به نام خدا

درس: یادگیری عمیق

استاد: دکتر فاطمی‌زاده

# گزارش تمرین عملی شماره ۲

سیّدمحمّدامین منصوری طهرانی

۹۴۱۰۵۱۷۴

## Comparison of Optimization Methods

توجه: از آن‌جایی که بعضاً تصاویر این فایل خوانا نیستند یا کوچک هستند، تمامی آن‌ها پیوست شده‌اند و شماره شکل‌ها به ترتیبی است که در گزارش می‌آید.(فقط نمودارها. تصاویر قطعه کدها چون لازم نبوده پیوست نشده‌است!)

۱. کد توابع فوق هر کدام در یک فایل جداگانه py. نوشته و ذخیره شده است. نام هر فایل تابع آن کد را مشخص می‌کند.

۲. در ادامه گرادیان توابع داده شده محاسبه شده‌است.

Rastrigin:

$$\nabla f = (2\,(x + 10\,\pi\,sin(2\,\pi\,x)), 2\,(y + 10\,\pi\,sin(2\,\pi\,y)))$$

Ackley:

$$\nabla f = \left(\frac{2.82843\,x\,e^{-0.141421\sqrt{(x^2+y^2)}}}{\sqrt{(x^2+y^2)}} + 2\,\pi\sin(2\,\pi\,x)\,e^{\cos(2\,\pi\,x)+\cos(2\,\pi\,y)}, \frac{2.82843\,y\,e^{-0.141421\sqrt{(x^2+y^2)}}}{\sqrt{(x^2+y^2)}} + 2\,\pi\sin(2\,\pi\,y)\,e^{\cos(2\,\pi\,x)+\cos(2\,\pi\,y)}\right)$$

Levi:

$$\nabla f = \big(3\,(x + \pi\sin(6\,\pi\,x) - 1) - (x - 1)\cos(6\,\pi\,y), 3\,\pi(x - 1)^2\sin(6\,\pi\,y) + 2\,\pi(y - 1)^2\sin(4\,\pi\,y) - (y - 1)(\cos(4\,\pi\,y) - 3)\big)$$

Bukin:

$$f(x,y) = 100\sqrt{|y-0.01x^2|} + 0.01|x+10|$$

$$= \begin{cases} 100\sqrt{y-0.01x^2} + 0.01(x+10) & x > -10 \ and \ y > \dfrac{x^2}{100} \\ 100\sqrt{0.01x^2 - y} + 0.01(x+10) & x > -10 \ and \ y < \dfrac{x^2}{100} \\ 100\sqrt{y-0.01x^2} - 0.01(x+10) & x < -10 \ and \ y > \dfrac{x^2}{100} \\ 100\sqrt{0.01x^2 - y} - 0.01(x+10) & x < -10 \ and \ y < \dfrac{x^2}{100} \end{cases}$$

$$\nabla f(x,y)$$

$$= \begin{cases} \left(0.01 - \dfrac{x}{\sqrt{(y - 0.01\,x^2)}}, \dfrac{50}{\sqrt{(y - 0.01\,x^2)}}\right) & x > -10 \ and \ y > \dfrac{x^2}{100} \\ \left(0.01 + \dfrac{x}{\sqrt{(-y + 0.01\,x^2)}}, -\dfrac{50}{\sqrt{(-y + 0.01\,x^2)}}\right) & x > -10 \ and \ y < \dfrac{x^2}{100} \\ \left(-0.01 - \dfrac{x}{\sqrt{(y - 0.01\,x^2)}}, \dfrac{50}{\sqrt{(y - 0.01\,x^2)}}\right) & x < -10 \ and \ y > \dfrac{x^2}{100} \\ \left(-0.01 + \dfrac{x}{\sqrt{(-y + 0.01\,x^2)}}, -\dfrac{50}{\sqrt{(-y + 0.01\,x^2)}}\right) & x < -10 \ and \ y < \dfrac{x^2}{100} \end{cases}$$

N-D Rastrigin

$$\nabla f_{x_i} = \left(2\left(x_i + 10\,\pi\,sin(2\,\pi\,x_i)\right)\right)\widehat{x_i}$$

۳. بر اساس روابط بالا کد محاسبه گرادیان برای هر تابع نیز در همان فایل py. مربوط به تابع خودش نوشته شد. به این ترتیب در هر کدام از فایل‌ها دو تابع وجود دارد که یکی مقدار تابع و دیگری گرادیان آن را در هر نقطه محاسبه کرده و باز می‌گرداند.

۴. با توجه به این‌که این روش‌ها در کلاس درس تدریس شده‌اند از آوردن روابط آن‌ها خودداری می‌کنیم. برای هر کدام به توضیحی مختصر اکتفا می‌کنیم.

Gradient Descent:

این همان روش اولیه‌ای است که از قبل از آن اطلاع داشتیم.

```python
# Hyper parameter: just learning rate (eta)

# noinspection PyUnusedLocal
def gradient_descent(target_function, coordinates, gradient_of_target_function, optimization_hyper_parameters):

    value = []
    counter = 0

    learning_rate_gd = optimization_hyper_parameters

    gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                 gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2)

    while gradient_magnitude > 1e-4 and counter < 1e5:

        [x_new, y_new] = list(np.array(coordinates) - learning_rate_gd *
                              gradient_of_target_function(coordinates[0], coordinates[1]))

        coordinates = [x_new, y_new]

        gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                     gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2)

        value.append(target_function(x_new, y_new))
        counter += 1

    return value
```

حلقه آپدیت کردن تنها در حالتی که هم تعداد دفعات آپدیت کمتر از عددی باشد و هم اندازه گرادیان بیشتر از عددی باشد وارد حلقه می‌شود و در غیر این صورت یکی از شرط‌های توقف رخ داده و به پایان عملیات بهینه‌سازی رسیده‌ایم. مقدار value که خروجی تابع است، مقدار تابع پس از هر iteration را در یک لیست ذخیره می‌کند که برای رسم نمودار سرعت همگرایی و مقایسه سرعت استفاده می‌شود. ضمناً فقط یک هایپرپارامتر دارد و آن نرخ یادگیری است.

Nesterov:

```python
# Hyper parameters: learning rate (eta) and momentum (gamma)

def nesterov(target_function, coordinates, gradient_of_target_function, optimization_hyper_parameters):

    value = []
    counter = 0

    learning_rate_nesterov = optimization_hyper_parameters[0]
    momentum_coeff = optimization_hyper_parameters[1]
    gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                 gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)
    # First Iteration of this method has to be done using regular GD since we do not have any previous iteration.
    [x_new, y_new] = list(np.array(coordinates) - learning_rate_nesterov *
                          gradient_of_target_function(coordinates[0], coordinates[1]))
    coordinates = [x_new, y_new]
    last_iteration_gradient = gradient_of_target_function(coordinates[0], coordinates[1])

    while counter < 1e4 and gradient_magnitude > 1e-4:

        update_term = momentum_coeff * last_iteration_gradient + \
                      learning_rate_nesterov * gradient_of_target_function(coordinates[0] -
                                                                           momentum_coeff * last_iteration_gradient[0],
                                                                           coordinates[1] - momentum_coeff *
                                                                           last_iteration_gradient[1])
        [x_new, y_new] = list(np.array(coordinates) - update_term)
        coordinates = [x_new, y_new]
        last_iteration_gradient = update_term

        gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                     gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)

        value.append(target_function(x_new, y_new))
        counter += 1

    return value
```

