EECS 127/227AT Optimization Models in Engineering UC Berkeley Spring 2023

Speeding up Gradient Descent

1 Introduction and Background

Gradient descent has been the optimization algorithm of choice in a large amount of optimization applications, especially those which operate at large scales (where Hessians are too difficult to repeatedly compute). Yet gradient descent is not the fastest converging algorithm, even compared to other algorithms which have just zeroth and first-order information (i.e., knowledge of the function and its gradient). Indeed, Nesterov showed that it is possible to design a sped-up version of gradient descent, called *accelerated gradient descent*, which has asymptotically faster convergence rate than gradient descent [1].[1] This accelerated method is, at least at first, seemingly complicated and unmotivated; the original proof of convergence rate given by Nesterov amounted to a set of mysterious algebraic manipulations. Even one of the most well-known optimization theorists, Sebastian Bubeck, posted on his blog about how understanding Nesterov's acceleration is difficult to understand.

Recently, there have been several lines of research attempting to understand and extend Nesterov's acceleration. One of these attempts uses a generalization of gradient descent called *mirror descent* combined with gradient descent to improve the convergence rate of the algorithm [3], giving an interpretable modification of gradient descent which achieves optimal convergence rates. In this project, we introduce mirror descent and apply it to accelerate gradient descent.

2 Problems

1. Overview of Relevant Literature

In this problem, you will read two research papers: Nesterov's original accelerated gradient descent, and a follow-up which aims to achieve the same convergence rate by mixing gradient descent and mirror descent. The aim is to gain a better understanding of the topics discussed in this project and to get insights into the state-of-the-art development in our understanding of accelerating gradient descent. You will summarize the main results and findings of each paper and answer a few questions about them.

(a) Read and summarize the main results in the paper "A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$" by Nesterov [1]. In your summary include answers to the following questions:

i. Describe the main assumptions made in the paper on the function f to optimize. What kinds of functions are shown to be optimized via accelerated gradient descent? (*HINT:* In this paper, Nesterov uses the notation f' to denote the gradient ∇f.)

ii. What is the update rule at the k^{th} step?

iii. After T iterations, what is an upper-bound on $f(\vec{x}_T) - \min_{\vec{x}\in\mathbb{R}^n} f(\vec{x})$?

(b) Read and summarize the main results in the paper "Linear Coupling: An Ultimate Unification of Gradient and Mirror Descent" by Zhu and Orecchia. In your summary include answers to the following questions:

i. What do the authors claim is the kind of progress made by gradient descent? What about mirror descent?

[1] In fact, it achieves asymptotically optimal convergence rates; a proof of this is contained in [2].

ii. Describe the main assumptions made in the paper on the function f to optimize. What kinds of functions are shown to be optimized via the algorithm `AGM` proposed by Zhu and Orecchia?

iii. After T iterations of `AGM`, what is an upper-bound on $f(\vec{y}_T) - \min_{\vec{x} \in Q} f(\vec{x})$?

iv. What is the relationship between this paper [3] and Nesterov's original accelerated gradient descent paper [1]?

a) https://vsokolov.org/courses/750/2018/files/nesterov.pdf

This paper outlines two different problems and the methods to solve them. However, these problems are almost completely equivalent, and just in different forms. The first problem is $\min\{ f(x) | \, x \in E\}$ where E is the hilbert space. The assumptions on f is that f is convex, and the constraint that:

$$||f'(x) - f'(y)|| \leq L||x - y|| : \forall \, x, \, y \in E, \, L > 0.$$

The way that the accelerated gradient descent works is to set:

$$k = 0, \, a_0 = 1, \, x_{-1} = y_0, \, \alpha_{-1} = \frac{||y_0 - z||}{||f'(y_0) - f'(z)||},$$

where z is an arbitrary point in E such that $f'(z) \neq f'(y_0)$ and where y_0 is our initial point.

At the kth iteration, find the smallest integer $i \geq 0$ such that:

$$f(y_k) - f(y_k - 2^{-i}\alpha_{k-1} f'(y_k)) \geq 2^{-i-1}\alpha_{k-1}||f'(y_k)||^2,$$

$$\text{where } \alpha_k = 2^{-i}\alpha_{k-1}, \, x_k = y_k - \alpha_k f'(y_k), \, a_{k+1} = \frac{1}{2}(1 + \sqrt{4a_k^2 + 1}),$$

$$y_{k+1} = x_k + (a_k - 1)(x_k - x_{k-1})/a_{k+1}$$

After T iterations, the upper bound on the difference between the min and the Tth value is:

$$f(x_T) - min_{\hat{x} \in R} f(\hat{x}) \leq \frac{4L||y_0 - \hat{x}||^2}{(T+2)^2}$$

The second problem is $\min\{F(\bar{f}(x)) | x \in Q\}$ where Q is any convex set that is in the hilbert space E. Essentially the only difference is that in the first problem, our problem is to minimize a function given an initial vector, but in the second problem, we are now minimizing a function given a vector of convex functions that take inputs on E.

b) https://arxiv.org/pdf/1407.1537.pdf

The authors claim that whereas mirror descent makes progress in the Bregman divergence between successive iterations, gradient descent makes progress in the Euclidean distance between them.

The function f to be optimized is assumed to be convex, smooth, and strongly convex in the paper. It is demonstrated that the Zhu and Orecchia algorithm AGM can optimize functions that adhere to these assumptions as well as functions that are weakly convex but still satisfy to the weaker growth condition.

The upper-bound on $f(\vec{y}_T) - \min_{\vec{x} \in Q} f(\vec{x})$ after T iterations of AGM is O(1/T^2), which is the same as the rate of convergence of the initial accelerated gradient descent algorithm suggested by Nesterov.

By suggesting a new optimization process that combines gradient descent with mirror descent, this study expands upon Nesterov's earlier accelerated gradient descent paper. In comparison to the individual approaches, the authors demonstrate that the new algorithm achieves quicker convergence rates and greater generalization performance, and while also offering theoretical guarantees for its convergence.

2. Mirror Descent

In this problem, we will go through a proof of mirror descent in the case of entropy and ℓ^2 regularization.

Consider the problem

$$f^\star = \min_{\vec{x}\in\mathcal{X}} f(\vec{x}). \tag{1}$$

where $\mathcal{X} \subseteq \mathbb{R}^n$ is a convex subset of $\mathbb{R}^n$. Often, running (projected) gradient descent is not the right thing to do and can lead to slow convergence. This is usually because the convergence rates of Euclidean gradient descent are of the form $f(\vec{x}_k) - f^\star \leq O(L\|\vec{x}_0 - \vec{x}^\star\|_2^2/k)$ and it may be the case for $\mathcal{X}$ that the ℓ^2 distance of the initial point from the optimal and/or the smoothness parameter L that's measured according to the ℓ^2 may be quite large. But taking the "right geometry" into account when *defining the update step* may lead to much faster convergence. For the sake of simplicity, however, for all except one subpart, we will only work for the ℓ^2 case in this problem.

For any convex (doubly)-differentiable function $h : \mathcal{X} \to \mathbb{R}$, define the Bregman divergence of h as

$$D_h(\vec{y};\vec{x}) = h(\vec{y}) - h(\vec{x}) - \langle \nabla h(\vec{x}), \vec{y} - \vec{x}\rangle \tag{2}$$

where $\langle \vec{u}, \vec{v}\rangle$ is the dot product between two vectors $\vec{u}$ and $\vec{v}$.

Algorithm 1 Mirror Descent Algorithm

$\vec{x}_0$ is a uniformly random point in $\mathcal{X}$
$k = 0$
while $k \leq T$ **do**
  $\eta_k > 0$ a step size.
  $\vec{g}_k \leftarrow \nabla f(\vec{x}_k)$
  $\vec{x}_{k+1} = \text{Mirr}(\eta_k \vec{g}_k; \vec{x}_k)$ where $\text{Mirr}(\vec{g};\vec{x}) = \operatorname{argmin}_{\vec{z}\in\mathcal{X}} \{\langle \vec{g}, \vec{z}\rangle + D_h(\vec{z};\vec{x})\}$ is the Mirror Descent step.
  $k \leftarrow k + 1$
end while
return $\overline{x}_T = \frac{1}{T}\sum_{k=0}^{T} \vec{x}_k$

(a) Prove that for any convex set $\mathcal{X}$ and α-strongly convex function $h : \mathcal{X} \to \mathbb{R}$, for any fixed $\vec{x} \in \mathcal{X}$, the Bregman divergence $D_h(\vec{y};\vec{x})$ is a α-strongly convex function of $\vec{y}$. Note that by taking $\alpha = 0$, this proves that if h is convex, the Bregman divergence is convex as well.

