Momentum 최적화 알고리즘 구현

20191595 박승현

목차

Momentum 알고리즘

개요

모멘텀 알고리즘을 이해하기 전에 먼저 경사 하강법을 이해하여야 했습니다. 결국 모멘텀 알고리즘이 경사 하강법의 종류 중 하나 이기 때문이며 경사 하강법의 한계를 개선하기 위해 나온 알고리즘 이기 때문입니다.

경사 하강법과 모멘텀 등장 배경

결국 최적화를 위한 여러 알고리즘의 공통적인 목표는 함수에서 최솟값 혹은 최댓값을 찾는 것입니다. 이때 사용하는 것이 미분을 통해 구하는 기울기(경사)이고 이를 구하여 경사가 낮은 쪽으로 계속 이동시켜 0인 지점을 찾아 극 값을 구하자는 것이 이 알고리즘의 기본 개념입니다. 경사 하강법이라는 이름이 붙여진 것도 이러한 단계를 거치기 때문입니다.

최적화할 함수 f(x)에 대하여 시작점 x0를 정하고 현재 위치 xi가 주어 졌을 때 다음 위치 x(i+1)은

X(i+1) = xi − h*f‘(xi) 입니다 (h는 이동할 거리를 조절하는 매개변수)

따라서 최소값을 찾아가는 도중에 기울기가 0이면 멈춰버리는 한계가 생깁니다. 또한 값을 찾기 위해 많은 단계를 거쳐야 하는 단점도 있습니다.

그래서 이를 보완하기 위해 나온 알고리즘 중 하나가 이번 프로젝트에서 구현할 모멘텀 알고리즘 입니다.

Momentum

모멘텀은 경사하강법에서 관성이라는 개념을 추가하였습니다.

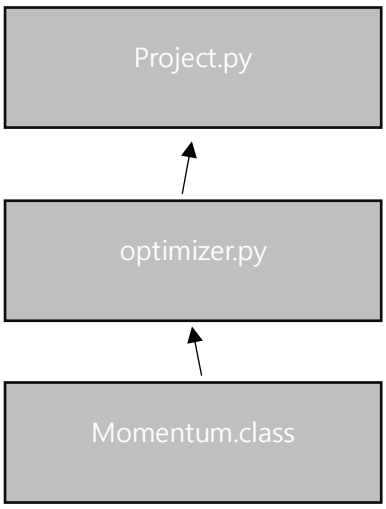
Learning rate = A, momentum = B 와 초기값 x0이라 할 때

a(n+1) = B*a(n) + f‘(xn), a(0) = 0

x(n+1) = x(n) − A*a(n+1)

입니다. 따라서 momentum계수 B가 0이면 위의 경사 하강법과 같은 알고리즘이 됩니다. 하지만 경사 하강법과는 달리 f‘(xn) = 0임에도 a(n)에 의한 관성효과로 xn은 업데이트 되는 것이 momentum 알고리즘입니다.

코드의 구조



Momentum 클래스

앞서 설명 했던 수식을 바탕으로 값을 받아들여 새로운 위치를 업데이트 시켜주는 역할을 합니다.

Optimizer.py

Numpy를 import하여 모멘텀 클래스를 구현합니다. 객체지향적 관점으로 보면 이 파일에 다른 최적화 알고리즘을 구현하여 사용할 수 있을 것이라고 생각하였습니다.

Project.py

최적화를 할 함수를 입력하고 직접 미분 알고리즘의 Learning rate 설정과 처음 위치 등을 설정합니다. 알고리즘을 통해 좌표가 업데이트되는 것을 배열에 저장하고 이를 matplotlib를 이용하여 사용자가 그림으로 파악하기 쉽게 그리는 역할을 합니다.

동작 코드 설명

optimizer.py

전체 코드

```python
import numpy as np


class Momentum:
    #모멘텀

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum * self.v[key] - self.lr * grads[key]
            params[key] += self.v[key]
```

Update 함수를 이용해 점을 이동시킵니다.

