

دانشگاه صنعتي امیرکبیر  
(پلی تکنیک تهران)

تمرین شماره 2 بهینه‌سازی محدب – بخش پیاده سازی

نگارش

امین عبدی­پوراصل

401133011

استاد درس

دکتر امیرمزلقانی

دی 1402

### شما همچنین می‌توانید از طریق [لینک گیت‌هاب](https://github.com/abdipourasl/Conex-Optimization-1402) به کدها به صورت کامل نیز دسترسی داشته باشید.

## سوال‌ 1

هدف این سوال پیاده‌سازی روش کاهش گرادیان[[1]](#footnote-1) (GD) با استفاده ازPyTorch 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 0.001 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 0.001 اصلاح را انجام می‌دهد. 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.

بدین منظور باید یک نقطه شروع برای آن در نظر گرفت و پس از آن تابع هدف[[2]](#footnote-2) که در اینجا 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 1000 ایپاک مقدار تابع و خطا را محاسبه و در خلاف جهت گرادیان با گام 1000 حرکت می‌کنیم و برای هر گام مقدار تابع و خطا را نمایش می‌دهیم.

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 صورت زیر خواهد بود. همانگونه که مشاهده می‌شود، پس از 1000 ایپاک مقادیر بهینه برای برابر با 1.286 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 3 لایه تمام متصل و با تابع محاسبه خطای Cross Entropy که برای طبقه‌بندی مجموعه دادگان چند طبقه‌ای مناسب است، مجموعه دادگان MNIST که شامل 10 طبقه عکس اعداد 0 تا 9 است را طبقه بندی کنیم. پس از لود کردن این دیتاست و نرمالیزه کردن آن به مقادیر بین -1 و 1 و تبدیل آن به تنسور، به تعریف مدل می‌پردازیم. مدل استفاده شده در این بخش یک شبکه خطی با 3 لایه تمام متصل دارد که ورودی آن به صورت 28\*28 که به دلیل عکس‌ها انتخاب شده است و در خروجی 10 نورون قرار دارد که بیانگر 10 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.

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 20 ایپاک به صحت 91 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

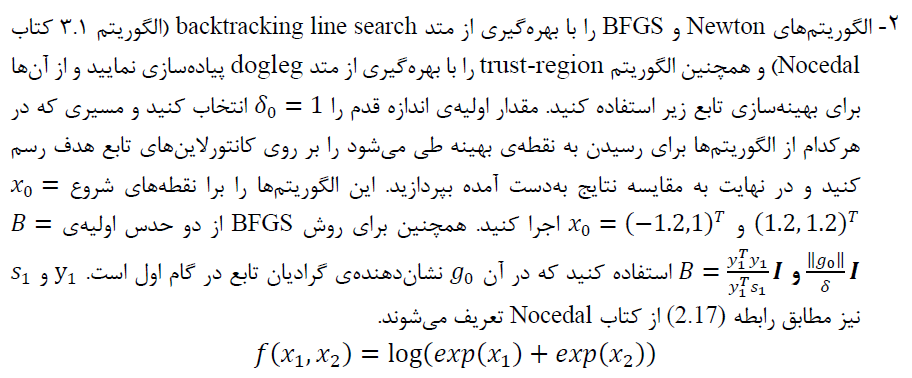
Epoch [20/20], Loss: 0.2341915369033813

Test Accuracy: 91.01%

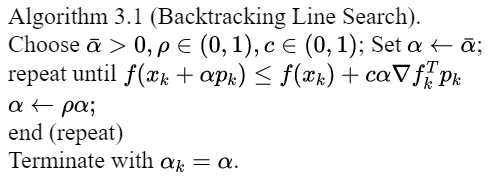


شکل 1: نمودار تابع خطا برای آموزش شبکه با دادگان MNIST با استفاده از بهینه ساز MyGD

### سوال 2



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



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

پسس از آن به تعریف الگوریتم منطقه اعتماد[[3]](#footnote-3) 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 1 استفاده کرد.

این الگوریتم اندازه 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.

