دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

تمرین شماره ۲ بهینه‌سازی محدب – بخش پیاده سازی

نگارش

امین عبدی‌پوراصل

۴۰۱۱۳۳۰۱۱

استاد درس

دکتر امیرمزلقانی

دی ۱۴۰۲

شـما همچنین می‌توانید از طریق لینک گیت‌هاب به کدها به صورت کامل نیز دسـترسـی داشـته باشید.

سوال ۱

هدف این سوال پیاده‌سازی روش کاهش گرادیان[1] (GD) با استفاده از PyTorch است. بدین منظور از تکه کدی که در صورت پروژه تعریف شده که از torch.optim.Optimizer ارث بری کرده است استفاده کرده‌ایم. هدف اصلی این بهینه ساز انجام به روز رسانی پارامترها بر اساس گرادیان تابع خطا با توجه به پارامترهای مدل است. آن را بدین صورت تکمیل نموده ایم:

```
import torch
import torch.nn as nn

class MyGD(torch.optim.Optimizer):
    def __init__(self, params, lr=0.001):
        defaults = dict(lr=lr)
        super(MyGD, self).__init__(params, defaults)

    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                p.data.add_(-group['lr'], grad)

        return loss
```

در این کد، نرخ یادگیری را ۰.۰۰۱ در نظر گرفت ایم و پارامترهای پیش فرض را در "Defaults" ذخیره کرده‌ایم. این کلاس برای هر پارامتر p، بر اساس نزول گرادیان و با اندازه ۰.۰۰۱ اصلاح را انجام می‌دهد.

[1] Gradient Descent

این کلاس به عنوان بهینه ساز استفاده می‌شود و پس از تعریف مدل و همچنین تعریف نحوه محاسبه خطا، آن را باید فراخوانی کرد و اصلاح را انجام داد و سپس با استفاده از آن‌ها آموزش را انجام داد تا به پاسخ بهینه برسیم. در ادامه به دو کاربرد از این کلاس برای مسائل بهینه سازی می‌پردازیم.

بدست آوردن پاسخ بهینه مسئله بهینه‌سازی

هدف در این بخش حل مسئله بهینه سازی زیر است تا نقطه بهینه آن محاسبه شود.

$$\min f(x_1, x_2) = \frac{x_1^2}{x_2}$$

$$Subject\ to\ x_2 > 0$$

بدین منظور باید یک نقطه شروع برای آن در نظر گرفت و پس از آن تابع هدف[1] که در اینجا $f(x_1, x_2) = \frac{x_1^2}{x_2}$ است را به همراه گرادیان آن بر حسب هر دو متغیر مسئله تعریف کنیم.

```
def objective(x1, x2):
    return x1**2 / x2

def grad_objective(x1, x2):
    grad_x1 = 2 * x1 / x2
    grad_x2 = -1 * x1**2 / x2**2
    return torch.tensor([grad_x1, grad_x2])

# Initial values
x1 = torch.tensor([1.0], requires_grad=True)
z = torch.tensor([0.0], requires_grad=True)
```

سپس به سراغ بهینه کردن تابع هدف می‌رویم. پس از فراخوانی MyGD که تعریف کرده بودیم، برای ۱۰۰۰ ایپاک مقدار تابع و خطا را محاسبه و در خلاف جهت گرادیان با گام ۱۰۰۰ حرکت می‌کنیم و برای هر گام مقدار تابع و خطا را نمایش می‌دهیم.

[1] Objective Function

```
optimizer = MyGD([x1, z], lr=0.01)
for i in range(1000):
    x2 = torch.exp(z)
    loss = objective(x1, x2)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()

    # Print or log the iteration information
    print(f'Iteration {i + 1}/{1000}, x1: {x1.item()}, x2:
{x2.item()}, Loss: {loss.item()}')
```