(b) Something that's going to be useful in a convergence proof of mirror descent is going to be the so-called Bregman three-point inequality. Formally, prove that

$$\langle \nabla h(\vec{x}) - \nabla h(\vec{y}), \vec{y} - \vec{u}\rangle = D_h(\vec{u};\vec{x}) - D_h(\vec{u};\vec{y}) - D_h(\vec{y};\vec{x}) \tag{3}$$

(c) Let's try to understand the mirror descent update in some special cases. For this part, assume $\mathcal{X} = \{\vec{x} \in \mathbb{R}^n | x_i \geq 0 \ \forall i \in [n] \text{ and } \sum_{i=1}^n x_i = 1\}$ is the n-dimensional probability simplex. We will take $h(\vec{x}) = \sum_{i=1}^n (x_i \log(x_i) - x_i)$, the entropy function. Given $\vec{x} \in \mathcal{X}$ and given some $\vec{g} \in \mathbb{R}^n$ and $\eta > 0$, compute $\text{Mirr}(\eta\vec{g};\vec{x})$. Since $\mathcal{X}$ is constrained to be the simplex, you will have to use a Lagrange multiplier for the $\sum_{i=1}^n x_i = 1$ constraint and eliminate the Lagrange multiplier from the final solution. In this setting, this algorithm goes by a more popular name which is "multiplicative weights update method".

a) $D_h(\vec{y};\vec{x}) = h(\vec{y}) - h(\vec{x}) - \underbrace{\langle \nabla h(\vec{x}), \vec{y}-\vec{x}\rangle}_{\nabla h(\vec{x})^T(\vec{y}-\vec{x})}$

$\Rightarrow \nabla D_h(\vec{y};\vec{x}) = \nabla h(\vec{y}) - \nabla h(\vec{x})$

$\Rightarrow \nabla^2 D_h(\vec{y};\vec{x}) = \nabla^2 h(\vec{y})$

Thus, $h(\vec{y})$ being α-strongly convex

$\Leftrightarrow \nabla^2 h(\vec{y}) - \alpha I \succeq 0 \Leftrightarrow \nabla^2 D_h(\vec{y};\vec{x}) - \alpha I \succeq 0$

$\Leftrightarrow D_h(\vec{y};\vec{x})$ is α-strongly convex

Thus, if $h(\vec{y})$ is convex [so $\alpha=0$], then $D_h(\vec{y};\vec{x})$ also convex.

b) $D_h(\vec{u};\vec{x}) - D_h(\vec{u};\vec{y}) - D_h(\vec{y};\vec{x}) =$

$h(\vec{u}) - h(\vec{x}) - \langle \nabla h(\vec{x}), \vec{u}-\vec{x}\rangle$

$- h(\vec{u}) + h(\vec{y}) + \langle \nabla h(\vec{y}), \vec{u}-\vec{y}\rangle$

$- h(\vec{y}) + h(\vec{x}) + \langle \nabla h(\vec{x}), \vec{y}-\vec{x}\rangle$

$= \langle \nabla h(\vec{x}), \vec{y}-\vec{x}\rangle - \langle \nabla h(\vec{x}), \vec{u}-\vec{x}\rangle + \langle \nabla h(\vec{y}), \vec{u}-\vec{y}\rangle$

$= \langle \nabla h(\vec{x}), \vec{y}-\vec{u}\rangle + \langle \nabla h(\vec{y}), \vec{u}-\vec{y}\rangle$

$= \langle \nabla h(\vec{x}) - \nabla h(\vec{y}), \vec{y}-\vec{u}\rangle$

$\therefore D_h(\vec{u};\vec{x}) - D_h(\vec{u};\vec{y}) - D_h(\vec{y};\vec{x}) = \langle \nabla h(\vec{x}) - \nabla h(\vec{y}), \vec{y}-\vec{u}\rangle$

c) $\text{mirr}(\eta\vec{g};\vec{x}) = \underset{\vec{z}\in\mathbb{R}^n;\, z_i>0,\, \sum_{i=1}^n z_i = 1}{\text{argmin}} \langle \eta\vec{g}, \vec{z}\rangle + h(\vec{z}) - h(\vec{x}) - \langle \nabla h(\vec{x}), \vec{z}-\vec{x}\rangle$

$= \underset{\vec{z}\in\mathbb{R}^n:\, z_i>0,\, \sum_{i=1}^n z_i = 1}{\text{argmin}} \eta\langle \vec{g}, \vec{z}\rangle + \sum_{i=1}^n [z_i \log z_i - z_i] - \sum_{i=1}^n [x_i \log x_i - x_i] - \langle \log \vec{x}, \vec{z}-\vec{x}\rangle.$

$= \underset{\vec{z}\in\mathbb{R}^n:\, z_i>0,\, \sum_{i=1}^n z_i = 1}{\text{argmin}} \eta\vec{g}^T\vec{z} + \sum_{i=1}^n [z_i \log z_i - z_i] - \sum_{i=1}^n z_i \log x_i + \lambda \sum_{i=1}^n z_i - \lambda$

$= \underset{\vec{z}\in\mathbb{R}^n,\, \lambda\in\mathbb{R}}{\text{argmin}} \eta\vec{g}^T\vec{z} + \sum_{i=1}^n z_i(\log z_i - \log x_i + \lambda - 1) - \lambda.$

$= \underset{\vec{z}\in\mathbb{R}^n,\, \lambda\in\mathbb{R}}{\text{argmin}} \eta\vec{g}^T\vec{z} + \sum_{i=1}^n z_i(\log z_i - \log x_i + \lambda) - \lambda.$

FOC: $\frac{\partial f}{\partial z_i} = \eta g_i + \log z_i - \log x_i + \lambda - 1 + 1$

$\Rightarrow \log \frac{z_i^*}{x_i} = -\lambda - \eta g_i \Rightarrow x_i e^{-\lambda - \eta g_i} = z_i^*$

Then, $\sum z_i^* = 1 \Rightarrow \sum x_i e^{-\lambda - \eta g_i} = 1$

$\Rightarrow e^{\lambda} = \sum x_i e^{-\eta g_i} \Rightarrow \lambda = \log(\sum x_i e^{-\eta g_i})$

$\Rightarrow z_i^* = \frac{x_i e^{-\eta g_i}}{\sum x_i e^{-\eta g_i}}$

$\therefore \text{mirr}(\eta\vec{g};\vec{x}) = \frac{\vec{x}e^{-\eta\vec{g}}}{\sum_{i=1}^n x_i e^{-\eta g_i}}.$

(d) Now, for the rest of the problem, for simplicity, we will assume $\mathcal{X} = \mathbb{R}^n$ and $h(\vec{x}) = \frac{1}{2}\|\vec{x}\|_2^2$. In this case, given a $\vec{g} \in \mathbb{R}^n$, $\eta > 0$ and $\vec{x} \in \mathcal{X}$, compute $\text{Mirr}(\eta\vec{g}; \vec{x})$. Also compute $D_h(\vec{y}, \vec{x})$ in this case. Do these look familiar to something you have already seen?

$$D_h(\vec{y};\vec{x}) = h(\vec{y}) - h(\vec{x}) - \langle \nabla h(\vec{x}), \vec{y} - \vec{x} \rangle$$
$$= \frac{1}{2}\vec{y}^T\vec{y} - \frac{1}{2}\vec{x}^T\vec{x} - \langle \vec{x}, \vec{y} - \vec{x} \rangle$$
$$= \frac{1}{2}\vec{y}^T\vec{y} - \frac{1}{2}\vec{x}^T\vec{x} - \vec{x}^T\vec{y} + \vec{x}^T\vec{x}.$$
$$= \frac{1}{2}\vec{x}^T\vec{x} + \frac{1}{2}\vec{y}^T\vec{y} - \vec{x}^T\vec{y} = \frac{1}{2}\|\vec{x} - \vec{y}\|_2^2$$

$$\Rightarrow \text{mirr}(\eta\vec{g}; \vec{x}) = \underset{\vec{z} \in \mathbb{R}^n}{\text{argmin}} \langle \eta\vec{g}, \vec{z} \rangle + \frac{1}{2}\|\vec{x} - \vec{z}\|_2^2 \longrightarrow \text{Looks like a constrained least squares problem.}$$
$$= \underset{\vec{z} \in \mathbb{R}^n}{\text{argmin}} \frac{1}{2}\vec{z}^T\vec{z} - \vec{x}^T\vec{z} + \eta\vec{g}^T\vec{z}. \quad [\nabla^2 f(\vec{z}) = I_{n\times n} \succeq 0]$$
$$\Rightarrow \nabla f(\vec{z}) = \vec{z}^* - \vec{x} + \eta\vec{g} = \vec{0} \Rightarrow \vec{z}^* = \vec{x} - \eta\vec{g}.$$

Then, $D_h(\vec{z}^*; \vec{x}) = \frac{1}{2}\|\eta\vec{g}\|_2^2 = \frac{\eta^2}{2}\|\vec{g}\|_2^2$.

