Университет ИТМО

Методы оптимизации, лабораторная работа №4. Отчет

Работу выполнили
Миронов Кирилл
Корнильев Артемий

Проверяющий:
к.т.н, доцент ______Х.З. Здесь будет один из допущенных проверяющих

Санкт-Петербург 2023

Содержание

1 Постановка задачи

1. Изучить использование вариантов SGD (torch.optim) из PyTorch. Исследовать эффективность и сравнить с собственными реализациями из 2 работы.

2. Изучить использование готовых методов оптимизации из SciPy (scipy.optimize.minimize, scipy.optymize.least_squares)

   (a) Исследовать эффективность и сравнить с собственным реализациями из 3 работы.

   (b) Реализовать использование $PyTorch$ для вычисления градиента и сравнить с другими подходами.

   (c) Исследовать как задание границ изменения параметров влияет на работу методов из $SciPy$.

3. Исследовать использование линейных и нелинейных ограничений при использовании scipy.optimize.minimize из SciPy (scipy.optimize.LinearConstraint и scipy.optimize.NonlinearConstraint).Рассмотреть случаи когда минимум находится на границе заданной области и когда он расположен внутри.

2 Задача 1.

Для сравнения методов реализуем SGD с помощью библиотеки *torch* и сопоставим с данными от методов в собственной реализации на $f(x) = 2.6 + 3 * x$

```
class LinearRegression(torch.nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

def torch_optim():
    eps = 1e-6
    model = LinearRegression()
    time_start = time.time()
    tracemalloc.start()
    criterion = torch.nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)
    prev_loss = None
    num_epochs = 300
    for epoch in range(num_epochs):
        outputs = model(x)
        loss = criterion(outputs, y)
        if prev_loss is not None and abs(prev_loss - loss) < eps:
            break
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        prev_loss = loss
    time_end = time.time()
    mem = tracemalloc.get_tracemalloc_memory()
    tracemalloc.stop()
    for name, param in model.named_parameters():
        if param.requires_grad:
            print(param.data.item(), epoch, time_end - time_start,
                mem, loss.item())
```

Данные полученные от методов, которые реализовали мы.

Name	W	Iterations	Time (s)	Memory (bytes)	loss
SGD	$[2.6035615, 3.0011443]$	101	0.66500	1824	0.78036
$Momentum$	$[2.603494, 3.00116]$	86	0.56552	1632	0.780361
$Adagrad$	$[2.6035750, 3.0011459]$	29	0.18199	1600	0.78036
$RMSprop$	$[2.6035762, 3.0011441]$	28	0.17400	1632	0.78360
$Adam$	$[2.6035670, 3.0011817]$	10	0.06399	2240	0.78036

Данные полученные с помощью методов, реализуемые библиотекой.

Name	W	Iterations	Time (s)	Memory (bytes)	loss
SGD	$[2.60159, 2.99522]$	280	0.1562	17456	0.00784
$Momentum$	$[2.60217, 2.99874]$	123	0.063	15984	0.00781
$Adagrad$	$[2.60330, 3.00013]$	30	0.0279	8496	0.007804
$RMSprop$	$[2.6038, 3.0014]$	11	0.0099	8176	0.007803
$Adam$	$[2.60553, 3.00101]$	127	0.108	19696	0.00780

Из данных приведенных в таблицах, можно сделать вывод, что все методы отработали достаточно точно, но при этом методы с собственной реализацией проработали за большее количество времени и затратили меньше памяти, в свою очередь методы из библиотеки оказались быстрее, но потребляют больше памяти, частично это связанно с тем, что в библиотеке используются оптимизации для производительности.

3 Задача 2a

Для анализа Gauss-Newton, BFGS , L-BFGS используем функции $f(x) = (5*x)^2 + \cos x * 12^2$ для Gauss-Newton и $f(x, y) = 0.5*x^2 + \sin y * 2 * \cos x + y * 2$ для BFGS и L-BFGS

```
def residuals(w, x, y):
    return y - (w[0] * x) ** 2 - np.cos(x) * w[1] ** 2
def func(w, x):
    return (w[0] * x) ** 2 + np.cos(x) * w[1] ** 2
res = least_squares(residuals, [1.0, 1.0], args=(X, y), method='lm')

def f_B(x):
    return 0.5 * x[0] ** 2 + np.sin(x[1]) * 2 * np.cos(x[0] + x[1] * 2)


initial_guess = [2.0, 2.0]
result = minimize(f_B, initial_guess, method='BFGS', tol=1e-7)
resultB = minimize(f_B, initial_guess, method='L-BFGS-B', tol=1e-7)
```

Данные полученные с помощью методов, реализуемые библиотекой.

Name	W	Iterations	Time (s)	Memory (bytes)	F(x)
GN	$[5.0, 12.0]$	20	0.002	10208	-
$BFGS$	$[7.059e-09, 1.571]$	39	0.011	19584	-1.999
$L-BFGS$	$[-4.041e-07, 1.571]$	33	0.0059	16032	-1.999

Данные полученные от методов, которые реализовали мы.

Name	W	Iterations	Time (s)	Memory (bytes)	F(x)
GN	$[5.0, 12.0]$	7	0.0169	5184	-
$BFGS$	$[-9.53360e-05, 1.570849]$	300	0.44299	10304	-1.99
$L-BFGS$	$[-1.38997e-08, 1.570796]$	10	0.01200	48224	-2.0

По данным в таблицах видно, что методы из библиотек на порядок быстрее отрабатывают, но при этом используют в 2 раза больше памяти, кроме L-BFGS, данный метод в нашей реализации использовал больше всего памяти.Но не смотря на различия в потребление ресурсов они достигают одинаковых результатов.