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

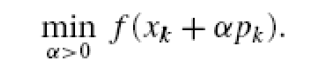
    b = 2 \* (p\_B - p\_Unc).T @ (p\_Unc - p\_H)

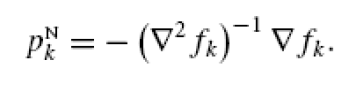
    c = np.linalg.norm(p\_Unc - p\_H) \*\* 2 - delta \*\* 2

    tau = (-b + np.sqrt(b \*\* 2 - 4 \* a \* c)) / (2 \* a)

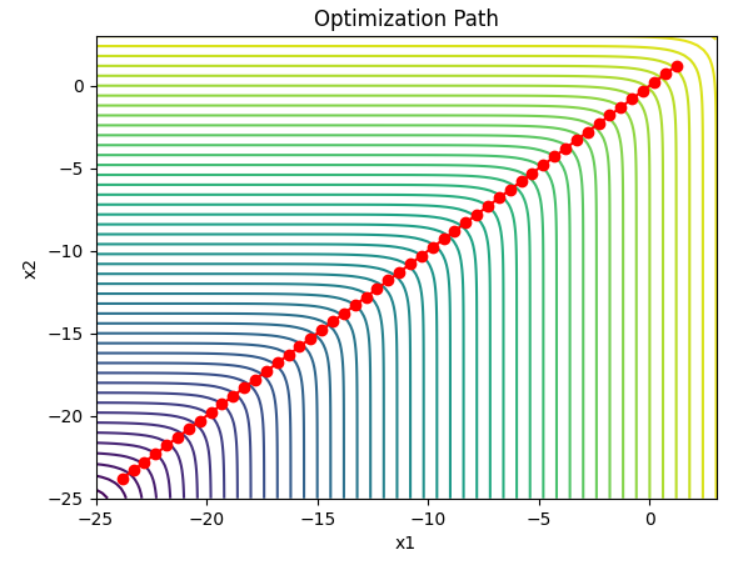
    return p\_Unc + tau \* (p\_B - p\_Unc)

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

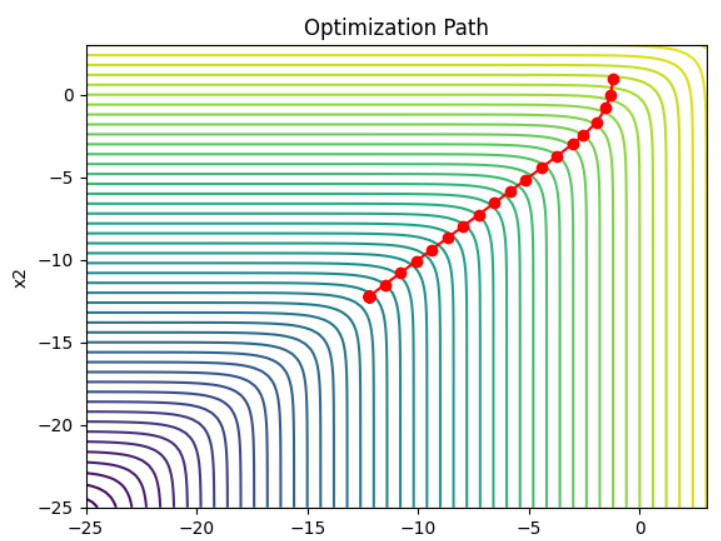




پس از تعریف توابعی برای محاسبه گرادیان و هسین (برای تقریب تیلور مرتبه 2) برای محاسبه آن‌ها برای تابع هدف، الگوریتم نیوتون را برای 50 ایپاک اجرا می‌کنیم. بدین صورت که پس از محاسبه آلفا، جهت حرکت بعدی تعیین می‌شود. شکل 2 نمایش حرکت به سمت نقطه بهینه با نقطه شروع (1.2,1.2) و شکل 3 نمایش حرکت به سمت نقطه 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 2: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (1.2,1.2)



شکل 3: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (-1.2,1)

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 @ (

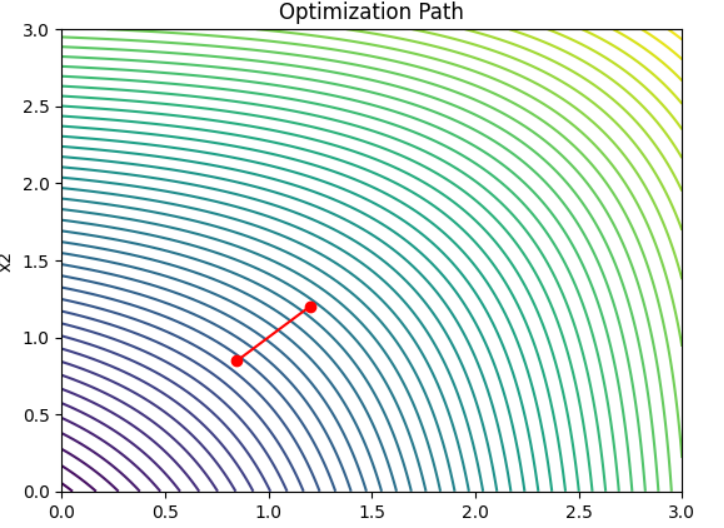
            np.eye(len(x)) - np.outer(y, s) / np.dot(y, s)

        ) + np.outer(s, s) / np.dot(y, s)