در این روش momentum هم اضافه می‌شود تا رفتار GD را تعدیل کند. مثل توپی که که در دره‌ای می‌افتد. می‌خواهیم کور عمل نکنیم و سرعت را در نزدیکی مینیمم کم کنیم تا دوباره زیاد بالا نرویم.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

که در رابطه فوق تتاها همان نقاط هستند که هربار تغییر می‌کنند و $v_t$ ها جملات آپدیت‌کننده هستند. دو پارامتر گاما و نرخ یادگیری در این روش وجود دارد که به تابع داده می‌شود. در ابتدای کد این تابع نیز این دو مقدار از ورودی تابع برداشته شده‌اند. سپس از آن‌جایی که در مرحله اول از $v_{t-1}$ نمی‌توان استفاده کرد (در دسترس نیست) به روش GD نقطه اول را آپدیت می‌کنیم. پس از آن وارد حلقه تکرارمان می‌شویم که مثل قبل با دو شرط به صورت همزمان وارد حلقه می‌شود. سپس جمله آپدیت مطابق روابط بالا محاسبه شده و نقاط جدید محاسبه می‌شوند. بعد از آن $v_{t-1}$ برای مرحله بعدی set می‌شود. گرادیان در نقطه فعلی محاسبه شده تا برای حلقه بعد شرط چک شود. خروجی value نیز مثل قبل برای مقایسه سرعت‌ها return شده‌است.

RMSProp:

```python
# Hyper parameters: learning rate (eta) and epsilon (gamma)

def rms_prop(target_function, coordinates, gradient_of_target_function, optimization_hyper_parameters):

    value = []
    counter = 0

    learning_rate_rms_prop = optimization_hyper_parameters[0]
    epsilon = optimization_hyper_parameters[1]
    gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                 gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)

    # First Iteration of this method has to be done using regular GD since we do not have any previous iteration.
    [x_new, y_new] = list(np.array(coordinates) - learning_rate_rms_prop *
                          gradient_of_target_function(coordinates[0], coordinates[1]))

    coordinates = [x_new, y_new]
    decaying_average_of_squared_gradients = 0
    n = 0
    decaying_average_of_squared_gradients += gradient_magnitude ** 2
    n += 1

    while counter < 1e4 and gradient_magnitude > 1e-4:
        # print(gradient_of_target_function(coordinates[0], coordinates[1]))
        n += 1
        average_of_squared_gradients = (0.9 * decaying_average_of_squared_gradients + 0.1 * gradient_magnitude ** 2) / n
        update_term = (learning_rate_rms_prop / np.sqrt(average_of_squared_gradients + epsilon)) * \
                      gradient_of_target_function(coordinates[0], coordinates[1])
        [x_new, y_new] = list(np.array(coordinates) - update_term)
        coordinates = [x_new, y_new]
        gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                     gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)

        value.append(target_function(x_new, y_new))
        counter += 1

    return value
```

این روش یک نوع adaptive learning rate را پیاده‌سازی می‌کند تا اگر برای پارامترهای مختلف نیازهای مختلفی داشته باشیم بتوانیم برآورده کنیم. رابطه آن به صورت زیر است:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

باز هم حلقه اول با توجه به روابط با GD معمولی انجام می‌شود. در متغیری نیز این متوسط مجذور گرادیان ذخیره می‌شود. در ادامه هم رابطه فوق پیاده‌سازی شده و بقیه دستورات مثل قبل است.

Adam:

```
learning_rate_adam = optimization_hyper_parameters[0]
beta_1 = optimization_hyper_parameters[1]
beta_2 = optimization_hyper_parameters[2]
epsilon = optimization_hyper_parameters[3]
gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                             gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)
# First Iteration of this method has to be done using regular GD since we do not have any previous iteration.
[x_new, y_new] = list(np.array(coordinates) - learning_rate_adam *
                      gradient_of_target_function(coordinates[0], coordinates[1]))
coordinates = [x_new, y_new]
decaying_average_of_squared_gradients = 0
decaying_average_of_gradients = 0
n = 0
decaying_average_of_squared_gradients += gradient_magnitude ** 2
decaying_average_of_gradients += gradient_of_target_function(coordinates[0], coordinates[1])
n += 1

while counter < 1e4 and gradient_magnitude > 1e-4:
    n += 1
    average_of_squared_gradients = (beta_2 * decaying_average_of_squared_gradients + (1 - beta_2) *
                                    gradient_magnitude ** 2) / n
    average_of_squared_gradients_bias_corrected = average_of_squared_gradients / (1 - beta_2 ** n)
    average_of_gradients = (beta_1 * decaying_average_of_gradients + (1 - beta_1) *
                            gradient_of_target_function(coordinates[0], coordinates[1])) / n
    average_of_gradients_bias_corrected = average_of_gradients / (1 - beta_1 ** n)

    update_term = (learning_rate_adam / (np.sqrt(average_of_squared_gradients_bias_corrected) + epsilon)) * \
                  average_of_gradients_bias_corrected

    [x_new, y_new] = list(np.array(coordinates) - update_term)
    coordinates = [x_new, y_new]

    gradient_magnitude = np.sqrt(gradient_of_target_function(coordinates[0], coordinates[1])[0] ** 2 +
                                 gradient_of_target_function(coordinates[0], coordinates[1])[1] ** 2)

    value.append(target_function(x_new, y_new))
    counter += 1

return value
```

به کد قبلی شباهت بسیاری دارد با این تفاوت که میانگین گرادیان‌ها نیز استفاده و ذخیره می‌شود و با توجه به روابط زیر تغییراتی نسبت به کد قبل دارد که اعمال شده است.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

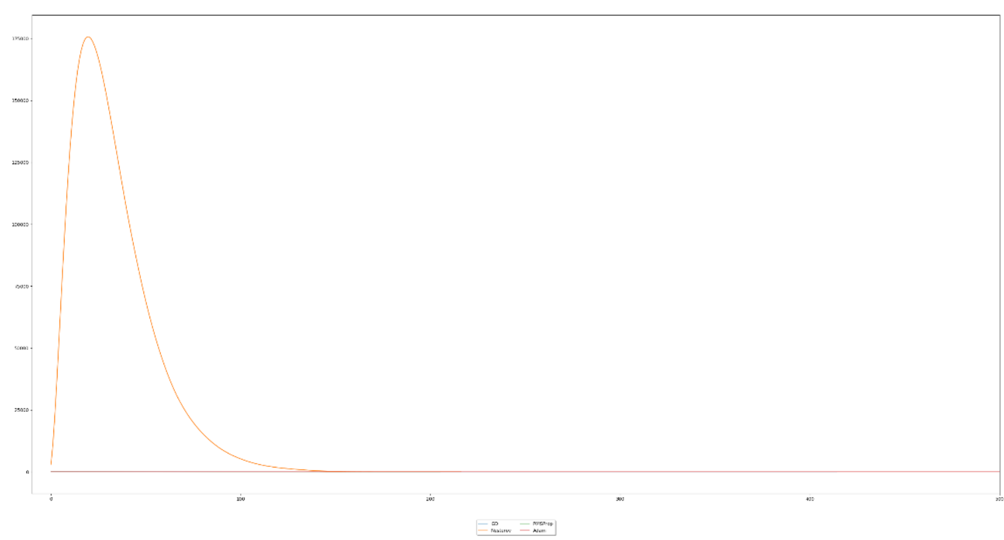
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

۵. توجه می‌کنیم که برای همگرایی و یافتن نقطه مینیمم باید در نواحی خاصی گشت (Test Functions) و در آدرس فوق محدوده شروع x,y داده شده است. ما هم از همین بازه‌ها استفاده می‌کنیم. اعداد تصادفی در این بازه‌ها تولید می‌کنیم و برای هر دسته توابعی که مقایسه سرعت همگرایی آن‌ها مد نظر است از این نقطه تصادفی تولید شده استفاده می‌کنیم. در صورتی که خواستید کد را اجرا کنید و نمودارها را ببینید کد مربوط به سایر بخش‌ها را کامنت کنید تا سرعت اجرا کاهش نیابد و دچار مشکل نشوید و سپس کد را اجرا کنید.