نتایج این بخش به صـــورت زیر خواهد بود. همانگونه که مشـــاهده می‌شـــود، پس از ۱۰۰۰ ایپاک مقادیر بهینه برای x_1, x_2 برابر با ۱.۲۸۶ و تقریبا صفر بدست آمده‌اند و مقدار کمینه تابع نیز تقریبا برابر با صفر بدست آمده است.

```
Iteration 1/1000, x1: 0.98, x2: 1.0, Loss: 1.0
Iteration 2/1000, x1: 0.96, x2: 1.01, Loss: 0.950843870639801
Iteration 3/1000, x1: 0.94, x2: 1.01, Loss: 0.9049161076545715
                                ⋮
Iteration 300/1000, x1: 0.008, x2: 1.286, Loss: 5.212990799918771e-05
Iteration 301/1000, x1: 0.007, x2: 1.286, Loss: 5.052211054135114e-05
Iteration 302/1000, x1: 0.007, x2: 1.286, Loss: 4.896391328657046e-05
                                ⋮
Iteration 998/1000, x1: 1.44e-07, x2: 1.286, Loss: 1.6627022973508473e-14
Iteration 999/1000, x1: 1.417e-07, x2: 1.286, Loss: 1.6114230921306615e-14
Iteration 1000/1000, x1: 1.395e-07, x2: 1.286, Loss: 1.5617252974229993e-14
```

```
Optimal solution: x1 = 1.3956349675936508e-07, x2 = 1.2868976593017578,
Minimum value: 1.5135601099989274e-14
```

طبقه‌بندی دادگان MNIST

در این بخش می‌خواهیم تا با استفاده از بهینه ساز MyGD که تعریف کردیم و یک شبکه با ۳ لایه تمام متصل و با تابع محاسبه خطای Cross Entropy که برای طبقه‌بندی مجموعه دادگان چند طبقه‌ای مناسب است، مجموعه دادگان MNIST که شامل ۱۰ طبقه عکس اعداد ۰ تا ۹ است را طبقه بندی کنیم. پس از لود کردن این دیتاست و نرمالیزه کردن آن به مقادیر بین ۱- و ۱ و تبدیل آن به تنسور، به تعریف مدل می‌پردازیم. مدل استفاده شده در این بخش یک شبکه خطی با ۳ لایه تمام متصل دارد که ورودی آن به صورت ۲۸*۲۸ که به دلیل عکس‌ها انتخاب شده است و در خروجی ۱۰ نورون قرار دارد که بیانگر ۱۰ طبقه عکس‌ها (هر یک ارقام) می‌باشد.

```
class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
model = MyNet()
optimizer = MyGD(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()
```

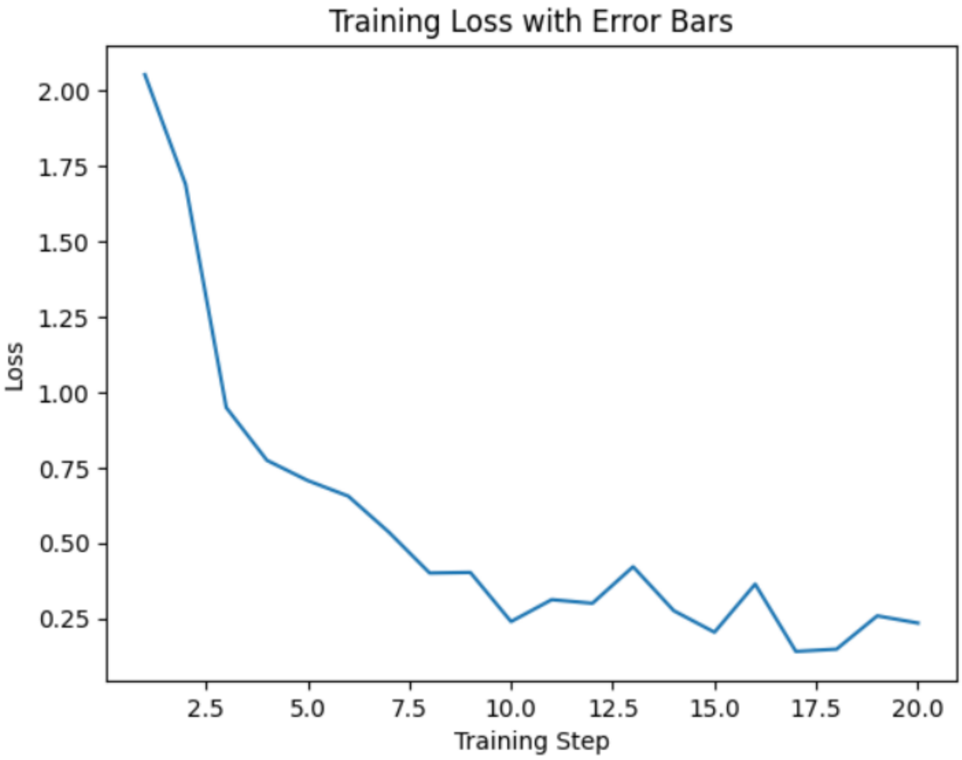
پس از آن به آموزش شبکه با استفاده از بهینه‌ساز تعریف شده و تابع خطای Cross Entropy به صورت پس انتشار خطا می‌پردازیم.

```python
loss_values = []

for epoch in range(20):
    for i, (images, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(images)
        loss = loss_fn(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
    loss_values.append(loss.item())

        # Print the loss
    print(f'Epoch [{epoch + 1}/{20}], Loss: {loss.item()}')
```