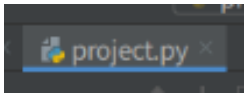
Parmas는 현재 점의 위치 grads는 현재위치의 미분계수 입니다.

a(n+1) = B*a(n) + f`(xn), a(0) = 0

x(n+1) = x(n) – A*a(n+1)

의 식에서 B = self.momentum , A = self.lr

Grads[key] = f`(xn) 입니다.



모듈 import

```python
import sys, os

sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict  # collections 모듈 내 OrderedDict 로드
from optimizer import *  # 직접 제작한 optimizer 모듈 로드
```

초기 설정

```python
init_pos = (-2.0, 2.0)
params = {}
params['x'], params['y'] = init_pos[0], init_pos[1]
grads = {}
grads['x'], grads['y'] = 0, 0

optimizers = OrderedDict()# 여러 가중치를 테스트 하기 위함
optimizers["Momentum"] = Momentum(lr=0.000512)# 최종

idx = 1
```

Init_pos를 통해 처음 시작할 위치를 지정합니다.

Params와 grads에 원함수의x, y값과 도함수의 x, y값을 저장합니다

최적의 매개변수 값들을 찾기위해 OrderedDict를 이용해 한번에 여러 개의 그래프를 그리도록 하였는데 최종 코드에는 하나의 그래프만 그리도록 하였습니다.

값 설정

```python
for key in optimizers:
    optimizer = optimizers[key]
    x_history = []
    y_history = []
    params['x'], params['y'] = init_pos[0], init_pos[1]

    for i in range(50):
        x_history.append(params['x'])
        y_history.append(params['y'])

        grads['x'], grads['y'] = df(params['x'], params['y'])
        optimizer.update(params, grads)

    xx = np.arange(-10, 10, 0.01)
    yy = np.arange(-5, 5, 0.01)
    X, Y = np.meshgrid(xx, yy)
    Z = f(X, Y)
```

```
41      for i in range(50):
42          x_history.append(params['x'])
43          y_history.append(params['y'])
44
45          grads['x'], grads['y'] = df(params['x'], params['y'])
46          optimizer.update(params, grads)
```

이 부분의 코드에서 점의 이동경로를 저장합니다.

반복문의 회수만큼 점을 찍습니다

각 반복문에서 x, y의 좌표를 x_history, y_history에 저장합니다.

45번째 줄의 코드를 통해 현재 좌표를 받아 그 좌표의 도함수에서의 값을 받아와

앞의 모멘텀 알고리즘을 적용시켜 값을 업데이트하는 것을 반복합니다.

함수와 도함수 입력

```
10  def f(x, y):
11      return x**2 / 20.0 + y**2
12
13
14  def df(x, y):
15      return x / 10.0, 2.0*y
16
17  # def f(x, y):
18  #     return (1 - x)**2 + 100.0 * ((y - x**2)**2)
19  # def df(x, y):
20  #     # f의 도함수 (미분)
21  #     return (2.0 * (x - 1) - 400.0 * x * (y - x**2), 200.0 * (y - x**2))
```

함수와 도함수를 입력합니다 위의 코드는 알고리즘을 테스트하기위해 비교적 간단한 함수를 먼저 사용하였으며 밑의 코드가 로젠브록 함수의 코드입니다.

```
53      # 그래프 그리기
54      plt.subplot(1, 1, idx)
55      idx += 1
56      levels = np.logspace(-2, 3, 15)
57      plt.plot(x_history, y_history, 'o-', color="black")
58
59      plt.contourf(X, Y, Z, alpha=0.2, levels=levels)
60      plt.contour(X, Y, Z, colors="gray",
61                  levels=[0.4, 3, 15, 50, 150, 500, 1500, 5000])
62      plt.plot(1, 1, 'ro', markersize=10, markeredgecolor="cornflowerblue")
63      plt.xlim(-4, 4)
64      plt.ylim(-3, 3)
65
66      plt.xticks(np.linspace(-4, 4, 9))
67      plt.yticks(np.linspace(-3, 3, 7))
68      plt.xlabel("$x$")
69      plt.ylabel("$y$")
70      plt.title("2D Rosenbrock func ")
71
72  plt.show()
73  print(x_history[-1],y_history[-1])
```