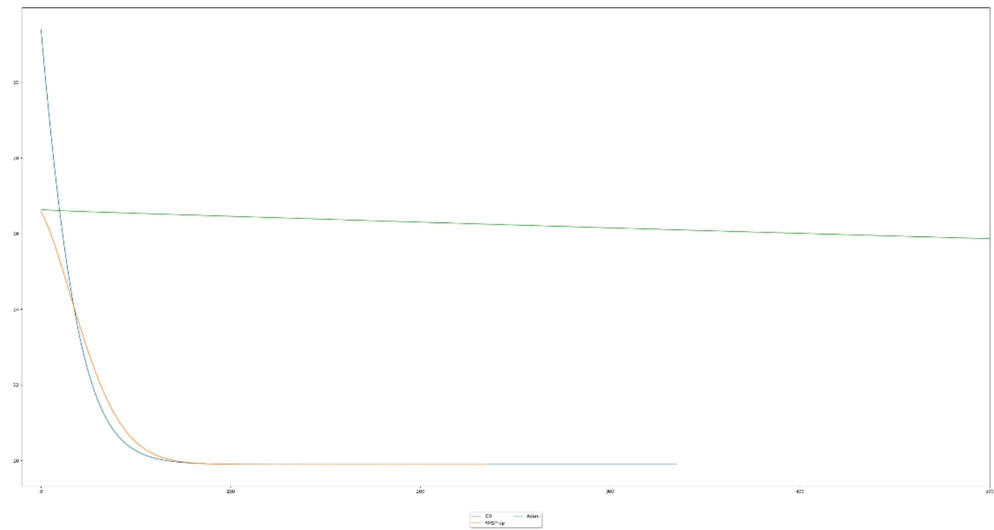
## Rastrigin:

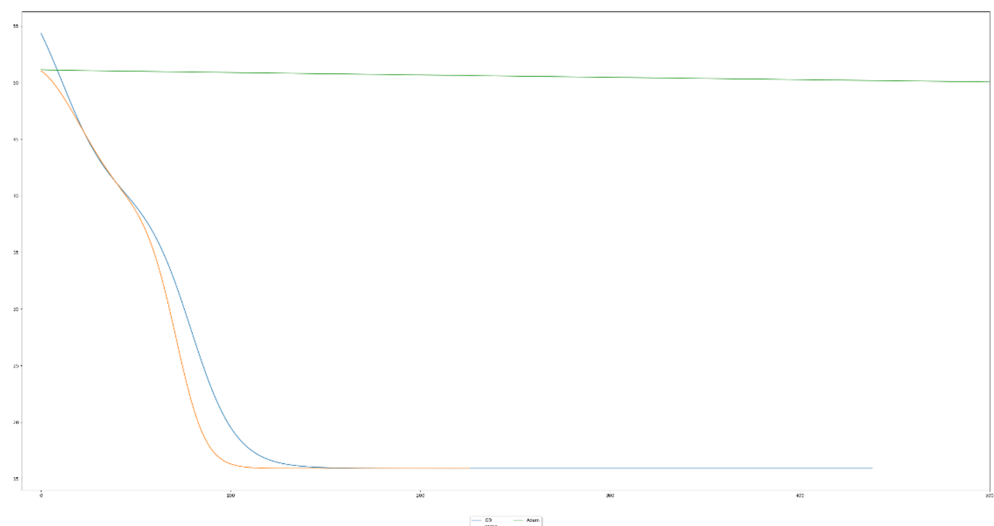
روش بهینه‌سازی Nesterov به دلیلی که متوجه آن نشدم، ابتدا به مقادیر زیاد واگرا شده و سپس همگرا می‌شود و در کنار سایر روش‌های بهینه‌سازی نموداری مطابق شکل زیر بدست می‌دهد که برای مقایسه مناسب نمی‌باشد و لذا نمودار آن به صورت جداگانه نیز آورده می‌شود.




## Nesterov Excluded:

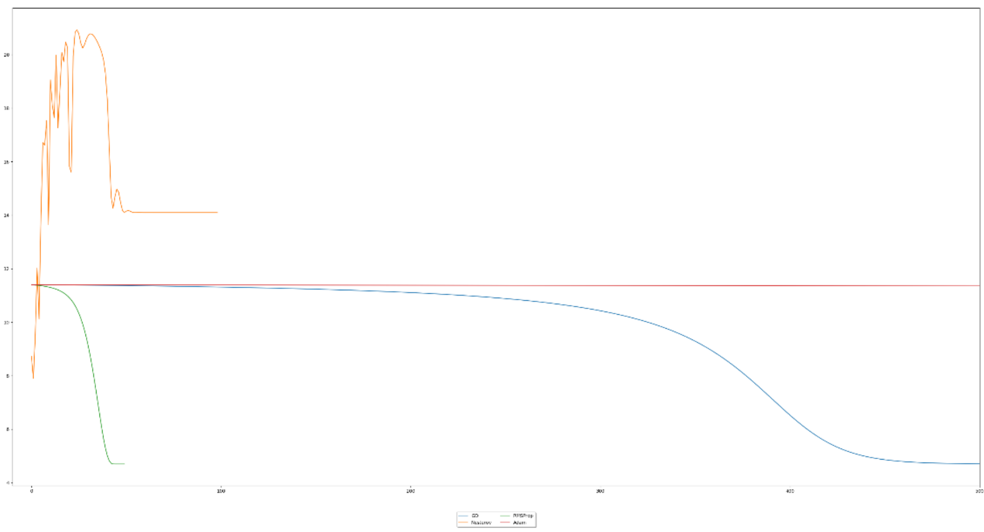
در صفحه بعد نتیجه به ازای دو run مختلف که اعداد مختلفی تولید می‌کند نشان داده شده‌است. به نظر می‌رسد RMSProp سرعت همگرایی بیشتری داشته باشد (با توجه به در نظر گرفتن ویژگی‌ای شبیه به اصطکاک برای حرکت به سمت مینیمم) و پس از آن GD و بعد از آن Adam و در نهایت روش Nesterov. اما احتمالاً در شرایط سخت‌تر GD مشکلات زیادی در همگرایی داشته باشد و سایر روش‌ها همگرایی را بهتر ولی در زمان طولانی‌تر تضمین کنند.