(e) To prove convergence of mirror descent, it's convenient to introduce a term from online learning, called regret. For any feasible solution $\vec{u} \in \mathcal{X}$, we define regret in the k^{th} iteration as $\text{Reg}_k(\vec{u}) = \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{u} \rangle$. We will first prove an upper bound on the regret of the k^{th} iteration. Formally, prove that

$$\text{Reg}_k(\vec{u}) = \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{u} \rangle = \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} \rangle + D_h(\vec{u}; \vec{x}_k) - D_h(\vec{u}; \vec{x}_{k+1}) - D_h(\vec{x}_{k+1}; \vec{x}_k) \quad (4)$$
$$= \frac{\eta_k^2\|\vec{g}_k\|_2^2}{2} + D_h(\vec{u}; \vec{x}_k) - D_h(\vec{u}; \vec{x}_{k+1}) \quad (5)$$

This inequality will show up again in the proof of accelerated gradient descent in another question in the

$$\text{Reg}_k(\vec{u}) = \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{u} \rangle = \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} + \vec{x}_{k+1} - \vec{u} \rangle$$
$$= \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} \rangle + \langle \eta_k\vec{g}_k, \vec{x}_{k+1} - \vec{u} \rangle$$

$[\nabla h(\vec{x}_k) - \nabla h(\vec{x}_{k+1}) = \vec{x}_k - \vec{x}_{k+1} = \vec{x}_k - \vec{x}_k + \eta_k\vec{g}_k = \eta_k\vec{g}_k]$ $\longrightarrow$

$$= \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} \rangle + \langle \nabla h(\vec{x}_k) - \nabla h(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{u} \rangle$$
$$= \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} \rangle + D_h(\vec{u}; \vec{x}_k) - D_h(\vec{u}; \vec{x}_{k+1}) - D_h(\vec{x}_{k+1}; \vec{x}_k) \quad [\text{By (b)}]$$

$$[\langle \eta_k\vec{g}_k, \vec{x}_k - \vec{x}_{k+1} \rangle - D_h(\vec{x}_{k+1}; \vec{x}_k) = \langle \eta_k\vec{g}_k, \eta_k\vec{g}_k \rangle - h(\vec{x}_{k+1}) + h(\vec{x}_k) + \langle \vec{x}_{k+1} - \eta_k\vec{g}_k \rangle$$
$$= \langle \eta_k\vec{g}_k - \vec{x}_k, \eta_k\vec{g}_k \rangle - \frac{1}{2}\|\vec{x}_k - \eta_k\vec{g}_k\|_2^2 + \frac{1}{2}\|\vec{x}_k\|_2^2 = \|\eta_k\vec{g}_k\|_2^2 - \vec{x}_k^T\eta_k\vec{g}_k - \frac{1}{2}\|\eta_k\vec{g}_k\|_2^2 + \vec{x}_k^T\eta_k\vec{g}_k$$
$$= \frac{1}{2}\eta_k^2\|g_k\|_2^2 \Big] \text{ o.w., } \text{Reg}_k(\vec{u}) = \frac{1}{2}\eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_k) - D_h(\vec{u}; \vec{x}_{k+1})$$

(f) Now we will consider the total regret over T iterations, i.e., $\text{TotalReg}_T(\vec{u}) = \sum_{i=0}^{T} \langle \eta_k\vec{g}_k, \vec{x}_k - \vec{u} \rangle$. Prove that

$$\text{TotalReg}_T(\vec{u}) \leq \sum_{k=0}^{T} \eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_0)$$

$$\text{Total Reg}_T(\vec{u}) = \sum_{k=0}^{T} \text{Reg}_k(\vec{u}) = \sum_{k=0}^{T} \left\{ \frac{1}{2}\eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_k) - D_h(\vec{u}; \vec{x}_{k+1}) \right\}$$
$$= \frac{1}{2}\sum_{k=0}^{T} \eta_k^2\|\vec{g}_k\|_2^2 + \sum_{k=0}^{T} D_h(\vec{u}; \vec{x}_k) - \sum_{k=1}^{T+1} D_h(\vec{u}; \vec{x}_k)$$
$$= \frac{1}{2}\sum_{k=0}^{T} \eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_0) - D_h(\vec{u}; \vec{x}_{T+1})$$

$[D_h(\vec{u}; \vec{x}_{T+1}) = \frac{1}{2}\|\vec{u} - \vec{x}_{T+1}\|_2^2 > 0]$ $\leq \frac{1}{2}\sum_{k=0}^{T} \eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_0) \leq \sum_{k=0}^{T} \eta_k^2\|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_0)$

(g) Now, we will prove a lower bound on the regret in terms of the function value at $\bar{x}_T$ and at $\vec{u}$. Taking $\eta_k = \eta$ for all k, prove that

$$\text{TotalReg}_T(\vec{u}) \geq T\eta(f(\bar{x}_T) - f(\vec{u})) \tag{7}$$

Using this, conclude that

$$f(\bar{x}_T) \leq f(\vec{x}^\star) + \frac{1}{T}\left[\eta \sum_{i=0}^{T} \|\vec{g}_k\|_2^2 + D_h(\vec{x}^\star; \vec{x}_0)/\eta\right] \tag{8}$$

By convexity of $f(\vec{x})$, we derive the first order condition:
$f(\vec{u}) - f(\vec{x}_t) \leq \nabla f(\vec{u})^T(\vec{u} - \vec{x}_t)$ [and $\vec{g}_t = \nabla f(\vec{x}_t)$]

Also, $\text{Total Reg}_T(\vec{u}) = \sum_{k=0}^{T-1} \langle \eta_k \vec{g}_k, \vec{x}_k - \vec{u}\rangle = \sum_{k=0}^{T-1} \eta_k \langle \vec{g}_k, \vec{x}_k - \vec{u}\rangle$

$= \eta \sum_{k=0}^{T-1} \langle \nabla f(\vec{x}_k), \vec{x}_k - \vec{u}\rangle = \eta \sum_{k=0}^{T-1} \nabla f(\vec{x}_k)^T(\vec{x}_k - \vec{u})$

$\geq \eta \sum_{k=0}^{T-1} [f(\vec{x}_k) - f(\vec{u})] = \eta \left[\sum_{k=0}^{T-1} f(\vec{x}_k) - \eta T f(\vec{u})\right] = \eta T(f(\bar{x}_T) - f(\vec{u}))$

Thus, $\eta^2 \sum_{k=0}^{T} \|\vec{g}_k\|_2^2 + D_h(\vec{u}; \vec{x}_0) \geq \text{Total Reg}_T(\vec{u}) \geq T\eta(f(\bar{x}_T) - f(\vec{u}))$ $\forall\ \vec{u}$

i.e. $\eta^2 \sum_{k=0}^{T} \|\vec{g}_k\|_2^2 + D_h(\vec{x}^*; \vec{x}_0) \geq f(\bar{x}_T) - f(\vec{x}^*)$ [taking $\vec{u} = \vec{x}^*$]

$\Rightarrow \frac{1}{T}\left[\eta \sum_{k=0}^{T} \|\vec{g}_k\|_2^2 + D_h(\vec{x}^*; \vec{x}_0)/\eta\right] + f(\vec{x}^*) \geq f(\bar{x}_T)$.

(h) Now, assume that the function is L-Lipschitz (note that this is asking for the function to be Lipschitz and not the gradient of the function to be L-Lipschitz). It can be easily proven (and you may assume so without proof) that $\|\nabla f(\vec{x})\|_2 \leq L$ for all $\vec{x} \in \mathcal{X}$. Conclude that

$$f(\bar{x}_T) \leq f(\vec{x}^\star) + \eta L^2 + \frac{1}{2\eta T}\|\vec{x}_0 - \vec{x}^\star\|_2^2 \tag{9}$$

Show that there exists an $\eta > 0$ such that

$$f(\bar{x}_T) \leq f(\vec{x}^\star) + \frac{\sqrt{2}L\|\vec{x}_0 - \vec{x}^\star\|_2}{\sqrt{T}} \tag{10}$$

Using simplified expression for D_h from (d).