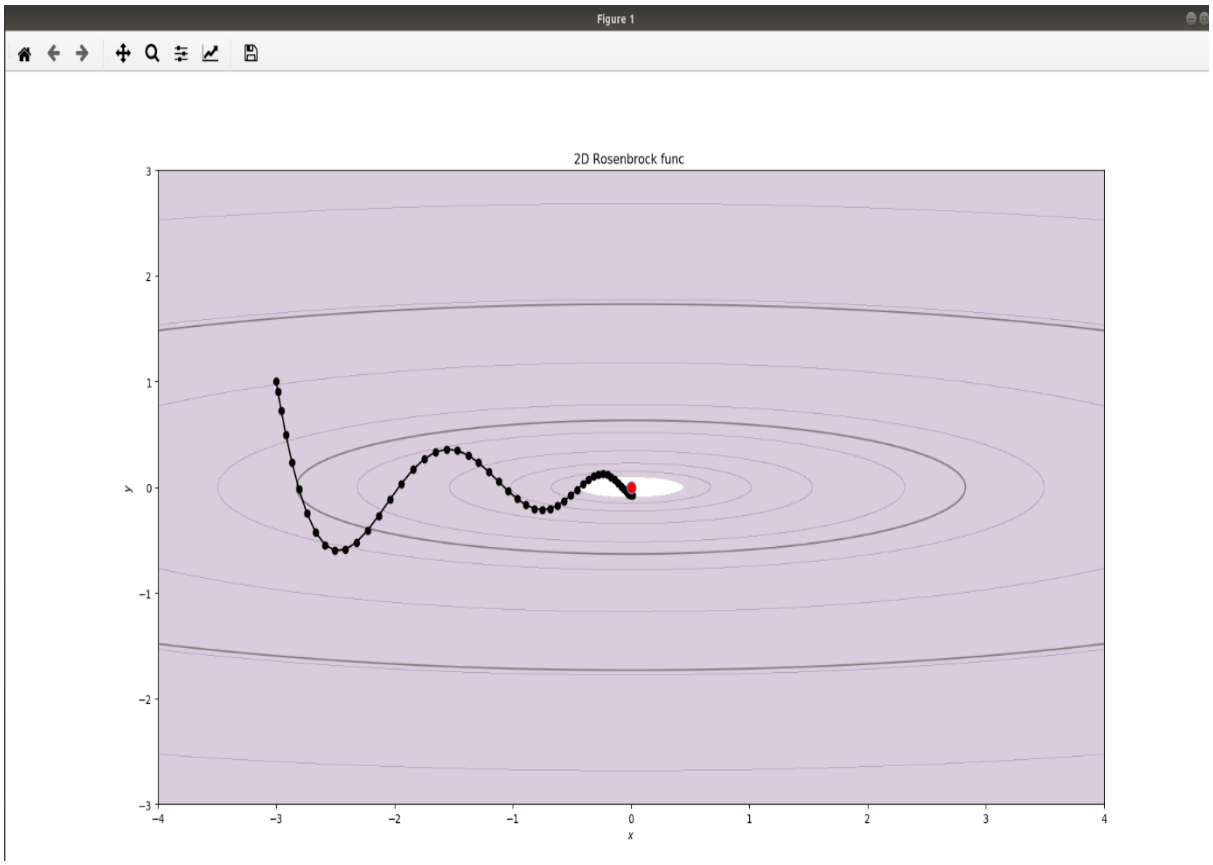
Matplotlib을 이용해 그래프를 그리는 코드입니다. 57번째 줄이 점의 이동경로를 그리는 코드이고, 나머지는 로젠브록 함수를 그리는 코드입니다 62번째줄은 원하는 값의 위치에 점을 찍어주는 코드입니다.

마지막에 print를 통해 점의 최종위치를 출력하여 원하는 값과 어느정도 일치하는지 비교할 수 있게 하였습니다.