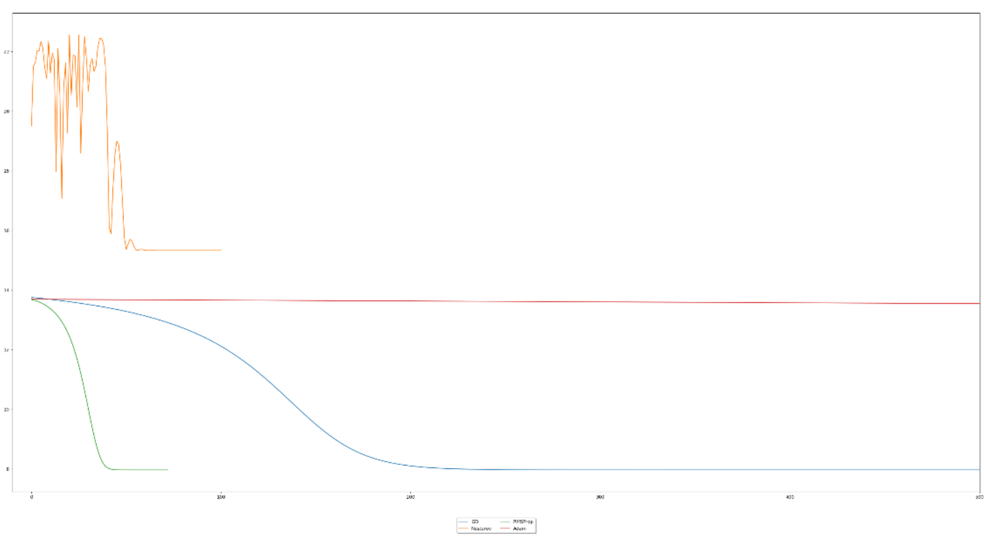



Ackley:

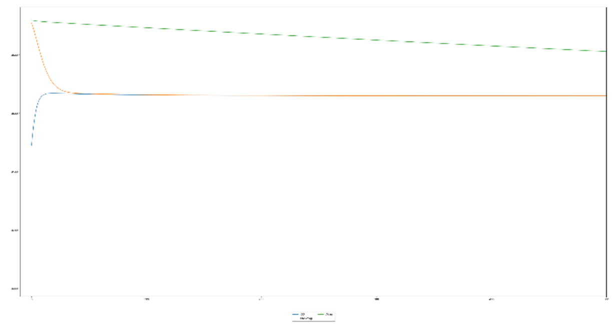
در صفحه بعد نیز نتیجه به ازای دو run مختلف نمایش داده شده است. مشاهده می‌شود که روش RMSProp به طرز فاحشی زودتر از سایر روش‌ها همگرا شده است و پس از آن بیشترین سرعت همگرایی به GD مربوط است. Adam در واقع در نقطه دیگری همگرا شده است که نقطه کمینه نیست. ولی احتمالاً local minimum است و به همین خاطر از آن‌جا تغییری نمی‌کند. نکته قابل توجه این است که Nesterov زودتر از GD همگرا می‌شود اما در مینیمم محلی و نه در Global Minimum و همچنین نوسانات زیادی دارد (که این نوسانات مثلاً با نوعی نیروی اصطکاک‌مانند در RMSProp رفع شده است.)

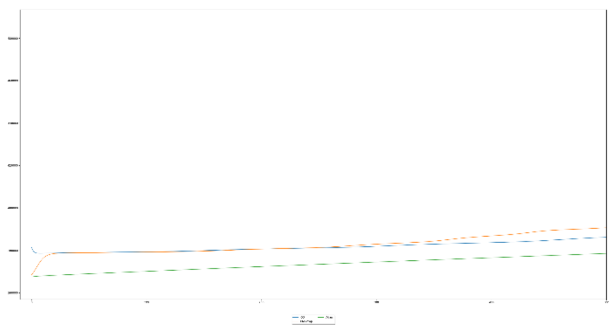


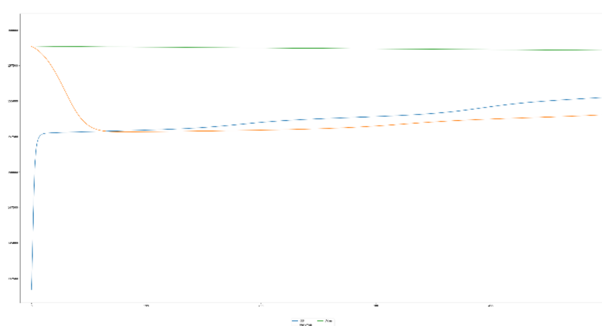



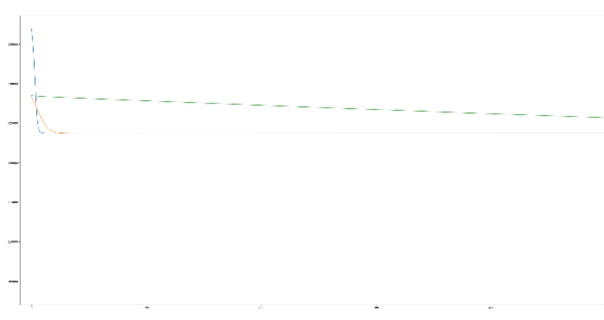

## Levi:

باز هم Nesterov مشکلاتی ایجاد می‌کند از جمله به خاطر شکل بسیار بد تابع آزمون به سرعت ممکن است واگرا شود و اعداد در این روش بزرگ می‌شوند و اکسپشن math domain error رخ می‌دهد. پس ما آن را جدای از بقیه رسم می‌کنیم. در زیر نتیجه به ازای چهار اجرای مختلف آورده شده‌است. (شکل‌های ۶ تا ۹ پیوست.) مشخص می‌شود برای این تابع آزمون GD و سپس RMSProp و سپس Adam همگرا می‌شوند ولی هیچ‌کدام در Global Minimum همگرا نشده‌اند بلکه به خاطر نقطه شروع تصادفی در Local Minimum دیگری همگرا شده‌اند. برای روش Nesterov به خاطر مشکل اشاره شده نتوانستم نموداری رسم کنم.