همانطور که در ادامه مشاهده می‌کنید، پس از ۲۰ ایپاک به صحت ۹۱ درصد برای دادگان تست دیتاست رسیدیم. در ادامه نتایج آموزش و تغییر خطا را مشاهده می‌نمایید.

```
Epoch [1/20], Loss: 2.053619384765625
Epoch [2/20], Loss: 1.6897128820419312
Epoch [3/20], Loss: 0.949285089969635
                    …
Epoch [18/20], Loss: 0.1475607454776764
Epoch [19/20], Loss: 0.2581514716148376
Epoch [20/20], Loss: 0.2341915369033813
```

```
Test Accuracy: 91.01%
```



شکل ۱: نمودار تابع خطا برای آموزش شبکه با دادگان MNIST با استفاده از بهینه ساز MyGD

سوال ۲

۲- الگوریتم‌های Newton و BFGS را با بهره‌گیری از متد backtracking line search (الگوریتم ۳.۱ کتاب Nocedal) و همچنین الگوریتم trust-region را با بهره‌گیری از متد dogleg پیاده‌سازی نمایید و از آن‌ها برای بهینه‌سازی تابع زیر استفاده کنید. مقدار اولیه‌ی اندازه قدم را $\delta_0 = 1$ انتخاب کنید و مسیری که در هرکدام از الگوریتم‌ها برای رسیدن به نقطه‌ی بهینه طی می‌شود را بر روی کانتورلاین‌های تابع هدف رسم کنید و در نهایت به مقایسه نتایج به‌دست آمده بپردازید. این الگوریتم‌ها را برا نقطه‌های شروع $x_0 = (1.2, 1.2)^T$ و $x_0 = (-1.2,1)^T$ اجرا کنید. همچنین برای روش BFGS از دو حدس اولیه‌ی $B = \frac{\|g_0\|}{\delta} \boldsymbol{I}$ و $B = \frac{y_1^T y_1}{y_1^T s_1} \boldsymbol{I}$ استفاده کنید که در آن g_0 نشان‌دهنده‌ی گرادیان تابع در گام اول است. s_1 و y_1 نیز مطابق رابطه (2.17) از کتاب Nocedal تعریف می‌شوند.

$$f(x_1, x_2) = \log(exp(x_1) + exp(x_2))$$

الگوریتم جستجوی عقبگرد (Backtracking Line Search) طبق کتاب به صورت زیر است:

Algorithm 3.1 (Backtracking Line Search).
Choose $\bar{\alpha} > 0, \rho \in (0,1), c \in (0,1)$; Set $\alpha \leftarrow \bar{\alpha}$;
repeat until $f(x_k + \alpha p_k) \leq f(x_k) + c\alpha \nabla f_k^T p_k$
$\alpha \leftarrow \rho\alpha$;
end (repeat)
Terminate with $\alpha_k = \alpha$.