$f(\bar{x}_T) \leq f(\vec{x}^*) + \frac{1}{T}\left[\eta \sum_{k=0}^{T-1} \|\nabla f(\vec{x}_k)\|_2^2 + \frac{1}{2\eta}\|\vec{x}_0 - \vec{x}^*\|_2^2\right]$

$\leq f(\vec{x}^*) + \frac{1}{T}\eta \sum_{k=0}^{T-1} L^2 + \frac{1}{2\eta T}\|\vec{x}_0 - \vec{x}^*\|_2^2 = f(\vec{x}^*) + \eta L^2 + \frac{1}{2\eta T}\|\vec{x}_0 - \vec{x}^*\|_2^2$

Then, $\eta L^2 + \frac{1}{2\eta T}\|\vec{x}_0 - \vec{x}^*\|_2^2 \leq \frac{\sqrt{2}L}{\sqrt{T}}\|\vec{x}_0 - \vec{x}^*\|_2$

$\eta^2 L^2 - \frac{2\eta L}{\sqrt{2}\sqrt{T}}\|\vec{x}_0 - \vec{x}^*\|_2 + \frac{1}{2T}\|\vec{x}_0 - \vec{x}^*\|_2^2 \leq 0$.

$\Rightarrow \left(\eta L - \frac{\|\vec{x}_0 - \vec{x}^*\|_2}{\sqrt{2}\sqrt{T}}\right)^2 \leq 0 \Rightarrow \eta = \frac{\|\vec{x}_0 - \vec{x}^*\|_2}{\sqrt{2T} \cdot L}$

(i) In the accompanying Jupyter notebook, you will implement the mirror descent update for the entropy regularizer and for the ℓ^2 regularizer. The input to the function will be step size η_k, a vector $\vec{g}$ which is meant to represent the gradient, and the current point $\vec{x}_k$.

Hence the convergence is at a rate of $1/\sqrt{T}$. While we did the proof for the unconstrained setting and with the ℓ^2 geometry, this proof with very few changes can be used to prove similar results in constrained settings and with other geometries, which show up, for example, in the probability simplex case corresponding to the multiplicative weights update method, whose mirror update you calculated above.

3. Accelerated Gradient Descent

In this problem, we will go through a proof of accelerated gradient descent by combining a gradient descent and a mirror descent step. You will also implement this algorithm in an accompanying jupyter notebook and use it to optimize some specific functions.

Recall that in lecture, in the proof of gradient descent for L-smooth functions, we proved the following inequality:

$$f(\vec{x}_+) \leq f(\vec{x}) - \frac{1}{2L}\|\nabla f(\vec{x})\|_2^2 \tag{11}$$

where $\vec{x}_+ = \vec{x} - \frac{1}{L}\nabla f(\vec{x})$ is the gradient descent step. We will need this inequality as well as the inequality you proved in part (e) in the Mirror Descent problem which bounds the per iteration regret.

Let's now describe an accelerated gradient descent algorithm. We will work in the unconstrained optimization setting so that $\mathcal{X} = \mathbb{R}^n$ and will use the ℓ^2 geometry and we assume f is convex and L-smooth.

Algorithm 2 Acceleration via Combining Gradient and Mirror Descent

$\vec{x}_0 = \vec{y}_0 = \vec{z}_0$ is a uniformly random point in $\mathcal{X}$
$k = 0$
while $k \leq T$ **do**
  $\vec{x}_{k+1} = \tau_k \vec{z}_k + (1-\tau_k)\vec{y}_k$ for $\tau_k = 2/(k+2)$
  $\vec{y}_{k+1} \leftarrow x_{k+1} - \frac{1}{L}\nabla f(\vec{x}_{k+1})$
  $\vec{z}_{k+1} = \text{Mirr}(\eta_{k+1}\nabla f(\vec{x}_{k+1}); \vec{z}_k)$ where $\eta_{k+1} = (k+2)/2L = 1/(\tau_k L)$
  $k \leftarrow k+1$
end while
  return y_T

Here the h function defining the Bregman divergence for the mirror descent step is just $h(\vec{x}) = \frac{1}{2}\|x\|_2^2$.

(a) We first understand the regret on the mirror update. Formally prove that,

$$\langle \eta_{k+1}\nabla f(\vec{x}_{k+1}), \vec{z}_k - \vec{u}\rangle \leq \frac{\eta_{k+1}^2}{2}\|\nabla f(\vec{x}_{k+1})\|_2^2 + D(\vec{u};\vec{z}_k) - D(\vec{u};\vec{z}_{k+1}) \tag{12}$$

HINT: In part (e) of the mirror descent question, would the proof still work if $\vec{g}_k$ was something other than $\nabla f(\vec{z}_k)$?

(b) Now, we try to understand the regret of $\vec{x}_{k+1}$. Formally, prove that,

$$\langle \eta_{k+1}\nabla f(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{u}\rangle \tag{13}$$

$$\leq \frac{(1-\tau_k)\eta_{k+1}}{\tau_k}\langle \nabla f(\vec{x}_{k+1}), \vec{y}_k - \vec{x}_{k+1}\rangle + \frac{\eta_{k+1}^2}{2}\|\nabla f(\vec{x}_{k+1})\|_2^2 + D(\vec{u};\vec{z}_k) - D(\vec{u};\vec{z}_{k+1}). \tag{14}$$

Furthermore, show that one can upper bound the RHS above by

$$\frac{(1-\tau_k)\eta_{k+1}}{\tau_k}(f(\vec{y}_k) - f(\vec{x}_{k+1})) + \eta_{k+1}^2 L(f(\vec{x}_{k+1}) - f(\vec{y}_{k+1})) + D(\vec{u};\vec{z}_k) - D(\vec{u};\vec{z}_{k+1}). \tag{15}$$

HINT: In the $\vec{x}_{k+1} - \vec{u}$ term, add and subtract $\vec{z}_k$ and use the previous part along with the definition of $\vec{x}_{k+1}$.

(c) Now deduce that

$$\eta_{k+1}^2 L f(\vec{y}_{k+1}) - (\eta_{k+1}^2 L - \eta_{k+1}) f(\vec{y}_k) - D(\vec{u}; \vec{z}_k) + D(\vec{u}; \vec{z}_{k+1}) \leq \eta_{k+1} f(\vec{u}) \tag{16}$$

HINT: You will need to use the specific values of η_k and τ_k as defined in the algorithm definition to observe some cancellations

(d) Now, summing up the inequality in the previous part and plugging in values for η_{k+1}, conclude that

$$f(\vec{y}_T) \leq f(\vec{x}^\star) + \frac{2L\|\vec{x}^\star - \vec{x}_0\|_2^2}{(T+1)^2} \tag{17}$$

(e) In the accompanying jupyter notebook, implement the above acceleration via gradient plus mirror descent step. Run the algorithm on a given low rank quadratic optimization problem. Report how the algorithm performs as compared to Gradient Descent, Adam, Adagrad algorithms.

a) $\langle \eta_k' \vec{g}_k', \vec{x}_k' - \vec{u} \rangle = \frac{\eta_k'^2 \|\vec{g}_k'\|_2^2}{2} + D_h(\vec{u}; \vec{x}_k') - D_h(\vec{u}; \vec{x}_{k+1}')$ by (e) for mirror descent.

$\vec{z}_{k+1} = \text{mirr}(\eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{z}_k) = \vec{z}_k - \eta_{k+1} \nabla f(\vec{x}_{k+1})$ [we used to have $\vec{x}_{k+1} = \vec{x}_k - \eta_k \vec{g}_k$ for mirror descent]

[thus, let $\vec{z}_{k+1} = \vec{x}_{k+1}'$, $\vec{z}_k = \vec{x}_k'$, $\eta_{k+1} = \eta_k'$, $\nabla f(\vec{x}_{k+1}) = \vec{g}_k'$]

Then, $\langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{z}_k - \vec{u} \rangle = \frac{\eta_{k+1}^2}{2} \|\nabla f(\vec{x}_{k+1})\|_2^2 + D(\vec{u}; \vec{z}_k) - D_h(\vec{u}; \vec{z}_{k+1})$

b) $\langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{u} \rangle = \langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{z}_k \rangle + \langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{z}_k - \vec{u} \rangle$

$= \langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \tau_k \vec{z}_k + (1-\tau_k)\vec{y}_k - \vec{z}_k \rangle + \frac{\eta_{k+1}^2}{2} \|\nabla f(\vec{x}_{k+1})\|_2^2 + D(\vec{u}; \vec{z}_k) - D(\vec{u}; \vec{z}_{k+1})$.

$= \eta_{k+1}(1-\tau_k) \langle \nabla f(\vec{x}_{k+1}), \vec{y}_k - \vec{x}_{k+1} \rangle + \frac{\eta_{k+1}^2}{2} \|\nabla f(\vec{x}_{k+1})\|_2^2 + D(\vec{u}; \vec{z}_k) - D(\vec{u}; \vec{z}_{k+1})$.

