powell's DogLeg и L-BFGS из библиотеки SciPy при задании области изменения аргументов, результаты можно наблюдать в таблицах 2.с.1-2.с.6

Powell's DogLeg:

Функция Розенброка $f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$

Минимум в точке (1; 1)

Таблица 2.с.1

Границы	Конечная точка	Число итераций	Число вычислений функции
[-10; 10] x [-10; 10] Старт (3, -1)	[1.00000003, 1.00000005]	16	642
[-3, 3] x [-3, 3] Старт (3, -1)	[-0.65398626, 0.43303074]	7	226
[-9; -5] x [-9; -5] Старт (-7, -7)	[-5.00003434, -5.]	2	138
[-7; 0] x [-7; 0] Старт (-5, -5)	[-2.58465629e-04, -9.94136546e-09]	2	137
[-4; 1] x [-4; 1] Старт (-3, 1)	[-0.17841219, 0.04201739]	8	301

Квадратичная функция $f(x) = 0.01x_0^2 + x_1^2$

Минимум в точке (0; 0)

Таблица 2.с.2

Границы	Конечная точка	Число итераций	Число вычислений функции
[-10; 10] x [-10; 10] Старт (3, -1)	[0., 0.]	2	26
[-3; 3] x [-3; 3] Старт (3, -1)	[0.00000000e+00, 1.11022302e-16]	2	26
[-9; -5] x [-9; -5] Старт (-7, -7)	[-5. , -5.00003434]	2	138

[5; 10] x [5; 10] Старт (7, 7)	[5.00000001, 5.00008156]	2	139
[-4; 0] x [-4; 0] Старт (-3, -1.5)	[-3.07094057e-10, -1.80337419e-17]	10	802

Функция $f(x) = 0.25\sqrt{e^{x_0} + e^{-x_0} + e^{x_1} + e^{-x_1}} - x_0^2 - 0.5x_0$

Минимум примерно в точке (10.25, 0)

Таблица 2.с.3

Границы	Конечная точка	Число итераций	Число вычислений функции
[-5; 15] x [-5; 15] Старт (3, -1)	[1.02475275e+01, -5.46402778e-06]	2	44
[8; 13] x [-3; 3] Старт (9, -1)	[1.02475440e+01, -2.22044605e-16]	2	34
[-5; 0] x [-10; -5] Старт (-3, -7)	[-4.99991844, - 5.00000001]	2	139
[15; 20] x [5; 10] Старт (17, 7)	[15.00006144, 5.00000001]	2	139
[10.25; 11] x [-3; 0] Старт (10.75, -1.5)	[1.02500351e+01, -1.37575029e-09]	2	129

L-BFGS:

Функция Розенброка $f(x) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2$

Минимум в точке (1; 1)

Таблица 2.с.4

Границы	Конечная точка	Число итераций	Число вычислений функции	Число вычислений гессиана
[-10; 10] x [-10; 10] Старт (3, -1)	[0.999997 , 0.99999399]	42	156	52
[-3, 3] x [-3, 3] Старт (3, -1)	[0.99999699, 0.99999398]	49	180	60
[-9; -5] x [-9; -5] Старт (-7, -7)	[-5., -5.]	1	6	2
[-7; 0] x [-7; 0] Старт (-5, -5)	[0., 0.]	1	6	2
[-4; 1] x [-4; 1] Старт (-3, 1)	[1., 1.]	1	6	2

Квадратичная функция $f(x) = 0.01x_0^2 + x_1^2$

Минимум в точке (0; 0)

Таблица 2.с.5

Границы	Конечная точка	Число итераций	Число вычислений функции	Число вычислений гессиана
[-10; 10] x [-10; 10] Старт (3, -1)	[-6.77552764e-06, 2.25826545e-07]	5	27	9
[-3; 3] x [-3; 3] Старт (3, -1)	[-6.77569061e-06, 2.25834924e-07]	5	27	9
[-9; -5] x [-9; -5] Старт (-7, -7)	[-5., -5.]	3	18	6
[5; 10] x [5; 10] Старт (7, 7)	[5., 5.]	3	18	6
[-4; 0] x [-4; 0] Старт (-3, -1.5)	[-9.84861986e-08, 0.00000000e+00]	3	18	6