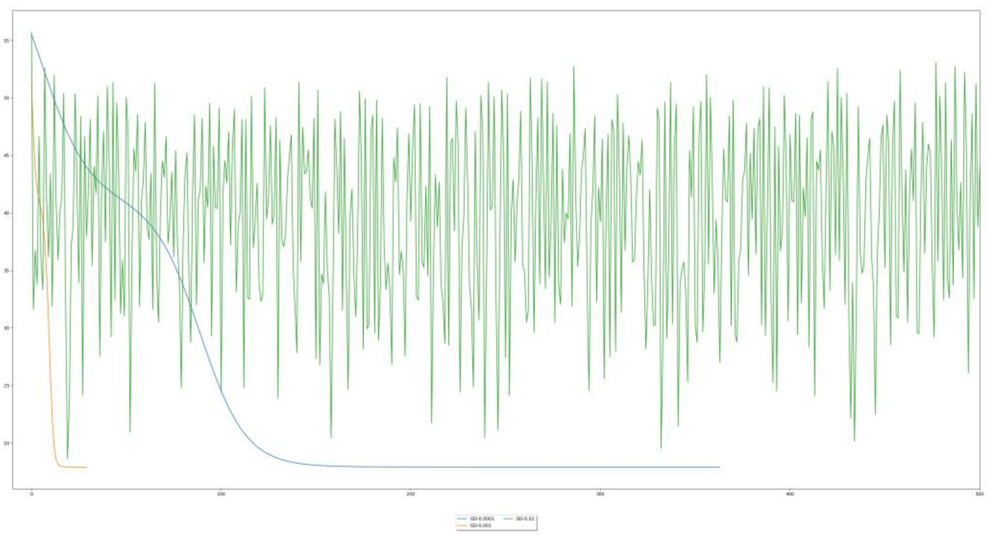




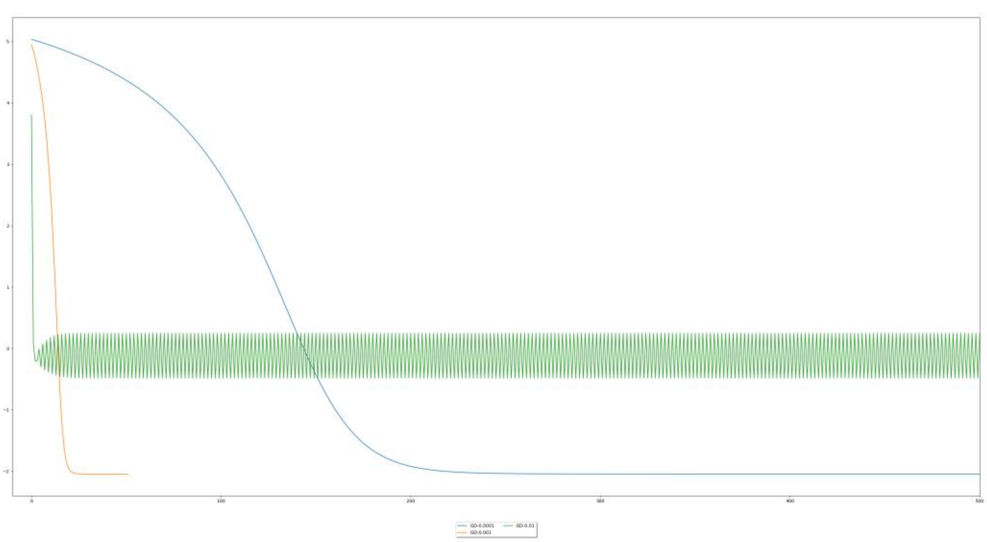


انتخاب Hyper parameter بهینه: کد این قسمت در زمان گزارش زده شد و چون کلیت کد را کثیف می‌کرد پاک شد. (به ازای ۰,۰۱ و ۰,۰۰۱ و ۰,۰۰۰۱ بررسی شده است.) جلوی هر title پارامتر بهینه نرخ یادگیری با توجه به نمودار نوشته شده‌است.