4 Задача 2b

```
def f(x):
    return 0.5 * x ** 2 + np.sin(x) * 2 * np.cos(x + x * 2)

def f_t(x):
    return 0.5 * x ** 2 + torch.sin(x) * 2 * torch.cos(x + x * 2)

def f2_t(x):
    return 0.5 * x ** 3 - torch.log2(x * 3 + 1)

def f2(x):
    return 0.5 * x ** 3 - np.log2(x * 3 + 1)

def num_grad(f, x, eps=1e-4):
    return (f(x + eps) - f(x - eps)) / (2 * eps)

def num_dif(f, x):
    df = nd.Gradient(f)
    return df(x)

def torch_grad(f, x):
    x_g = torch.tensor(x, requires_grad=True)
    z = f(x_g)
    z.backward()
    return x_g.grad.item()
```

Name	F(x)'	Time (s)	Memory (bytes)
num_grad_f1	4.45507518442855585	960	0.0
num_grad_f2	13.06719149262392	992	0.0
num_dif_f1	4.455075261629565	27232	0.00499
num_dif_f2	13.06719148773391	22240	0.0030002
torch_grad_f1	4.455075263977051	1456	0.003
torch_grad_f2	13.067191123962402	1168	0.0

Из таблицы выдно, что саммыми эффективными будут методы из torch и метод двусторонней разности, они использовали наименьшее время и память. Но наиболее выгодным для использования выглядит метод из torch, так-как он внутри содержит оптимизации, что может его ускорить на более сложных функциях.

5 Задача 2c

Проведем опыт на 4-ех отрезках $bounds = [[-1, 1], [-10, 10], [-50, 50], [-100, 100]]$ для функций $f(x) = 2 * x^2 + 0.5 * x^3 + 1$ и $f2(x) = 0.5 * x^2 + np.sin(x) * 2 * np.cos(x + x * 2)$

```
def f(x):
    return 2 * x ** 2 + 0.5 * x ** 3 + 1


def f2(x):
    return 0.5 * x ** 2 + np.sin(x) * 2 * np.cos(x + x * 2)
def test_bound(bound):
    ans1 = minimize(f, x0=-20, method='L-BFGS-B', bounds=(bound,))

    ans2 = minimize(f2, x0=3, method='L-BFGS-B', bounds=(bound,))
    print(ans1.x, ans1.fun, ans2.x, ans2.fun)
bounds = [[-1, 1], [-10, 10], [-50, 50], [-100, 100]]
for i in bounds:
    test_bound(i)
```

Name	F(x)	Time (s)	Memory (bytes)	x
f1_[-1,1]	1.00	0.0059	25792	1.76238e-08
f2_[-1,1]	-1.166	0.001	16128	1
f1_[-10,10]	-299	0.0009	16192	-10
f2_[-10,10]	0.2129	0.0069	23104	-1.9356
f1_[-50,50]	-57499	0.002	18272	-50
f2_[-50,50]	0.212	0.0059	22912	-1.9356
f1_[-100,100]	-479999	0.0019	18272	-100
f2_[-100,100]	0.212	0.006	22816	-1.9356

Границы задают, в каком диапозоне может быть наш ответ, это может , как положительно, так и отризательно повлиять, по этому надо подходить к заданию границ с осторожностью, к примеру , для $f1$ из точки -20 нас интересовал минимум в точке 0, и нам в этом помог отрезок от -1 до 1, т.к. при больших отрезках функция уходит на промежуток, который бесконечно убывает и не очень интресен для иследования.И наоборот при задание очень маленькой границы как с $f2$ и отрезком от -1 до 1, мы не смогли достичь фактического минимума и остановились на границах отрезка.Вывод: аналитически можно понять в каком отрезке лежит искомый минимум, и отсечь нежелательные значения, чтобы улучшить коррекстность.

6 Дополнительное задание

```
def f(x):
    return 2 * x ** 2 + 0.5 * x ** 3 + 1

A = np.array([[5.0]])
initial_guess = [10]
linear_constraint1 = LinearConstraint(A, -2.5, 3.0)
linear_constraint2 = LinearConstraint(A, 0, 1.0)



def constraint_function(x):
    return 2 * x ** 3 - x ** 2 + x


non_linear_constraint1 = NonlinearConstraint(constraint_function,
    lb=-5, ub=20)
non_linear_constraint2 = NonlinearConstraint(constraint_function,
    lb=0, ub=20)


result = minimize(f, initial_guess, method='SLSQP',
    constraints=non_linear_constraint)
optimized_parameters = result.x
optimal_value = result.fun
```

Name	F(x)	x	lb. ub.	memory	Time
linear$_{c}onstraint1$	1.000000094	-0.00021707	[-2.5, 3.0]	624	0.0
linear$_{c}onstraint1$	1.0	3.64153152e-13	[0, 1.0]	624	0.0
non$_{l}inear_{c}onstraint1$	1.0000001	0.000226	[-5, 20]	25120	0.0109
non$_{l}inear_{c}onstraint2$	1.0	8.81942686e-08	[0, 20]	25120	0.0139

Видно, что при задание ограничений, как линейный, так и нелиненых,чтобы минимум находил на границе, то значение будет более приблеженное к действительности.А также при нелинейной ограничении задание не на границе, метод отрабатывает немного быстрее.И методы использующие линейные ограничения используют меньше памяти и отрабатывают быстрее. чем при нелинейном ограничение.

Вывод: Мы научились использовать методы для оптимизации из библиотек, а также сравнили эффективность с темиже методами, которые реализовали сами