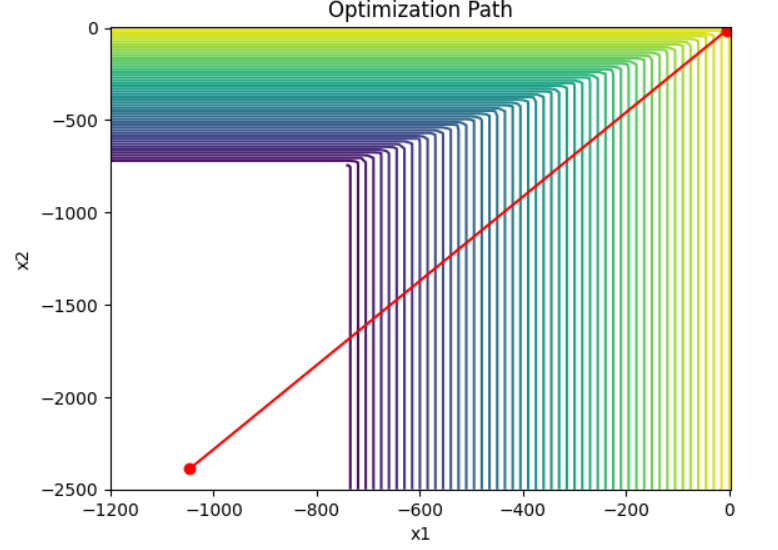
        x = x\_new

        path\_bfgs.append(x)

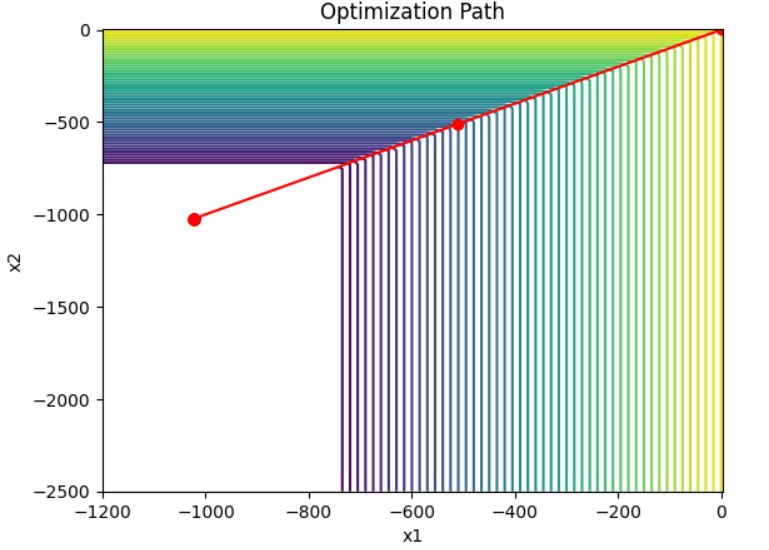
    return path\_bfgs



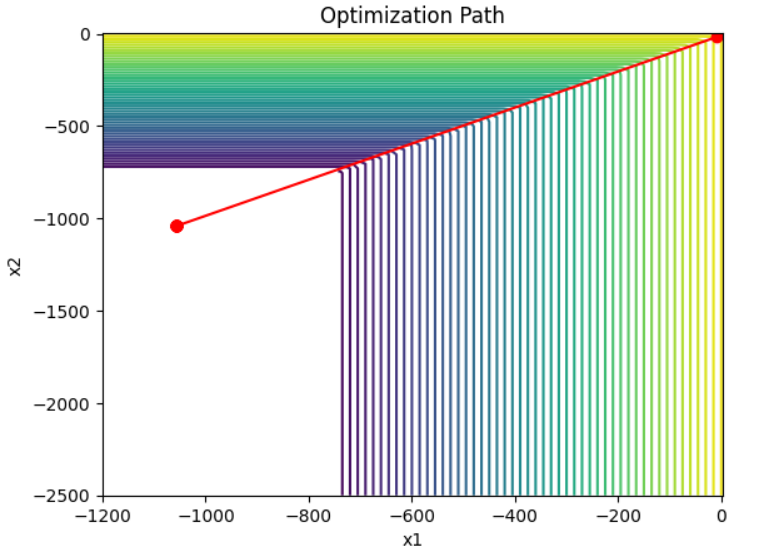
شکل 4: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (1.2,1.2) برای



شکل 5: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (-1.2,1) برای



شکل 6: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (1.2,1.2) برای



شکل 7: نمایش حرکت به سمت نقطه بهینه با نقطه شروع (-1.2,1) برای

1. Gradient Descent [↑](#footnote-ref-1)
2. Objective Function [↑](#footnote-ref-2)
3. Trust Region [↑](#footnote-ref-3)