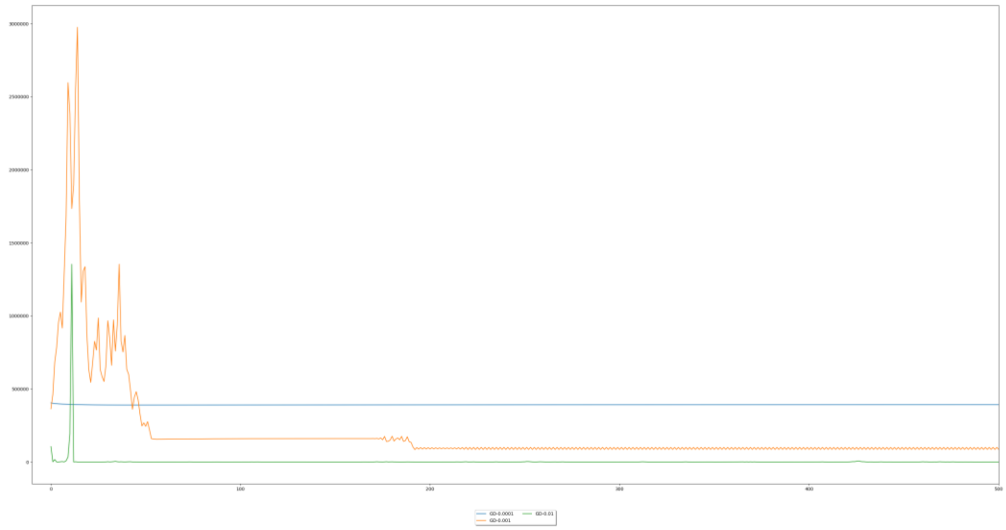
GD & Rastrigin: 0.001



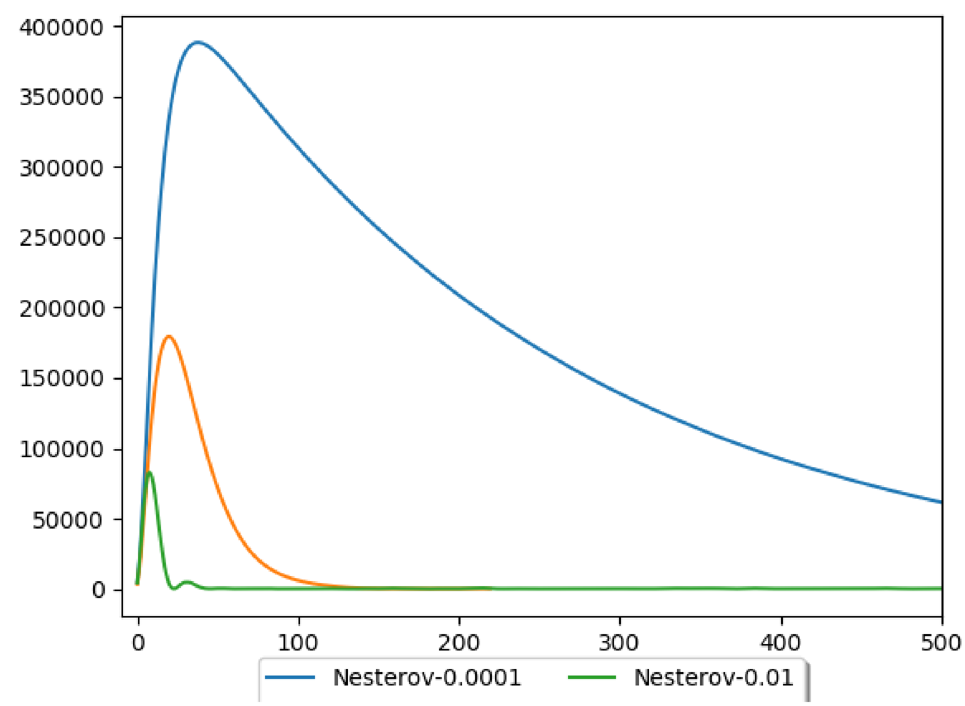
GD & Ackley: 0.01



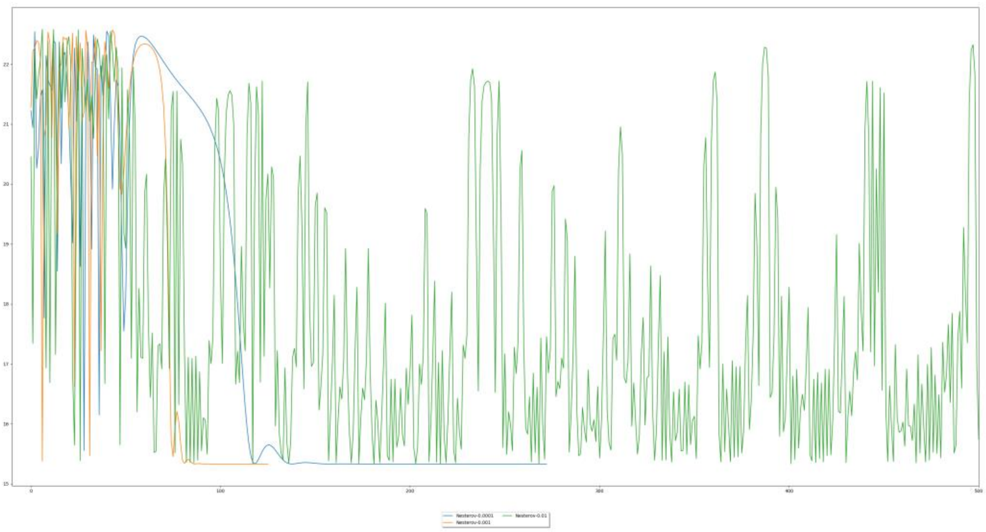
GD & Levi: 0.01



Nesterov & Rastrigin: 0.01

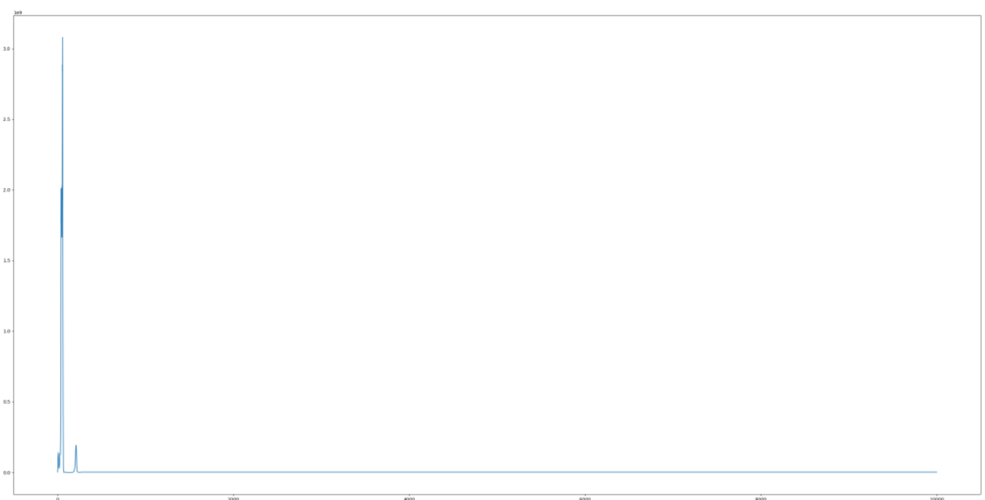


Nesterov & Ackley: 0.001

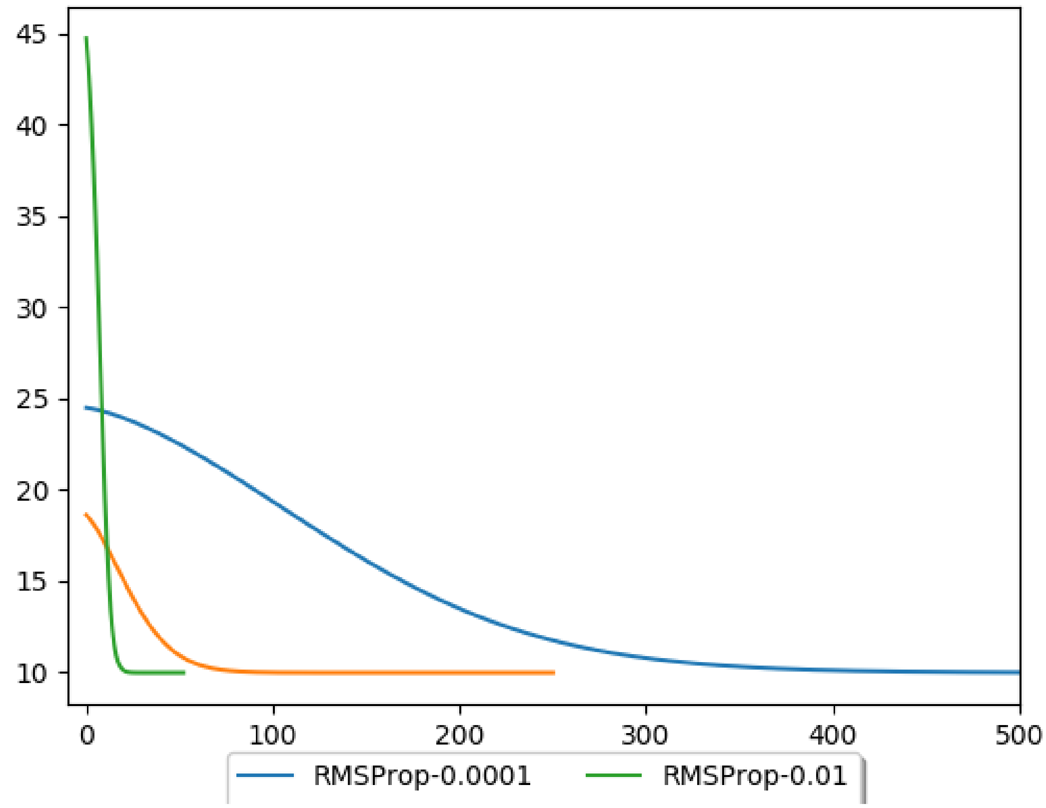



Nesterov & Levi: Cannot be determined.

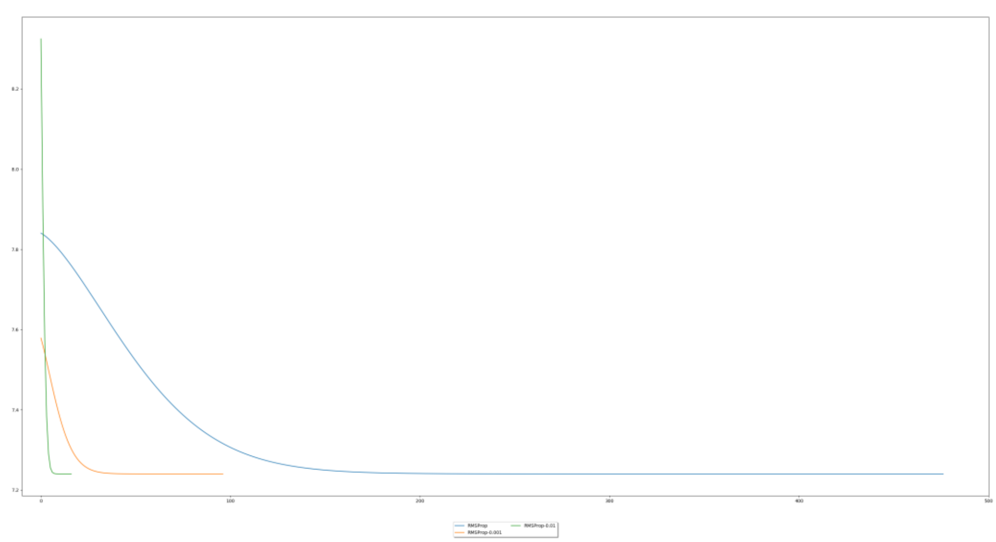
Has math domain error problem.