ابتدا باید این الگوریتم را تعریف نماییم. این الگوریتم تا زمانی بهینه سازی را انجام می‌دهد که شرط اول Wolfe را برآورده کند (البته نه به صورت گام خیلی کوتاه).

```
def backtracking_line_search(f, grad_f, x, p, alpha_bar, rho, c):
    alpha = alpha_bar
    while f(x + alpha * p) > f(x) + c * alpha * grad_f(x).T @ p:
        alpha *= rho
    return alpha
```

همچنین پارامترهای ابتدایی پس از آن بدین شکل تعریف می‌شوند:

```
alpha_bar = 1.0
rho = 0.8
c = 0.1
```

پس از آن به تعریف الگوریتم منطقه اعتماد[1] با روش dogleg برای مدیریت هر دو انحنای مثبت و منفی در فرآیند بهینه‌سازی پیاده‌سازی شد. الگوریتم از روش dogleg برای محاسبه جهت جستجو، با در نظر گرفتن ناحیه اعتماد تعریف شده با دلتای برابر با ۱ استفاده کرد.

این الگوریتم لندازه و جهت گام را بر اساس انحنای تابع هدف تنظیم می‌کند و از همگرایی در منطقه اعتماد اطمینان حاصل می‌کند. با این حال، در موارد خاص، با یک ماتریس منفرد مواجه شد، که نشان دهنده یک ماتریس هسین غیرقابل معکوس پذیر است. در چنین شرایطی، الگوریتم به استفاده از گرادیان منفی به عنوان جهت جستجو بازگشت.

[1] Trust Region

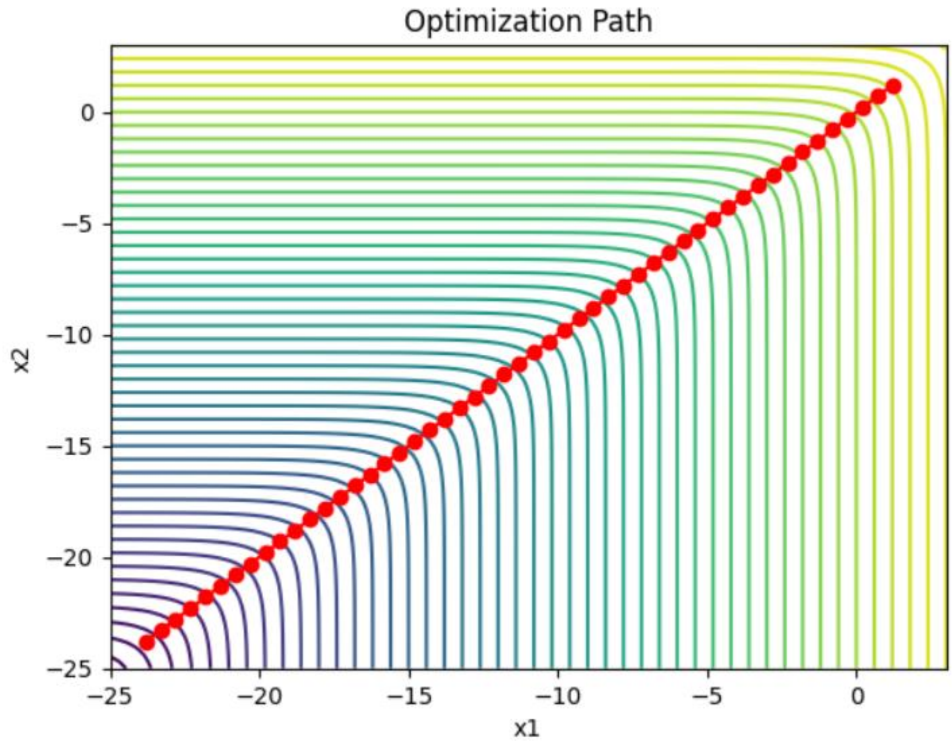
```python
import numpy as np
def dogleg_method(grad_f, hessian_f, delta):
    try:
        hessian_inv = np.linalg.inv(hessian_f)
    except np.linalg.LinAlgError:
        # Hessian is singular, use a different method or fallback
strategy
        return -grad_f

    p_Unc = -hessian_inv @ grad_f
    if np.linalg.norm(p_Unc) <= delta:
        return p_Unc

    p_B = -(grad_f.T @ grad_f) / (grad_f.T @ hessian_f @ grad_f) *
grad_f
    if np.linalg.norm(p_B) >= delta:
        return delta * p_B / np.linalg.norm(p_B)

    p_H = hessian_inv @ -grad_f
    a = np.linalg.norm(p_B - p_Unc) ** 2
    b = 2 * (p_B - p_Unc).T @ (p_Unc - p_H)
    c = np.linalg.norm(p_Unc - p_H) ** 2 - delta ** 2
    tau = (-b + np.sqrt(b ** 2 - 4 * a * c)) / (2 * a)
    return p_Unc + tau * (p_B - p_Unc)
```