Функция $f(x) = 0.25\sqrt{e^{x_0} + e^{-x_0} + e^{x_1} + e^{-x_1}} - x_0^2 - 0.5x_0$

Минимум примерно в точке (10.25, 0)

Таблица 2.с.6

Границы	Конечная точка	Число итераций	Число вычислений функции	Число вычислений гессиана
[-5; 15] x [-5; 15] Старт (3, -1)	[1.02475431e+01, 3.06323116e-03]	17	57	19
[8; 13] x [-3; 3] Старт (9, -1)	[1.02475219e+01, 6.88718792e-04]	12	48	16
[-5; 0] x [-10; -5] Старт (-3, -7)	[-5., -5.]	1	6	2
[15; 20] x [5; 10] Старт (17, 7)	[15., 5.]	5	30	10
[10.25; 11] x [-3; 0] Старт (10.75, -1.5)	[1.02500000e+01, -3.01437108e-04]	6	33	11

Внимательно проанализировав таблицы выше, можно заметить следующее:

1. Для случая, когда минимум функции находится в заданной области, чем меньше эта область, тем в среднем быстрее ищется оптимум, хотя количество итераций зависит сильно от самой функции и от стартовой точки
2. Для случая, когда минимум функции находится за пределами области, методы из SciPy срабатывают относительно быстро. Это объясняется тем, что в наших случаях за пределами минимума функции вели себя так, что их градиент был одинаков по знаку, а значит не происходило ситуации, когда градиентный спуск промахивался мимо оптимума
3. Для случая, когда минимум функции лежит на границе области, нельзя точно сказать, как именно поведёт себя метод, так как иногда он может дать не совсем ожидаемый результат, хоть и близко приближённый к реальному значению оптимума

Использование линейных и нелинейных ограничений

При оптимизации функций с помощью средств библиотеки scipy.optimize есть возможность задать ограничения в виде области, в границах которой будет производиться поиск локального минимума. Ограничения могут быть линейными (LinearConstraint) и

нелинейными (NonlinearConstraint) и задаются в аргументах функции scipy.optimize.minimize().

Линейные ограничения представляют собой систему линейных неравенств

$$\begin{cases} b_1 \le a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le c_1 \\ b_2 \le a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \le c_2 \\ \vdots \\ b_m \le a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n \le c_m \end{cases}$$

И задаются матрицей коэффициентов A, векторами нижних и верхних границ b и c

$$b \le Ax \le c$$

где A – матрица $m \times n$,

b, c – вектора $\mathbb{R}^m$,

x – вектор $\mathbb{R}^n$.

Пример задания линейных ограничений приведён ниже.

```
point = np.array([1, 2])
A = [[1, -1], [3, 2], [0.5, 4]]
lb = [12, -4, 0]
ub = [3, 1, -7]
result = scipy.optimize.minimize(fun=f, x0=point, method='trust-constr',
                        constraints=scipy.optimize.LinearConstraint(A, lb, ub)
```

Нелинейные ограничения – это система произвольных неравенств

$$\begin{cases} b_1 \le f_1(x) \le c_1 \\ b_2 \le f_2(x) \le c_2 \\ \vdots \\ b_m \le f_m(x) \le c_m \end{cases}$$

Такие ограничения определяются функцией и двумя векторами нижних и верхних границ

$$b \le f(x) \le c$$

где f – функция $\mathbb{R}^n \to \mathbb{R}^m$,

b, c – вектора $\mathbb{R}^m$,

x – вектор $\mathbb{R}^n$.

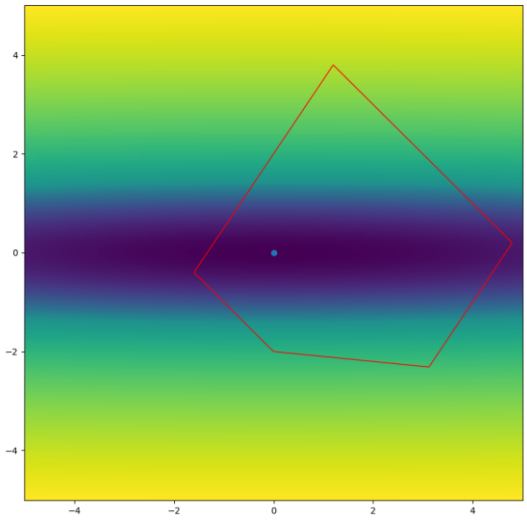
Оптимизировать функцию с учётом наложенных на вектор нелинейных ограничений можно следующим образом.