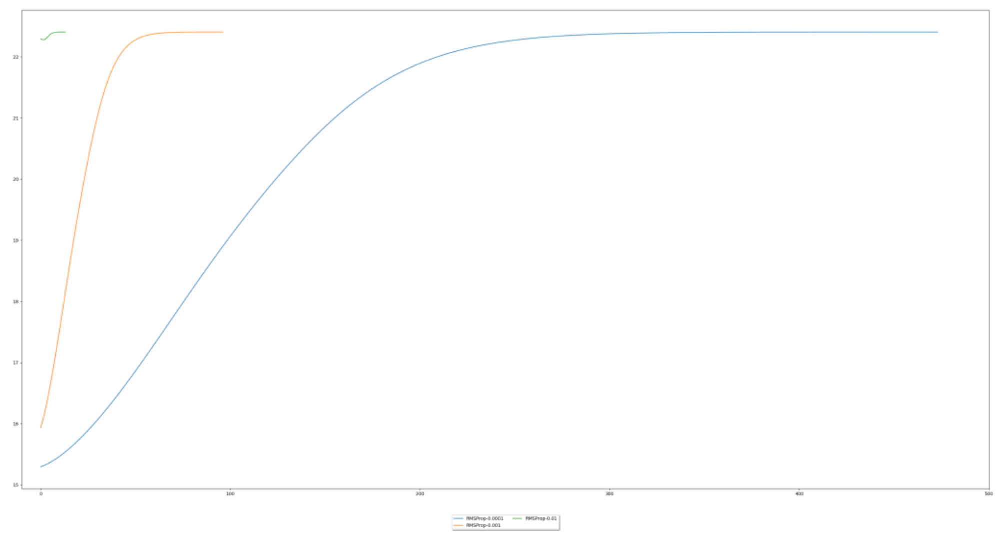
RMSProp & Rastrigin: 0.01



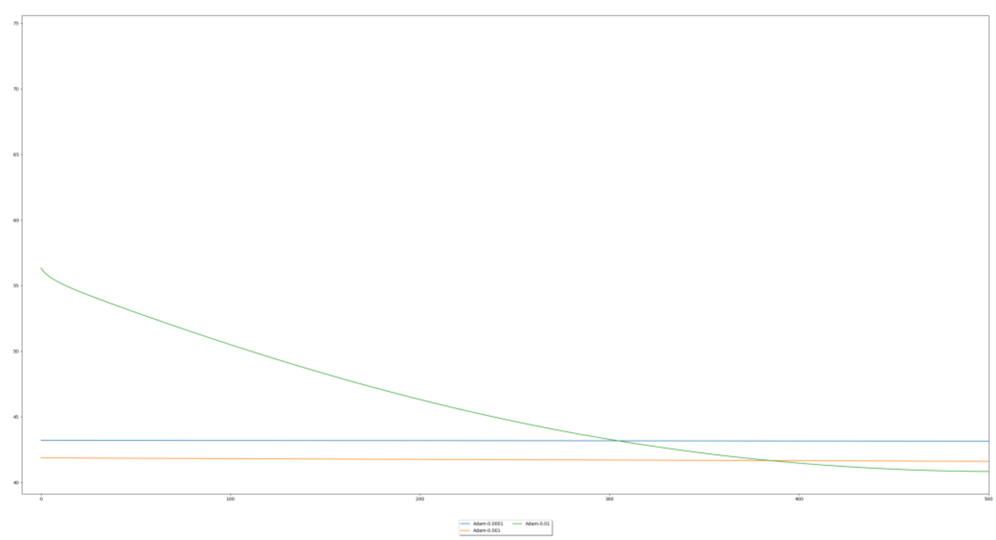

RMSProp & Ackley: 0.01



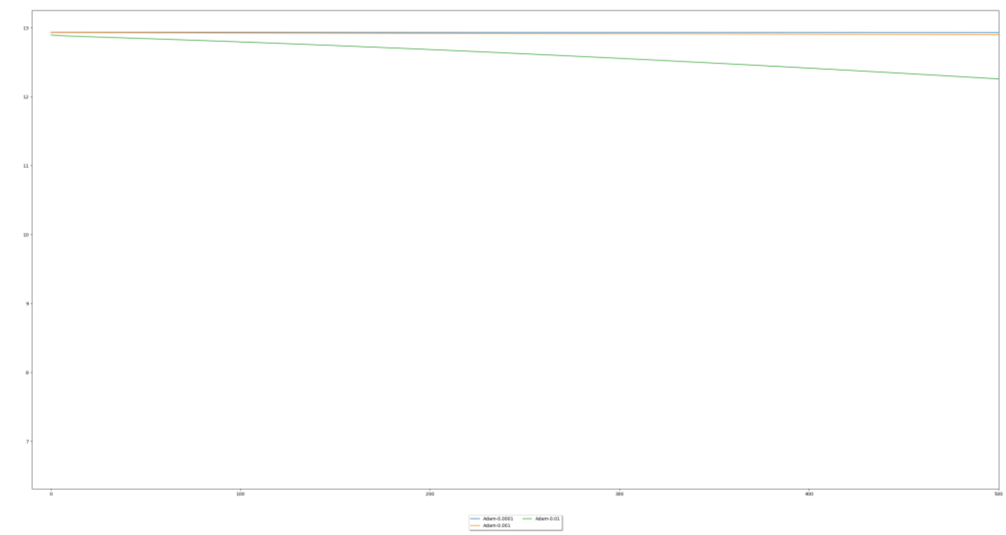
RMSProp & Levi: 0.01



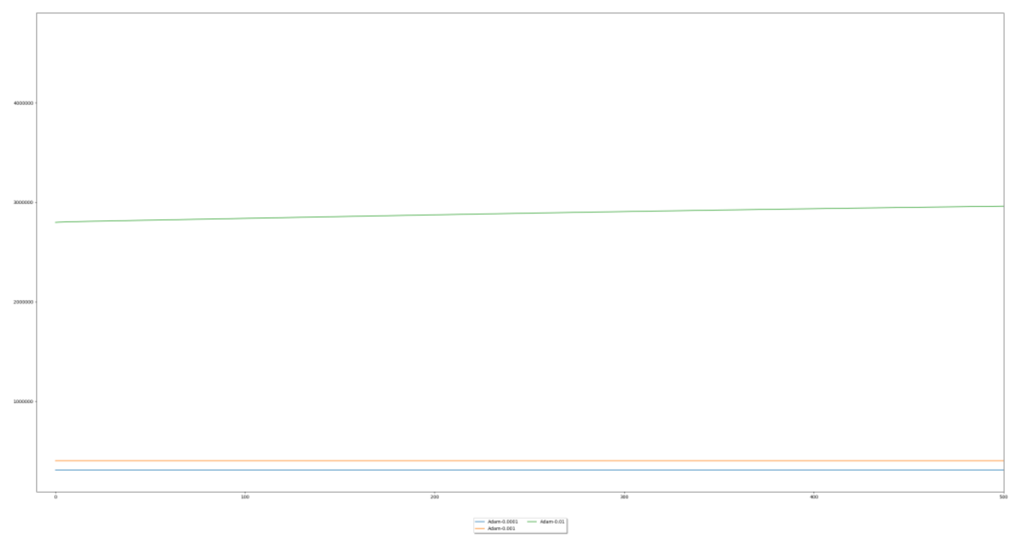

Adam & Rastrigin: 0.01



Adam & Ackley: 0.01




Adam & Levi: Cannot be determined. No one is converging.




در نمودارهای بالا پارامتر بهینه نرخ یادگیری به مسیری مربوط است که سریع‌تر از بقیه به حالت پایدار و هم‌چنین درست (Global Minimum) برسد و اعداد جلوی title ها بر این اساس نوشته شده‌اند.