سپس به پیاده سازی الگوریتم نیوتون با استفاده از این دو الگوریتم پرداختیم.

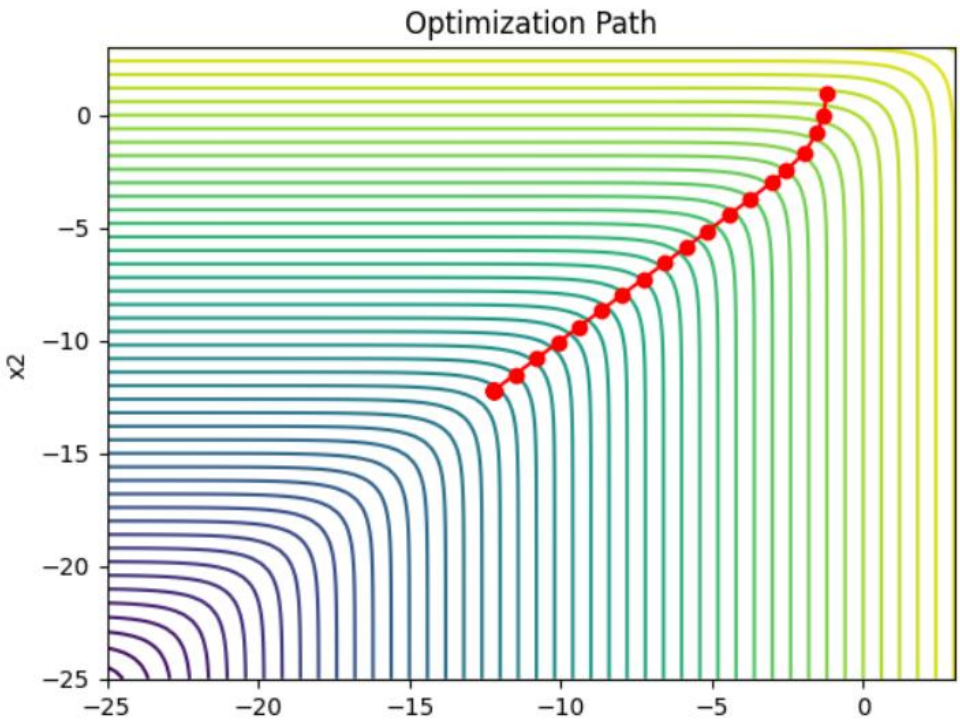
$$\min_{\alpha>0} f(x_k + \alpha p_k).$$

$$p_k^N = -\left(\nabla^2 f_k\right)^{-1} \nabla f_k.$$

پس از تعریف توابعی برای محاسبه گرادیان و هسین (برای تقریب تیلور مرتبه ۲) برای محاسبه آن‌ها برای تابع هدف، الگوریتم نیوتون را برای ۵۰ ایپاک اجرا می‌کنیم. بدین صورت که پس از محاسبه آلفا، جهت حرکت بعدی تعیین می‌شود. شکل ۲ نمایش حرکت به سمت نقطه بهینه با نقطه شروع (1.2,1.2) و شکل ۳ نمایش حرکت به سمت نقطه بهینه برای نقطه شروع (-1.2,1) را نمایش می‌دهد. به این نکته توجه باید داشت که این مسئله یک مسئله unbounded below است.