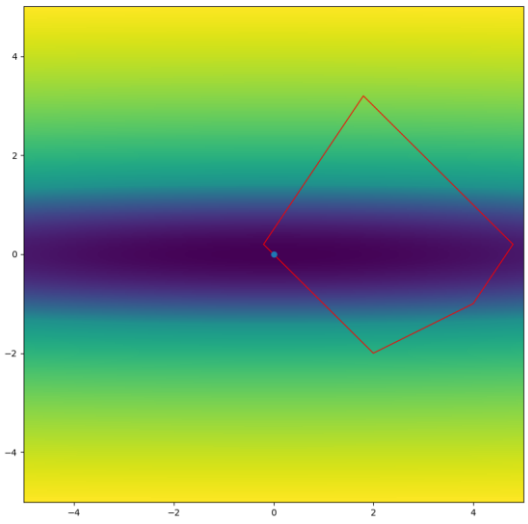
```
point = np.array([3, -1])

A = [[1, 1], [1, -2], [0.5, 1]]
```

```
lb = [11, -np.inf, -np.inf]

ub = [np.inf, 1, -66]

result = scipy.optimize.minimize(fun=f, x0=point, method='trust-constr',
constraints=scipy.optimize.NonlinearConstraint(fun=constraint, lb=lb, ub=ub))
```

Ниже приведено сравнение результатов оптимизации функции с различными ограничениями и без них.

$f(x, y) = 0.01x^2 + y^2$, Linear Constraint



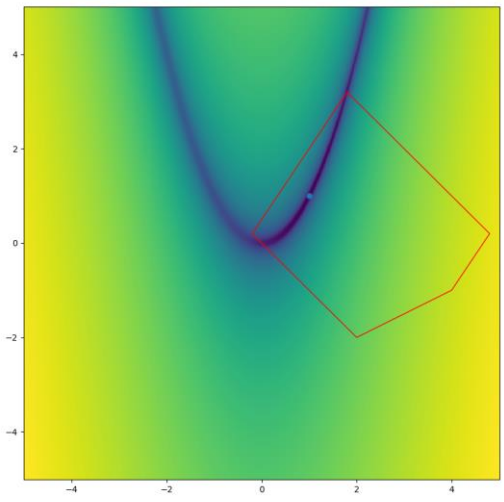


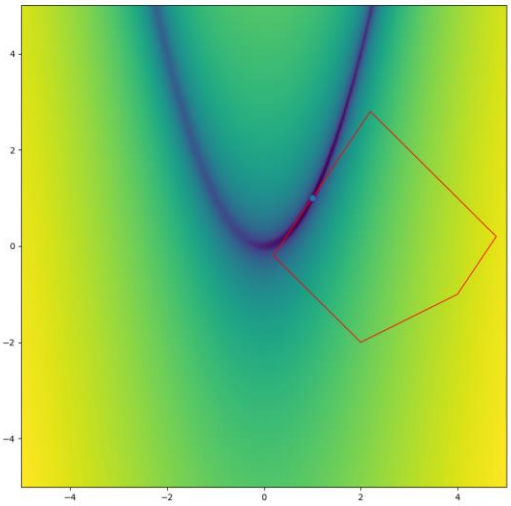
	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	21	23	27
Количество вычислений функции	63	45	57
Количество вычислений градиента	21	15	19
Погрешность вычисления минимума	0	0,00001	0,00005

$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$, Linear Constraint

	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	56	152	103

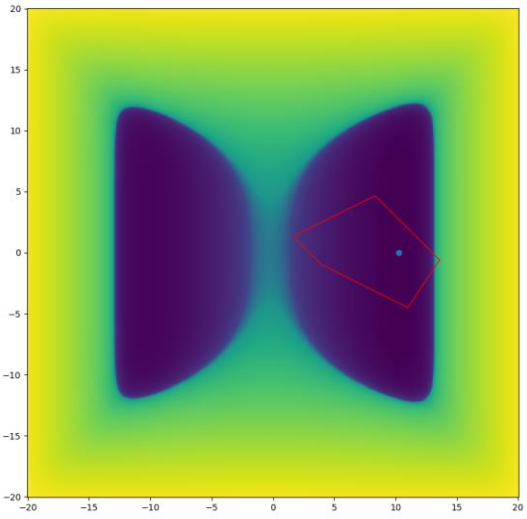
Количество вычислений функции	156	666	390
Количество вычислений градиента	52	222	130
Погрешность вычисления минимума	0	0	0,0008