Then, since f is L-smooth, we have $f(\vec{y}_{k+1}) \leq f(\vec{x}_{k+1}) - \frac{1}{2L}\|\nabla f(\vec{x}_{k+1})\|_2^2$

[and by f being convex $f(\vec{y}_k) - f(\vec{x}_{k+1}) \geq \nabla f(\vec{x}_{k+1})^T (\vec{y}_k - \vec{x}_{k+1})$] $\Rightarrow L(f(\vec{x}_{k+1}) - f(\vec{y}_{k+1})) \geq \frac{\|\nabla f(\vec{x}_{k+1})\|_2^2}{2}$.

$\therefore \langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{u} \rangle \leq \eta_{k+1} \frac{(1-\tau_k)}{\tau_k} (f(\vec{y}_k) - f(\vec{x}_{k+1})) + \eta_{k+1}^2 L (f(\vec{x}_{k+1}) - f(\vec{y}_{k+1})) + D(\vec{u}; \vec{z}_k) - D(\vec{u}; \vec{z}_{k+1})$.

c) From the previous part,

$\langle \eta_{k+1} \nabla f(\vec{x}_{k+1}), \vec{x}_{k+1} - \vec{u} \rangle \leq (\eta_{k+1}^2 L - \eta_{k+1}) f(\vec{y}_k) - \eta_{k+1}^2 L f(\vec{y}_{k+1}) + D(\vec{u}; \vec{z}_k) - D(\vec{u}; \vec{z}_{k+1})$ [plugging in $\eta_{k+1} = \frac{1}{L}$]

$\Rightarrow \eta_{k+1} \langle \nabla f(\vec{x}_{k+1}), \vec{u} - \vec{x}_{k+1} \rangle \geq \eta_{k+1}^2 L f(\vec{y}_{k+1}) - (\eta_{k+1}^2 L - \eta_{k+1}) f(\vec{y}_k) - D(\vec{u}; \vec{z}_k) + D(\vec{u}; \vec{z}_{k+1})$ [flipping negatives]

$\Rightarrow \eta_{k+1} f(\vec{u}) \geq \eta_{k+1}^2 L f(\vec{y}_{k+1}) - (\eta_{k+1}^2 L - \eta_{k+1}) f(\vec{y}_k) - D(\vec{u}; \vec{z}_k) + D(\vec{u}; \vec{z}_{k+1})$

[$\nabla f(\vec{x}_{k+1})^T (\vec{u} - \vec{x}_{k+1}) \leq f(\vec{u})$ by first order condition for convexity of f].

d) $\sum_{k=0}^{T-1} \eta_{k+1}^2 L (f(\vec{y}_{k+1}) - f(\vec{y}_k)) + \sum_{k=0}^{T-1} \eta_{k+1} f(\vec{y}_k) - \sum_{k=0}^{T-1} D(\vec{u}; \vec{z}_k) + \sum_{k=1}^{T} D(\vec{u}; \vec{z}_k) \leq \sum_{k=0}^{T-1} \eta_{k+1} f(\vec{u})$

$\Rightarrow D(\vec{u}; \vec{z}_T) - D(\vec{u}; \vec{z}_0) + \sum_{k=0}^{T-1} \left[\frac{(k+2)^2}{4L} f(\vec{y}_{k+1}) - \frac{k^2}{4L} f(\vec{y}_k) \right] \leq f(\vec{u}) \sum_{k=0}^{T-1} \frac{k+2}{2L}$.

$\Rightarrow D(\vec{u}; \vec{z}_T) - D(\vec{u}; \vec{z}_0) + \frac{(T+1)^2}{4L} f(\vec{y}_T) + \sum_{k=1}^{T-1} \frac{2k+1}{4L} f(\vec{y}_k) \leq f(\vec{u}) \frac{T^2+3T}{4L}$

$\Rightarrow (T+1) f(\vec{y}_T) + \sum_{k=1}^{T-1} (2k+1) f(\vec{y}_k) \leq f(\vec{u})(T^2+3T) - 4L(D(\vec{u}; \vec{z}_T) - D(\vec{u}; \vec{z}_0))$

$\Rightarrow$ [$f(\vec{x}^*) \leq f(\vec{y}_k)$ $\forall k$] [$-4L D(\vec{u}; \vec{z}_T) \leq 0$]

$(T+1) f(\vec{y}_T) + (T-1) f(\vec{x}^*) \leq (T^2+3T) f(\vec{x}^*) + 4L D(\vec{x}^*; \vec{z}_0)$

$\Rightarrow f(\vec{y}_T) \leq f(\vec{x}^*) + \frac{L}{(T+1)^2} \|\vec{x}^* - \vec{x}_0\|_2^2$ [$\vec{z}_0 = \vec{x}_0$]

opt_algos

April 30, 2023

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import animation

from scipy.optimize import minimize, OptimizeResult
from collections import defaultdict
from itertools import zip_longest
from functools import partial

from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.inspection import DecisionBoundaryDisplay

from copy import deepcopy
```

```
/Users/Yash/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138:
UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version
of SciPy (detected version 1.23.5)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is
required for this version of "
```

0.1 Implementation (Students do)

0.1.1 Methods

You will implement five optimization algorithms (descriptions available here). - Gradient descent (`gd`) - MirrorDescent (`mirrD`) - Accelerated gradient method (`acc`) - Adaptive gradient method (`adagrad`) - Adaptive moment estimation (`adam`)

The last is a very common optimizer used in practical applications – possibly the most common in the world.

In addition, you will also implement a method for mirrorstep for the entropy regularizer and the ell_2 regularizer `mirrorstep_entropy/ell_2`. The latter of which you will use for the mirror descent algorithm and may also use for the accelerated method implementation. Make note of the function headers: `def gd(func, x, lr, num_iters, grad):`. - `func`: [type: function] The loss

function. Takes in a point of type np.ndarray (n,) for some n and returns a float representing the value of the function at that point. - `x`: [type: np.ndarray (n,)] The starting point of the optimization. - `lr`: [type: real] Learning rate. - `num_iters`: [type: int] The number of iterations of the optimization method to run. - `grad`: [type: function] The gradient of the loss function. Takes in a point of type np.ndarray (n,) and returns an np.ndarray (n,) representing the gradient of the function at that point.

Each function will need to return a `np.ndarray` containing all the iterates over the course of the optimization.

[335]:
```
#takes as input an x which is on the n-dimensional unit simplex, lr is a FIXED␣
 ↪SCALAR Learning Rate (not a schedule function like the rest of the problem)␣
 ↪and grad is a vector which returns gradient of a given function at point x.␣
 ↪Return the next iterate of the mirror step.
def mirrorstep_entropy(x, lr, grad):
    temp = x*np.exp(-lr*grad(x))
    return temp/np.sum(temp)

#same as above except x lies in R^n inside of the unit simplex.
def mirrorstep_ell_2(x, lr, grad):
    return x - lr*grad(x)

def gd(func, x, lr, num_iters, grad):
    iterates = []
    x = deepcopy(x)
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        update = -lr*grad(x) #change the update function
        x += update
        iterates.append(deepcopy(x))

    return np.array(iterates)

def mirrorD(func, x, lr, num_iters, grad):
    iterates = []
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        x = mirrorstep_ell_2(x, lr, grad)
        iterates.append(deepcopy(x))
    return np.array(iterates)

# we will not use a learning rate but rather use a parameter L which denotes␣
 ↪the Lipshitz constant of the function to be optimized. Use the stepsize/
 ↪learning rates you got from the PDF part of the problem. Return the y␣
 ↪sequence of iterates
def acc(func, x, L, num_iters, grad):
    y = deepcopy(x)
    z = deepcopy(x)
```

```
    iterates = []
    iterates.append(deepcopy(x))
    for k in range(num_iters):
        t = 2/(k+2)
        x = t*z + (1-t)*y
        y = x - (1/L)*grad(x)
        z -= grad(x)/(L*t)
        iterates.append(deepcopy(x))
    return np.array(iterates)

def adagrad(func, x, lr, num_iters, grad, eps=1e-5):
    iterates = []
    iterates.append(deepcopy(x))
    G = 0
    for itr in range(num_iters):
        G += grad(x)**2
        update = -lr*np.linalg.inv(np.diag(np.sqrt(np.diag(G + eps*np.eye(*x.
 ↪shape)))))@grad(x)
        x += np.reshape(update, newshape = x.shape)
        iterates.append(deepcopy(x))
    return np.array(iterates)

def adam(func, x, lr, num_iters, grad, beta1=0.9, beta2=0.999, eps=1e-5):
    m = 0
    v = 0
    b1 = beta1
    b2 = beta2
    iterates = []
    iterates.append(deepcopy(x))
    for i in range(1,num_iters):
        g = grad(x)
        m = beta1*m + (1-beta1)*g
        v = beta2*v + (1-beta2)*(g**2)
        mh = m/(1-b1)
        vh = v/(1 - b2)
        b1 = beta1**i
        b2 = beta2**i
        x = x - lr*mh/(vh**0.5 + eps)
        iterates.append(deepcopy(x))
    return np.array(iterates)
```

0.2 Testing your code

We are not providing much structure here, but now is a good time to make sure your optimization methods are working well. The cell below tests your gradient descent method on the function $f(x) = x^2$.

[307]:
```
x_squared_fval = lambda x: x**2
x_squared_grad = lambda x: 2*x
iterates = acc(x_squared_fval,np.array([2.0]),1.9,100,x_squared_grad)
res=[(i,x,x_squared_fval(x)) for (i,x) in enumerate(iterates)]
print([x[2] for x in res])
```