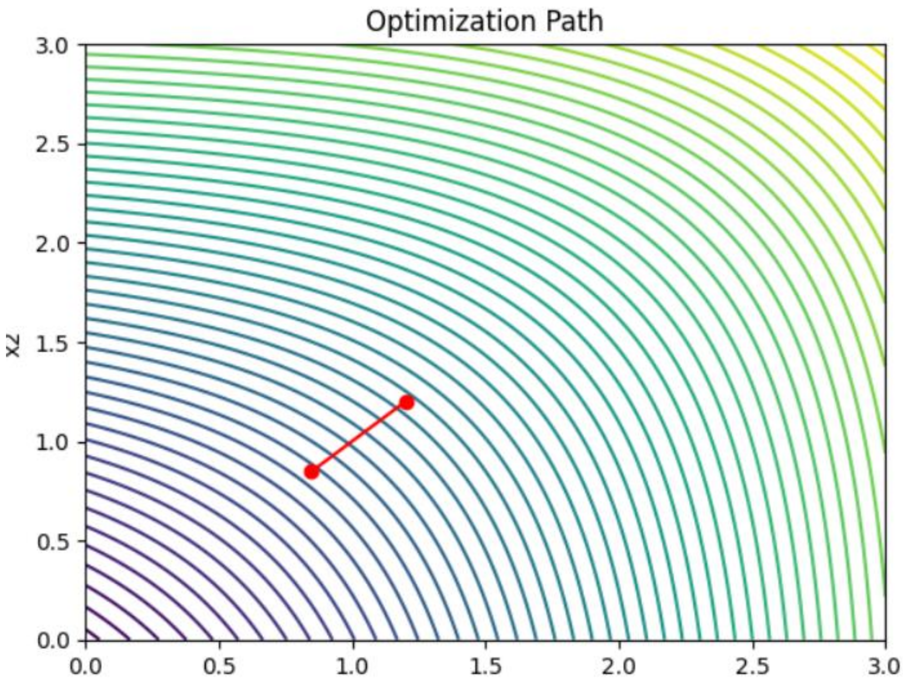
شکل ۲: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (۱.۲,۱.۲)



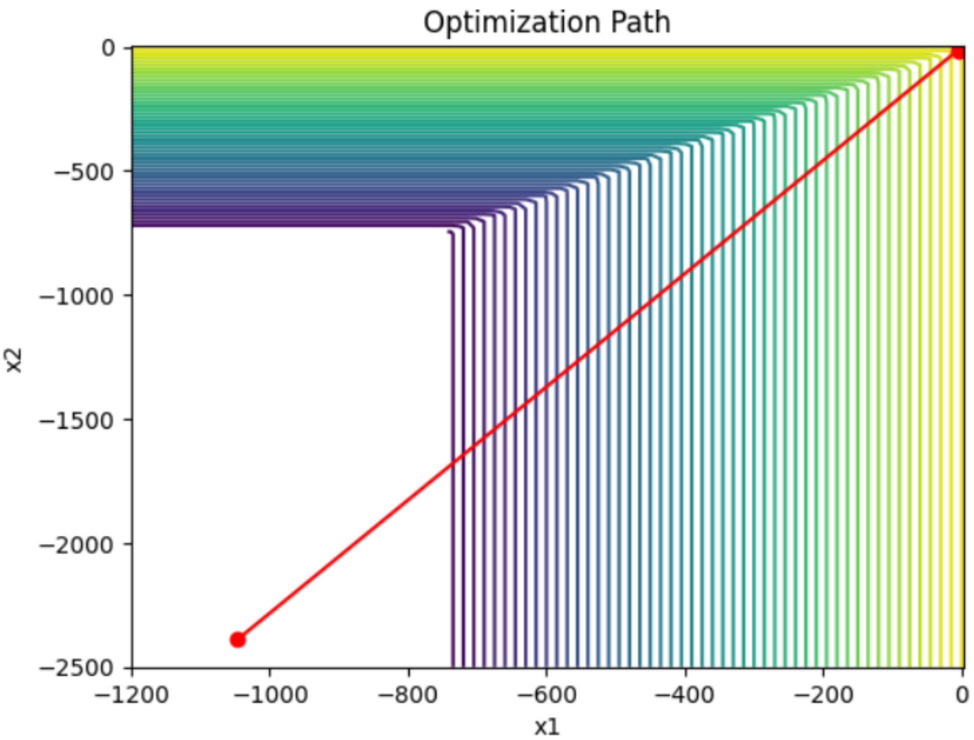
شکل ۳: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (۱.۲,۱-)