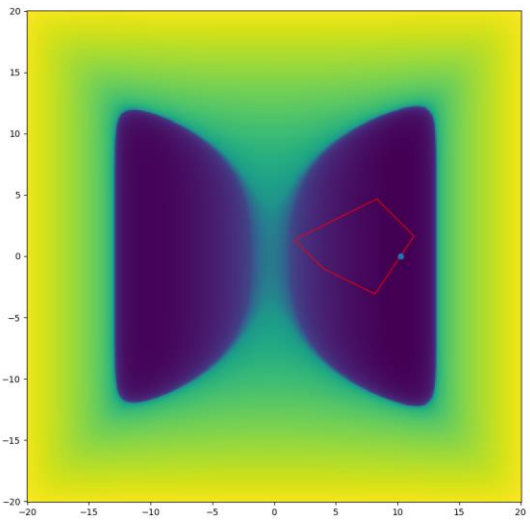




$f(x,y) = 0.25\sqrt{e^x + e^{-x} + e^y + e^{-y}} - x^2 - 0.5x$, Linear Constraint

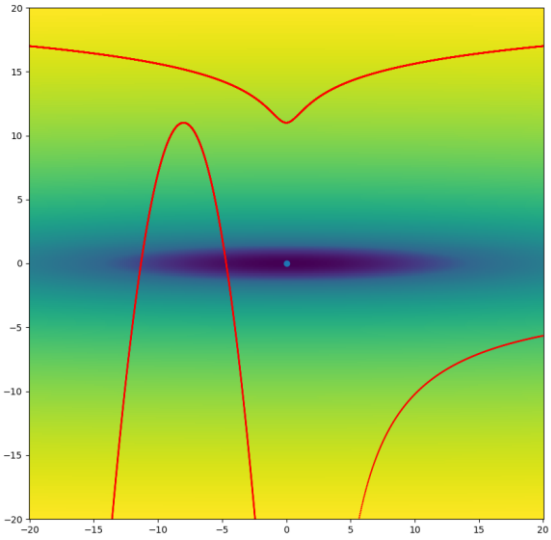
	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	19	66	29
Количество вычислений функции	57	273	63
Количество вычислений градиента	19	91	21
Погрешность вычисления минимума	0	0,00006	0,0001

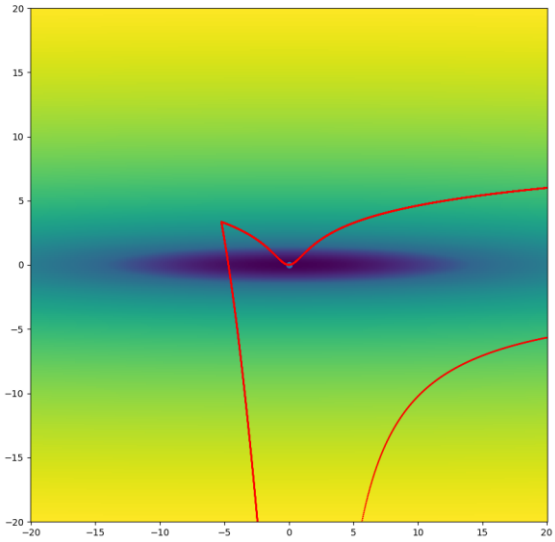




$f(x, y) = 0.01x^2 + y^2$, Nonlinear Constraint

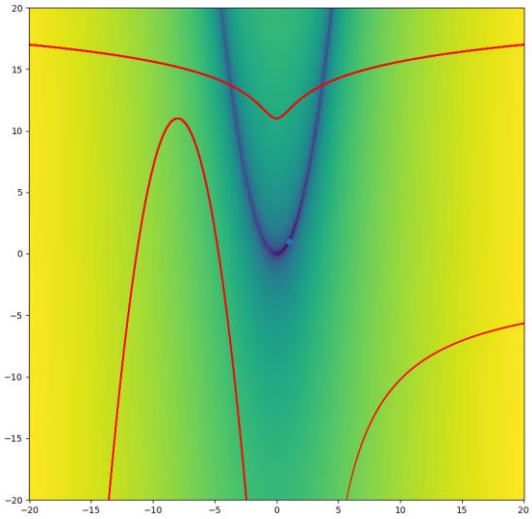
	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	21	31	125
Количество вычислений функции	63	93	537
Количество вычислений градиента	21	31	179
Погрешность вычисления минимума	0	0,006	0,02