```
[array([4.]), array([4.]), array([0.01108033]), array([0.00110496]),
array([2.17659863e-05]), array([1.54525879e-06]), array([5.91094574e-08]),
array([3.80497322e-09]), array([1.94368141e-10]), array([1.23903412e-11]),
array([7.28716454e-13]), array([4.73378344e-14]), array([3.00528511e-15]),
array([2.00212986e-16]), array([1.33131958e-17]), array([9.08501578e-19]),
array([6.23142428e-20]), array([4.34174382e-21]), array([3.04581605e-22]),
array([2.15954976e-23]), array([1.54144065e-24]), array([1.10895806e-25]),
array([8.02618895e-27]), array([5.84528136e-28]), array([4.27930205e-29]),
array([3.14891217e-30]), array([2.32763342e-31]), array([1.72799324e-32]),
array([1.28788074e-33]), array([9.63427449e-35]), array([7.23188208e-36]),
array([5.4460942e-37]), array([4.11365092e-38]), array([3.11603536e-39]),
array([2.36665685e-40]), array([1.80202757e-41]), array([1.37536888e-42]),
array([1.05209124e-43]), array([8.06514794e-45]), array([6.19510707e-46]),
array([4.7678089e-47]), array([3.67605379e-48]), array([2.83922669e-49]),
array([2.19653639e-50]), array([1.70201788e-51]), array([1.3208293e-52]),
array([1.02649675e-53]), array([7.98858858e-55]), array([6.2252825e-56]),
array([4.85736891e-57]), array([3.79467005e-58]), array([2.9679532e-59]),
array([2.32397025e-60]), array([1.82169669e-61]), array([1.42947344e-62]),
array([1.12283027e-63]), array([8.82824739e-65]), array([6.94772194e-66]),
array([5.47272805e-67]), array([4.31464909e-68]), array([3.40450988e-69]),
array([2.68855658e-70]), array([2.1248475e-71]), array([1.68061966e-72]),
array([1.33025162e-73]), array([1.05368529e-74]), array([8.35201609e-76]),
array([6.6246962e-77]), array([5.25806803e-78]), array([4.17603135e-79]),
array([3.31872064e-80]), array([2.63900038e-81]), array([2.0997266e-82]),
array([1.67160484e-83]), array([1.33151289e-84]), array([1.0611862e-85]),
array([8.46186528e-87]), array([6.75092044e-88]), array([5.3886074e-89]),
array([4.30329664e-90]), array([3.43820633e-91]), array([2.74829543e-92]),
array([2.19781345e-93]), array([1.75836671e-94]), array([1.40739091e-95]),
array([1.12694391e-96]), array([9.02750835e-98]), array([7.23448199e-99]),
array([5.79985284e-100]), array([4.65149634e-101]), array([3.73190755e-102]),
array([2.99521487e-103]), array([2.40480879e-104]), array([1.93145787e-105]),
array([1.55181094e-106]), array([1.24720583e-107]), array([1.00272114e-108]),
array([8.06421284e-110]), array([6.48754998e-111]), array([5.22075857e-112]),
array([4.20259925e-113])]
```

0.3 Submission: Challenge

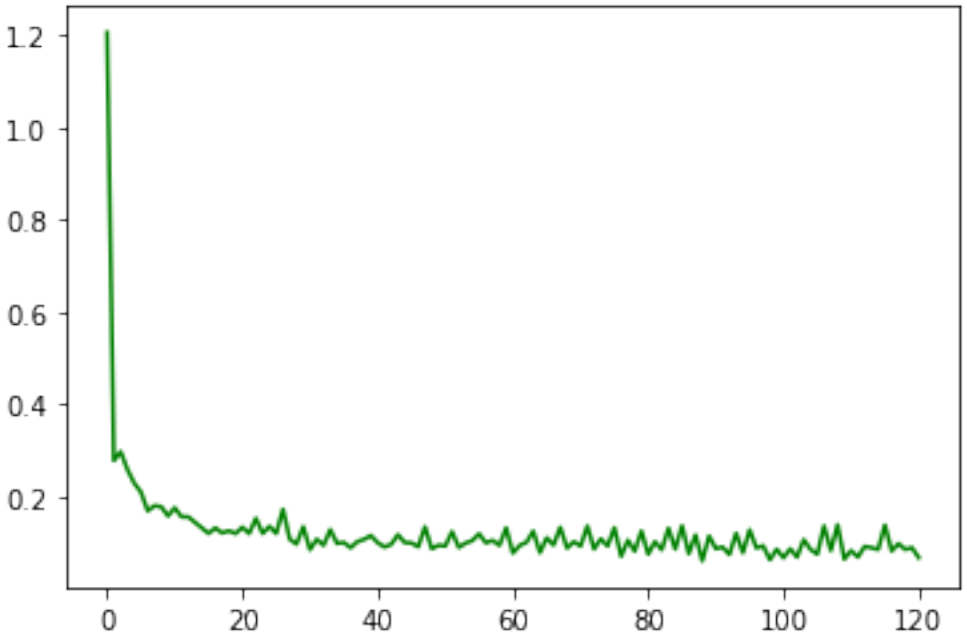
In this part, you will implement the lambda functions for the functional value and the gradient function for logistic regression function given a data matrix X and output vector y. Finally you will run the above algorithms that you implemented for a classification dataset.

[308]:
```
#1/n \sum_i y_i log(1/(1+e^(-x_i^t w)) + (1-y_i) log(1-1/(1+e^(-x_i^t w))
def logistic_regression_fval(X,y,w):
    p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
    return -np.mean(y*np.log(p+1e-3) + (1-y)*np.log(np.abs(1-p)+1e-3))
def logistic_regression_grad(X,y,w):
    p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
    return -X.T@(y - p)
```

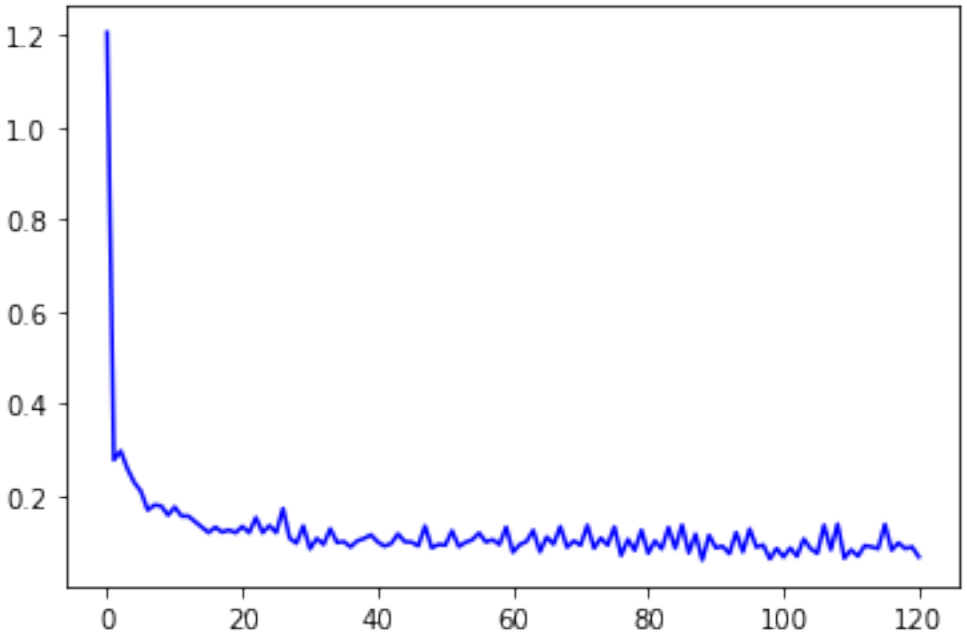
[404]:
```
#Logistic Regression Dataset
data=datasets.load_breast_cancer()
X=data.data
scaler=StandardScaler()
scaler.fit(X)
X=scaler.transform(X)
y=data.target
w0 = np.random.randn(X.shape[1])
maxitr = 120

iterates = gd(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resgd=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = mirrorD(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda␣
 ↪w:logistic_regression_grad(X,y,w))
resmd=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = acc(lambda w:logistic_regression_fval(X,y,w),w0,200,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resacc=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = adam(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resadam=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = adagrad(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda␣
 ↪w:logistic_regression_grad(X,y,w))
resadagrad=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in␣
 ↪enumerate(iterates)]
```