코드 실행을 통한 알고리즘 검증

```
10      def f(x, y):
11          return x**2 / 20.0 + y**2
12
13
14      def df(x, y):
15          return x / 10.0, 2.0*y
```

먼저 위의 함수를 가지고 실행해보았습니다. 이 함수는 0,0에서 최적의 값을 가집니다.



이런 결과가 나왔습니다.

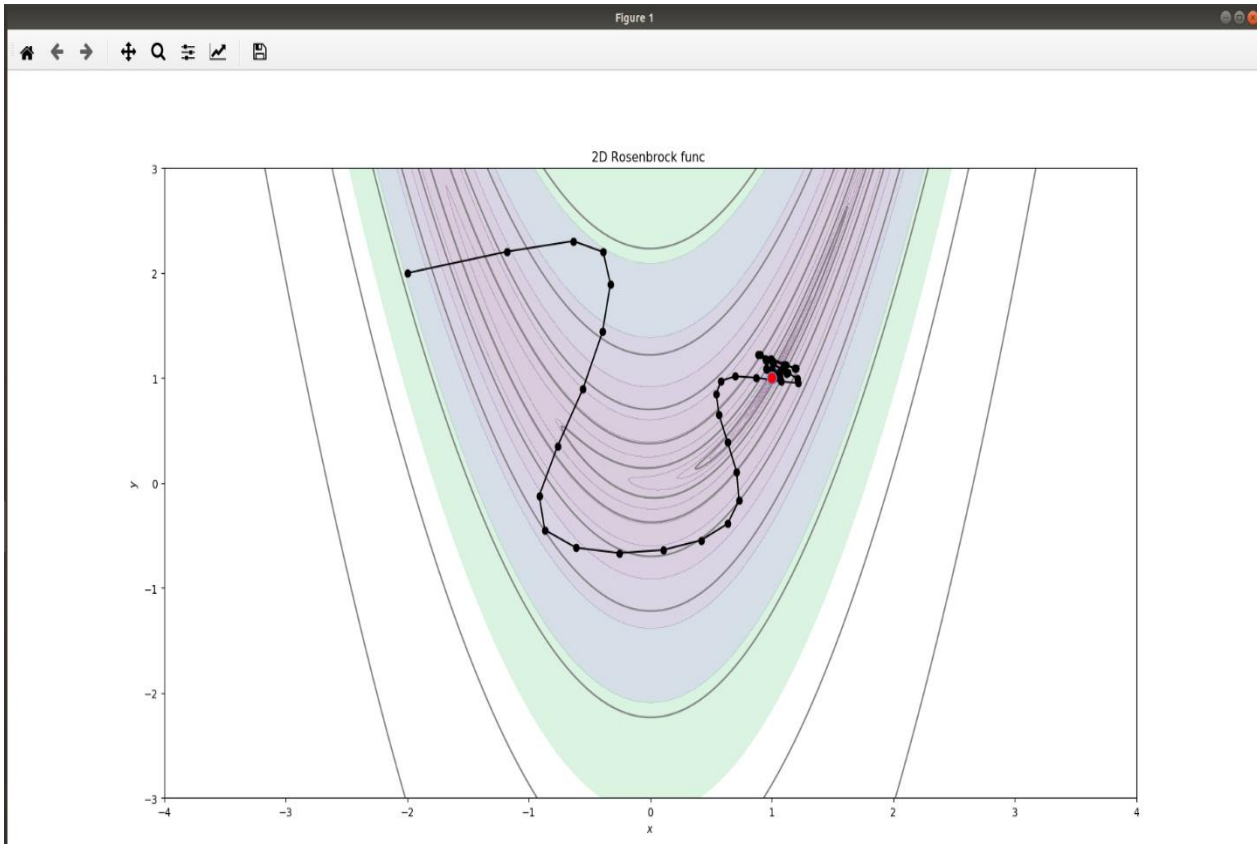
```
/home/seunghyun/anaconda3/bin/python3 /home/seunghyun/PycharmProjects/ML/project.py
Attribute Qt::AA_EnableHighDpiScaling must be set before QCoreApplication is created.
0.014799119794440055 -0.07554041185671469

Process finished with exit code 0
```

최종 위치도 0,0으로 수렴하였습니다.

```
17      def f(x, y):
18          return (1 - x)**2 + 100.0 * ((y - x**2)**2)
19      def df(x, y):
20
21          return (2.0 * (x - 1) - 400.0 * x * (y - x**2), 200.0 * (y - x**2))
22
```

프로젝트의 원래 목표였던 로젠브록 함수로 실행해보겠습니다.