الگوریتم BFGS که از الگوریتم‌های شبه نیوتون شمرده می‌شود، شبیه به الگوریتم نیوتون است با این تفاوت که به جای محاسبه مستقیم هسین تابع هدف، از تابع دیگری که B است استفاده می‌شود. پیاده سازی آن نیز به مانند نیوتون است اما به جای استفاده از هسین یک تابع BFGS تعریف می‌کنیم. در ادامه خروجی‌های این بخش را برای هرگام (به دلیل شکل‌های بزرگ) ترسیم نموده‌ایم. مشاهده می‌شود که تفاوت آن با روش نیوتون این است که به صورت مستقیم به سمت نقطه بهینه نمی‌رویم و در مسیر کجی‌هایی نیز خواهیم داشت. در حالی که اگر به خروجی‌های نیوتون و با استناد به کانتورها نگاه کنید، به صورت مستقیم به سمت نقطه بهینه پیش می‌رویم که دقیقتر از خروجی‌های پایینتر است.

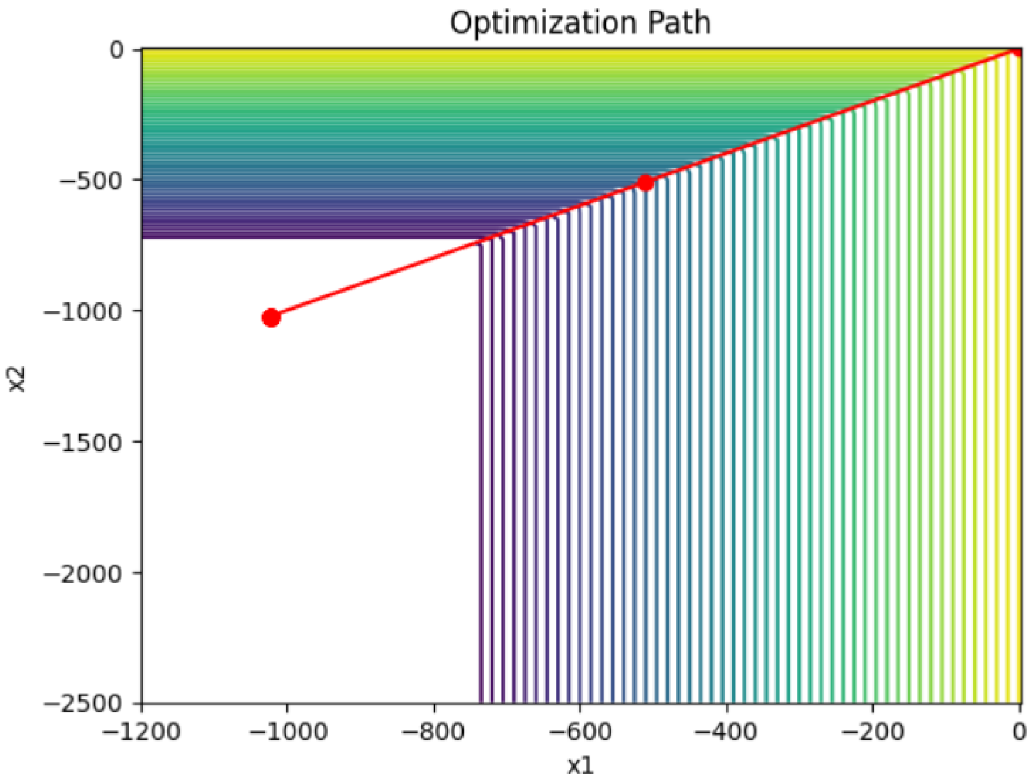
```
def bfgs_optimizer(f, grad_f, x0, alpha_bar, rho, c, delta,
max_iter=100):
    x = x0.copy()
    B_inv = (np.linalg.norm(grad_f(x0)) / delta) * np.eye(len(x0))

    path_bfgs = [x]
    for _ in range(max_iter):
        grad = grad_f(x)
        p = -B_inv @ grad
        alpha = backtracking_line_search(f, grad_f, x, p,
alpha_bar, rho, c)
        x_new = x + alpha * p
        s = x_new - x
        y = grad_f(x_new) - grad
        B_inv = (np.eye(len(x)) - np.outer(s, y) / np.dot(y, s)) @
B_inv @ (
            np.eye(len(x)) - np.outer(y, s) / np.dot(y, s)
        ) + np.outer(s, s) / np.dot(y, s)

        x = x_new
        path_bfgs.append(x)

    return path_bfgs
```