```
<ipython-input-308-790ea412211b>:6: RuntimeWarning: overflow encountered in exp
  p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
<ipython-input-308-790ea412211b>:3: RuntimeWarning: overflow encountered in exp
  p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
```

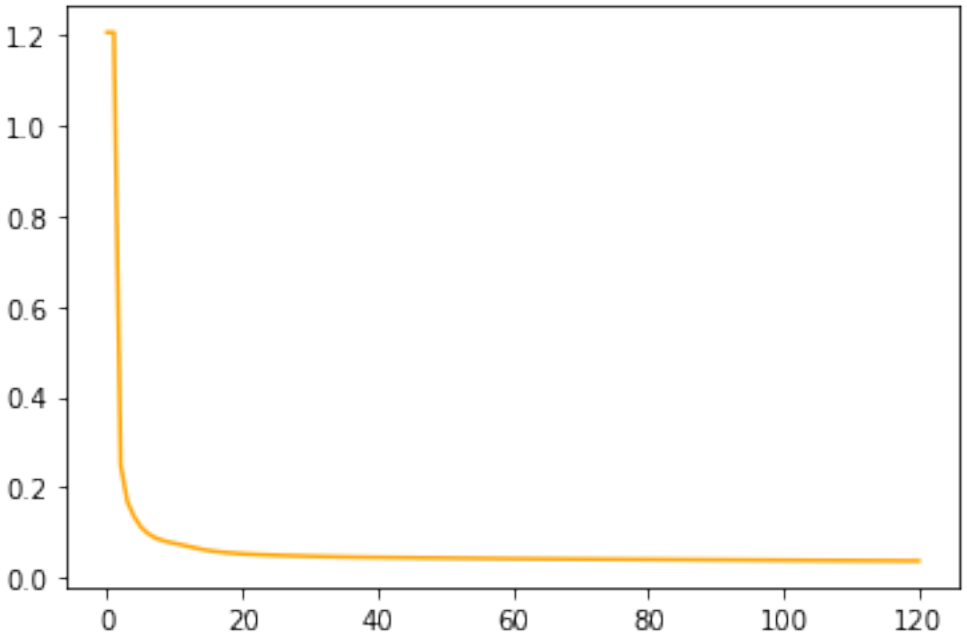
[406]:
```
a1, = plt.plot([x[2] for x in resgd], color = 'green', label = 'gradient␣
 ↪descent')
```



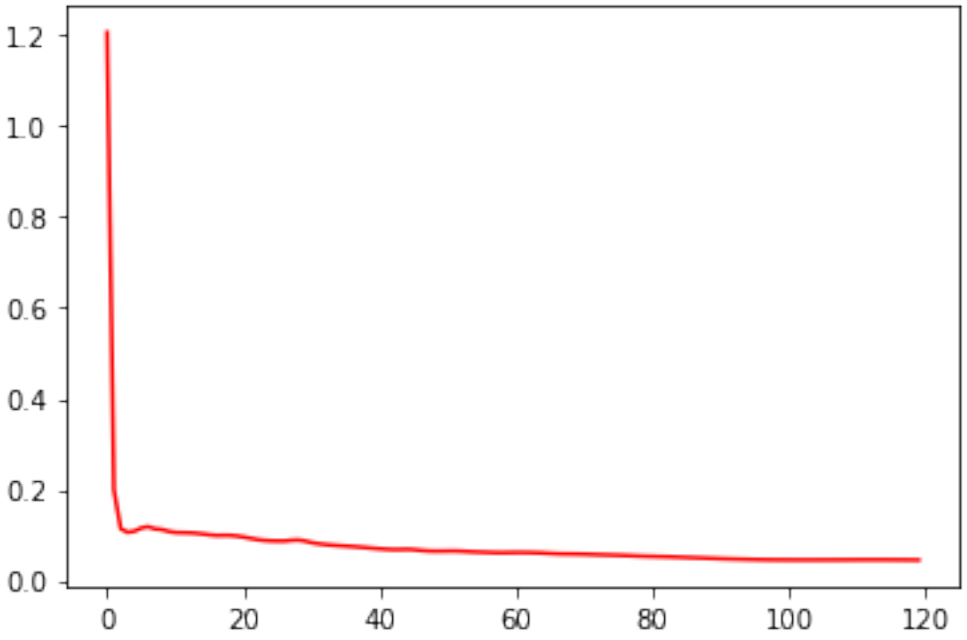
[407]:
```
a2, = plt.plot([x[2] for x in resmd], color = 'blue', label = 'mirror descent')
```



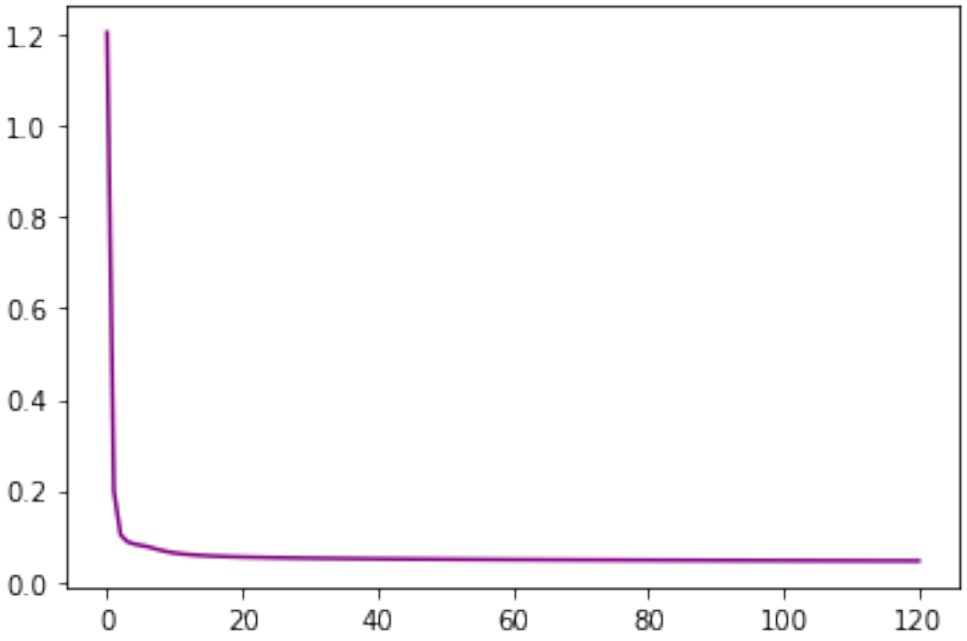
[408]:
```
a3, = plt.plot([x[2] for x in resacc], color = 'orange', label = 'accelerated␣
↪descent')
```



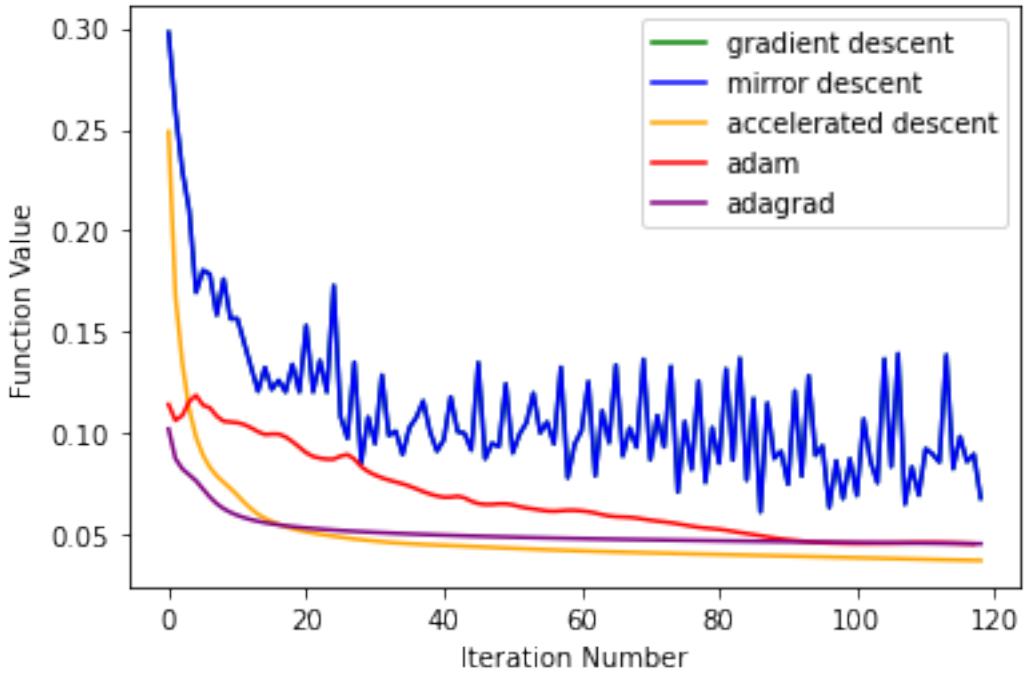
[405]:
```
a4, = plt.plot([x[2] for x in resadam], color = 'red', label = 'adam')
```



[409]:
```
a5, = plt.plot([x[2] for x in resadagrad], color = 'purple', label = 'adagrad')
```



[410]:
```
a1, = plt.plot([x[2] for x in resgd[2:]], color = 'green', label = 'gradient␣
 ↪descent')
a2, = plt.plot([x[2] for x in resmd[2:]], color = 'blue', label = 'mirror␣
 ↪descent')
a3, = plt.plot([x[2] for x in resacc[2:]], color = 'orange', label =␣
 ↪'accelerated descent')
a4, = plt.plot([x[2] for x in resadam[2:]], color = 'red', label = 'adam')
a5, = plt.plot([x[2] for x in resadagrad[2:]], color = 'purple', label =␣
 ↪'adagrad')
plt.legend(handles=[a1, a2, a3, a4, a5])