로젠브록 함수는 1,1에서 최솟값을 가집니다.



이동 경로입니다.

```
Run:  project ×
/home/seunghyun/anaconda3/bin/python3 /home/seunghyun/PycharmProjects/ML/project.py
Attribute Qt::AA_EnableHighDpiScaling must be set before QCoreApplication is created.
1.066968872884337 0.992181614594781

Process finished with exit code 0
```

(1, 1)에 수렴시키는데 성공하였습니다.

느낀 점

“프로젝트” 라는 단어를 들을 때 마다 저는 무언가 끓어오르는 것이 있으면서도 내가 잘 할 수 있을까 하는 마음이 생기곤 했습니다. 사실 수치해석이라는 과목을 들으면서 내용도 어렵고 이해도 힘들어서 걱정되는 마음이 컸던 것 같습니다. 하지만 프로젝트를 진행하면 진행할수록 수치해석이라는 분야에 흥미가 생겼고 점점 성장하는 것 같아 뿌듯하기도 했습니다. 사실 결과물이 완벽한 알고리즘이 아닐 수 있지만 처음 목표했던 것보다 훨씬 만족스러운 결과를 얻은 것 같아 뿌듯한 프로젝트가 되었던 것 같습니다. 또한 저의 미래 진로인 게임 프로그래머가 많이 사용한다는 인공지능과 그래픽과 관련한 내용을 입문하는데 큰 발판이 되었던 프로젝트였던 것 같습니다.

그리고 프로젝트를 진행하면서 고등학교 때 했었던 미분이 생각이 났었습니다. 그 당시에는 그다지 많이 복잡하지 않은 함수를 극값과 기울기를 구하기 위해 여러 번 미분하고 수많은 계산하고 그래프를 그리기 위해 많은 시간을 보내면서 고생하였던 것이 생각나면서 내가 지금 배우는 것이 정말 실용적으로 사용될 수 있고 시간을 절약하면서 정확도를 높이는 의미있는 활동이겠구나 라는 생각도 하게 되면서 큰 자부심과 동기부여가 되었습니다.

아주 복잡하고 어려운 함수가 1초도 되지 않는 시간에 그래프가 그려지고 점이 이동하여 최적화가 되는 과정을 보면서 최근 머신러닝, 딥러닝, 인공지능의 중요성이 강조되는 이유가 무엇인지 몸소 체감하기도 하였습니다. 이 느낀 점을 토대로 앞으로 더욱 열심히 공부하여 정진하는 사람이 되어야겠다, 라고 다짐하기도 하였습니다.

학문적인 부분 외에도 처음엔 솔직하게 그냥 포기해버릴까 라는 생각도 꽤 하였습니다. 코드를 짠 후 부분마음으로 실행을 해보았지만 점이 원하는 대로 이동하지 않고 그저 직선으로만 이동하였기 때문입니다. 하지만 천천히 이론을 다시 읽어보았고 코드도 다시 살펴보니 문제점이 무엇인지 눈에 보이기 시작하였고 차근차근 고쳐 나갈 때마다 점들도 완벽하진 않지만 원하는 모습으로 바뀌어 가고 있었습니다. 그 과정에서 어떤 성취감에 의한 희열을 느꼈고 마침내 프로젝트를 완성하였을 때 너무 기쁘고 뿌듯하였습니다. 이 감정은 정말 어느 분야에서나 쓸 수 있고 필요한 포기하지 않는 끈기와 끝까지 할 수 있게 만들어주는 의지를 저에게 심어 주었습니다.