۶. در روش نیوتن تابع به وسیله بسط تیلور تا مرتبه دوم تقریب زده می‌شود و در ابعاد بالاتر این مشتق‌ها بر اساس گرادیان و هسیان نوشته می‌شوند. الگوریتم بازگشتی برای update کردن متغیرها به صورت زیر خواهد بود:

$$x_{n+1} = x_n - \frac{\gamma \nabla f(x_n)}{\boldsymbol{H} f(x_n)}$$

تابع پیاده‌سازی آن درست بعد از پیاده‌سازی Adam نوشته شده و قطعه کد آن در تصویر زیر آورده شده‌است.

```python
############################################## Newton's Method ##############################################


def newton_method(target_function, coordinates, gradient_of_target_function, hessian_of_target_function,
                  optimization_hyper_parameters):
    counter = 0
    value = []
    step_size = optimization_hyper_parameters

    # while counter < 1e4 and target_function(coordinates[0], coordinates[1]) > 1e-3:
    while target_function(coordinates[0], coordinates[1]) > 1e-4 and counter < 1e4:

        hessian = hessian_of_target_function(coordinates[0], coordinates[1])
        hessian_inv = np.linalg.inv(hessian)
        [x_new, y_new] = list(np.array(coordinates) - step_size * np.matmul(hessian_inv, gradient_of_target_function(
            coordinates[0], coordinates[1])))
        counter += 1
        coordinates = [x_new, y_new]
        value.append(target_function(coordinates[0], coordinates[1]))

    return value

```

در رابطه بالا یک طول گام داریم که آن را یک فرض می‌کنیم. (البته قابل تنظیم است.) سپس به همان حلقه فعال با دو شرط همزمان برمی‌خوریم. تابع‌های هسیان نیز به عنوان ورودی به تابع روش نیوتن داده می‌شوند. هسیان محاسبه و سپس وارون می‌شود و در بردار گرادیان به صورت ماتریسی ضرب می‌شود. نهایتاً طبق رابطه روش نیوتن آپدیت انجام می‌شود. مقدار تابع در این iteration هم برای مقایسه سرعت در لیستی ذخیره می‌شود.

گرادیان برای همه توابع محاسبه شده و هسیان توابع آزمون (که تحویل الزامی داشتند) نیز به شرح زیر می‌باشد:

Rastrigin:

$$\text{Hessian}(f) = \begin{pmatrix} 40\pi^2\cos(2\pi x) + 2 & 0 \\ 0 & 40\pi^2\cos(2\pi y) + 2 \end{pmatrix}$$

## Levi:

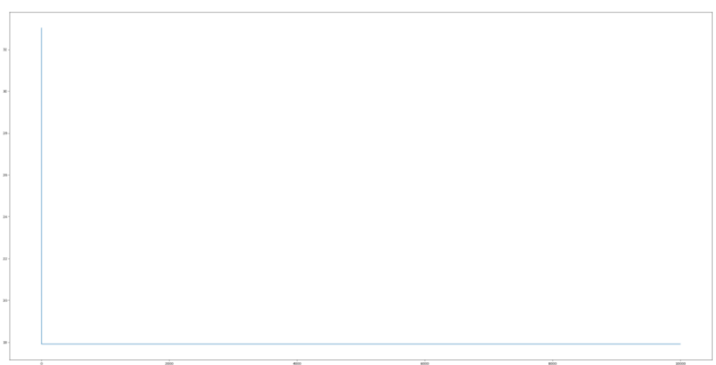
$$\mathrm{Hessian}(f) = \begin{pmatrix} -18\pi^2 \sin^2(3\pi x) + 18\pi^2 \cos^2(3\pi x) + 2(\sin^2(3\pi y) + 1) & 12\pi(x-1) \sin(3\pi y)\cos(3\pi y) \\ 12\pi(x-1)\sin(3\pi y)\cos(3\pi y) & -18\pi^2(x-1)^2 \sin^2(3\pi y) + 18\pi^2(x-1)^2\cos^2(3\pi y) + 2(4\pi^2 y^2 + 1) + 8\pi^2(y-1)^2 + 32\pi^2 y(y-1) \end{pmatrix}$$

در همان فایل‌های مربوط به Rastrigin و Levi کد این ماتریس‌ها زده شده و به عنوان تابع در اختیار سایر قسمت‌ها قرار می‌گیرند.

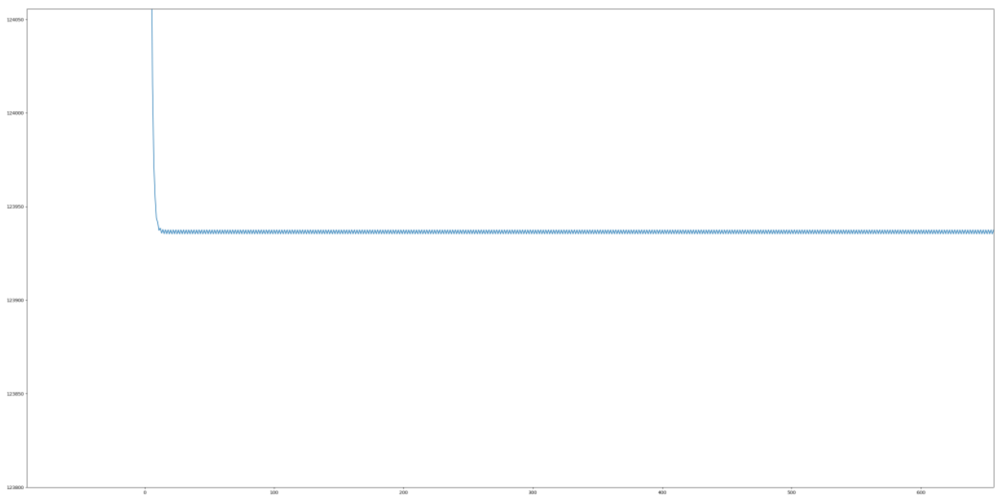
در انتهای کدهای تست توابع آزمون کد تست به روش نیوتن برای دو تابع آزمون الزامی زده شده‌است (همه قسمت‌های اجرایی به صورت دیفالت کامنت شده‌اند تا دستورات سنگین با هم اجرا نشوند و در صورتی که به اجرای بخشی نیاز داشتید لطفاً آن قسمت را از حالت کامنت خارج کنید. در صورت اجرای یک‌باره کد خروجی نمایش داده نخواهد شد!) (محدوده‌های اعداد تصادفی از ناحیه‌های مناسب برای سرچ در لینک قسمت‌های قبل انتخاب شده‌است.)

```
##################################### Testing Newton Method #####################################

# x = 2 * 5.12 * random.random() - 5.12
# y = 2 * 5.12 * random.random() - 5.12
#
# value = newton_method(Rastrigin.rastrigin, [x, y], Rastrigin.rastrigin_gradient, Rastrigin.rastrigin_hessian, 1)
# plt.plot(value)
# plt.show()
#
#
# x = 2 * 10 * random.random() - 10
# y = 2 * 10 * random.random() - 10
#
# value = newton_method(Levi.levi, [x, y], Levi.levi_gradient, Levi.levi_hessian, 1)
# plt.plot(value)
# plt.show()

```

## Rastrigin:



Levi:



سرعت همگرایی آن از بقیه روش‌ها بیشتر است اما اصلاً به نظر نمی‌رسد به جای درستی همگرا شده باشد به خصوص که در مورد Levi نوسانات دنباله‌داری نیز مشاهده می‌شود.

علت عدم استفاده از این روش بار محاسباتی سنگین آن در قسمت هسیان و وارون آن است. مشتق دوم بسیاری از توابع بسیار روابط پیچیده و سنگینی دارند که محاسبات را سنگین می‌کند و حتی در صورت ساده بودن توابع، در ابعاد وارون گرفتن از ماتریس یا حل همزمان دستگاه به روش‌های جبرخطی بسیار هزینه بر می‌شود و این مزیدی بر علت قبلی برای عدم استفاده از این روش برای بهینه‌سازی می‌باشد. اما نکته قابل توجه تضمین همگرایی یافتن ریشه در این روش است.