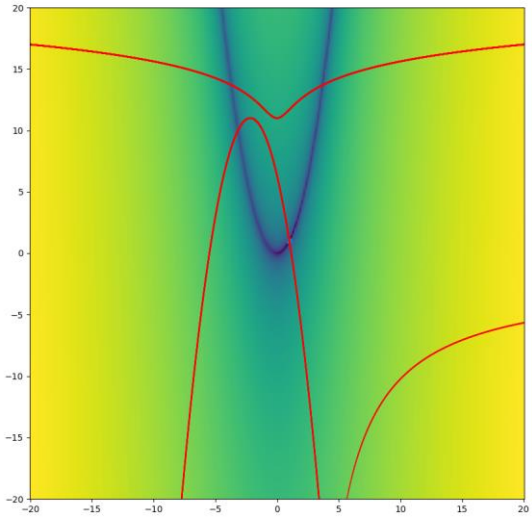




$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$, Nonlinear Constraint

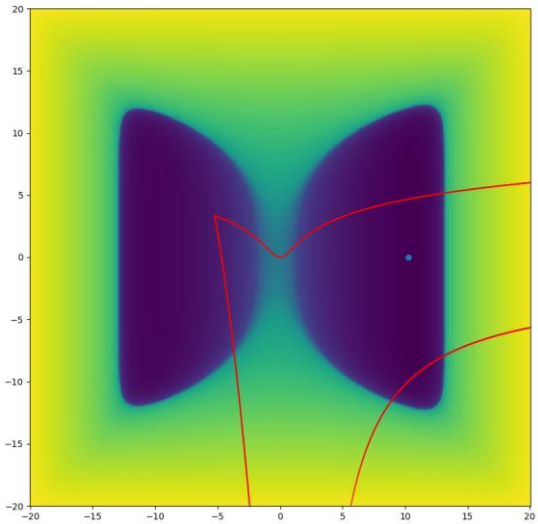
	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	21	31	125
Количество вычислений функции	63	93	537
Количество вычислений градиента	21	31	179
Погрешность вычисления минимума	0	0,006	0,02

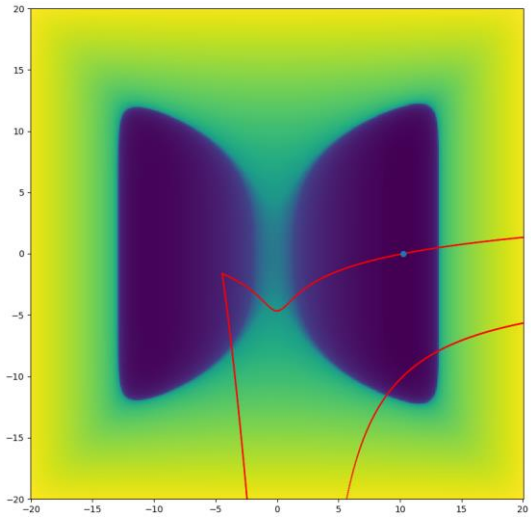




$f(x,y) = 0.25\sqrt{e^{x} + e^{-x} + e^{y} + e^{-y}} - x^{2} - 0.5x$, Linear Constraint

	Без ограничений	Минимум внутри области	Минимум на границе
Количество итераций	19	24	86
Количество вычислений функции	57	45	240
Количество вычислений градиента	19	15	80
Погрешность вычисления минимума	0	0,07	0,09





Как видно из полученных данных, поиск минимума с учётом заданных ограничений требует большее количество итераций алгоритма, а, следовательно, и больше вычислений функции и градиента. Кроме того, при заданных ограничениях алгоритм находит значение точки минимума с некоторой погрешностью, которая особенно заметна,

когда полученный минимум лежит на границе области. Линейные ограничения являются более простыми, чем произвольные, заданные функциями, поэтому оптимизация с линейными ограничениями происходит быстрее и точнее по сравнению с нелинейным случаем.