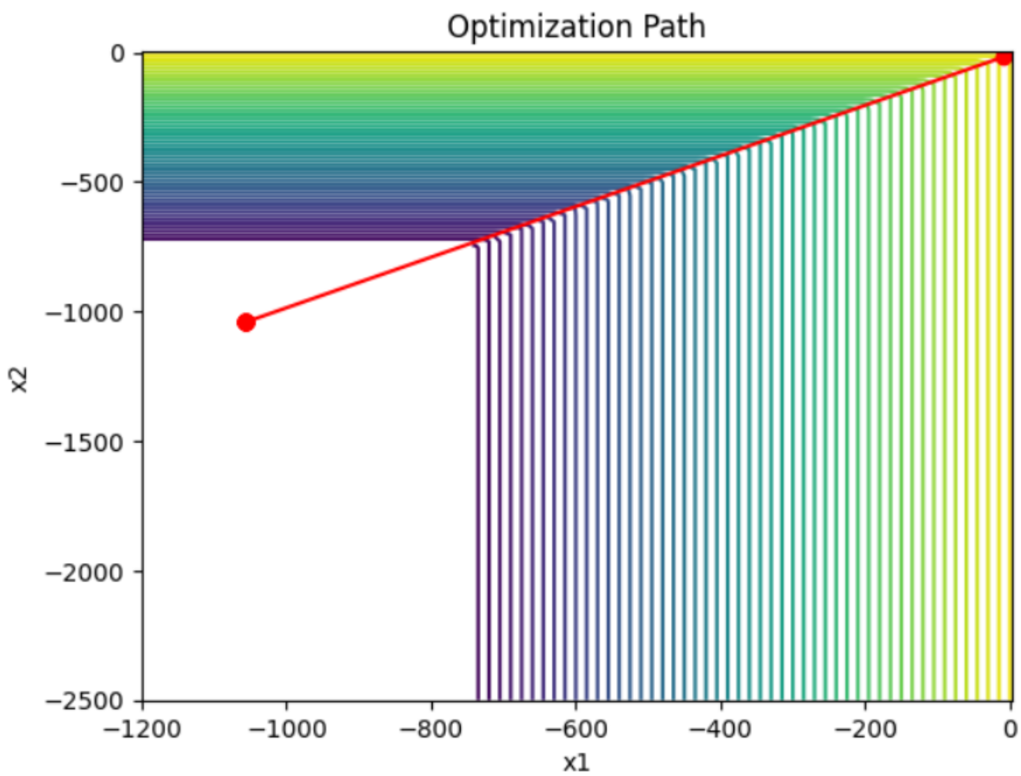
شکل ۴: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (۱.۲,۱.۲) برای $B = \frac{g_0}{\delta}$



شکل ۵: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (۱.۲,۱-) برای $B = \frac{g_0}{\delta}$



شکل ۶: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (۱.۲,۱.۲) برای $B = \frac{y_1^T y_1}{y_1^T s_1}$



شکل ۷: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (-۱.۲,۱) برای $B = \frac{y_1^T y_1}{y_1^T s_1}$