plt.xlabel('Iteration Number')
plt.ylabel('Function Value')
plt.show()
```



Run the above code for all the different algorithms and report the convergence behavior of all the algorithms. Add the plots of the function value decrease over all iterates to a PDF file (or add different cells for the run of each algorithm and the plotting and you can just use the PDF of the jupyter notebook.)

The main observations are as follows: - All algorithms are not used with the same learning rates. I kept 1 as the learning rate for all algorithms but I guess and checked what to use for accelerated descent, since it used the L constant instead of learning rate, and used 200 for it since it consistently gave a smooth and accurate curve. - Gradient descent and mirror descent are exactly the same because the choice of the h function (for divergence calculation) makes mirror descent and gradient descent have the same update step. They fluctuate highly towards the end but this changes with different learning rates. - Accelerated gradient descent works almost the same way as adagrad throughout, even achieving slightly higher accuracy after the first few iterations. Adam is close behind and catches up around 100 iterations. Acc descent has about the same speed as adagrad and even higher long-term accuracy than adam. Very impressive.

3 Rubric

Here's what the rubric looks like:

- To get a B: read and give a two-sentence summary for the papers in both problems and answer the questions associated to the two papers at the beginning of the problems, have a mostly correct implementation for problems 2(i) and 3(f), correctly solve any five parts from 2(a) — 2(h), and correctly solve any three parts from 3(a) — 3(e).
- To get a B+/A-: read and give a two-sentence summary for the papers in both problems, have a correct implementation for problems 2(i) and 3(f), correctly solve all of 2(a) — 2(h), and correctly solve all of 3(a) — 3(e).
- To get a A: complete all problems as stated above, plus one of the extensions below.

To get an A, you should complete one of the following extensions. You should include your extension(s) in a separate report that you attach to your project writeup. Exceptional projects that go above and beyond may receive extra credit at our discretion.

- Read the paper [4] and provide a summary of the key ideas of the SAM procedure, including especially why it may improve performance over gradient descent. Also, provide an implementation of the SAM procedure, including a performance evaluation on the benchmark functions used in the main project.
- Read the paper [5] and provide a summary of the key ideas of the optimal algorithm given in the paper, including especially the geometric intuition of what happens at each step of the algorithm. Also, provide an implementation of this algorithm, including a performance evaluation on the benchmark functions used in the main project.
- Read the paper [6] which shows how to solve min-max saddle point problems in special cases and summarize the algorithm and also why this algorithm (and the analysis) is called a mirror proximal algorithm. Implement the mirror prox algorithm in the special case which corresponds to Example 1 (page 17 of the paper) of matrix games.
- Propose your own extension of a similar level of interest and difficulty that you get approved by course staff.

3.1 Deliverables

Your submission should be in the form of one PDF with the following parts:

1. Your project writeup.
2. A PDF printout of the completed Jupyter notebook `main.ipynb`.
3. A report of any project extension you choose to complete. Your report should follow the template available on the course website under "Projects". The report must have a minimum of 1000 words and should include the following sections:
    - Introduction section: includes literature review of any relevant papers you summarized and identifies open problems in the understanding of multiplicative noise systems.
    - Methodology section: includes description of the methodology you follow in the project extension you chose to implement. Make sure you include description of the noise model you use and any parameters you use in the algorithm or in training.

- Results section: summarizes the results you obtained from the project extension you implement. You may use some of the visualizations from your project writeup for performance comparison.

In addition to the PDF submission, please do submit any code or Jupyter notebook you write for any project extension.

References

[1] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate o\bigl(k^2\bigr)," in *Doklady Akademii Nauk*, Russian Academy of Sciences, vol. 269, 1983, pp. 543–547.

[2] Y. Nesterov *et al.*, *Lectures on convex optimization*. Springer, 2018, vol. 137.

[3] Z. Allen-Zhu and L. Orecchia, "Linear coupling: An ultimate unification of gradient and mirror descent," *arXiv preprint arXiv:1407.1537*, 2014.

[4] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur, "Sharpness-aware minimization for efficiently improving generalization," *arXiv preprint arXiv:2010.01412*, 2020.

[5] S. Bubeck, Y. T. Lee, and M. Singh, "A geometric alternative to nesterov's accelerated gradient descent," *arXiv preprint arXiv:1506.08187*, 2015.

[6] A. Nemirovski, "Prox-method with rate of convergence o (1/t) for variational inequalities with lipschitz continuous monotone operators and smooth convex-concave saddle point problems," *SIAM Journal on Optimization*, vol. 15, no. 1, pp. 229–251, 2004.

A REPORT ON SHARPNESS-AWARE MINIMIZATION

ALEXANDER GE, OMER MOTON, AND YASH PANSARI

The paper is available here

1. Introduction

When training a model using regular gradient descent methods on a training set, sometimes the optimal point found using the training set does not translate into an optimal performance on the population set. When there are multiple close minimal points, one factor that tends to make a point better than another is the shape of the surrounding area. If one minimal point is surrounded by points with very high loss, in that this point is sharply protruding down, it is a sign that it could be a poorly chosen point, creating a noisy local minimum that fools SGD into thinking it is the global minimizer. Sharpness Aware Minimization (SAM) tries to solve this by considering the surrounding neighborhood — the intuition is that if the surrounding neighborhood has very low loss, then this point has a higher probability of working better on the population set.

The SAM algorithm performs better than regular gradient descent or stochastic gradient descent in most real world noisy cases, and at worst performs as well as batch gradient descent. Due to the nature of the algorithm however, less noisy functions do not benefit much from the SAM algorithm as in their case, every low-loss point will be surrounded by a low-loss neighbourhood. As such, the SAM algorithm performs considerably better on functions with a lot of noise, as the SAM algorithm can sift through outliers and find a better point.

2. Methodology

Our models will be mostly following the pseudocode on page 4 of the reading. Our algorithm assumes that the functions to be optimized must be bounded and continuous. The algorithm takes the training set, the loss function, the batch size, the step size, and the neighborhood size as input. We start by taking a random starting point x and a random batch from the training set, and we calculate the gradient of the loss function at x on our batch set, and then we compute the gradient approximation of the SAM function by using the approximation that the gradient of our SAM loss function at x is approximately the same as the gradient of the loss function evaluated at x plus ϵ which is a function of x. Then we can update by subtracting the gradient multiplied by step size from x. When $\rho = 0$ (the size of the neighbourhood), this process is theoretically identical to stochastic gradient descent. We also will test the SAM algorithm without batches, as the size of the training size is not extremeley large and will not be too time consuming. The issue with this is the problem of overfitting. When $\rho = 0$, this process is theoretically identical to regular gradient descent.

We will test the models tuning the hyperparameters of the size of the neighbourhood (rho), the step size, the batch size, as well as the norm value p. The paper indicates that the optimal value for $p = 2$, and that the optimal size of $\rho = 0.05$.

As mentioned above, the SAM algorithm performs better than the stochastic and regular gradient descent algorithm, so the upper bound on the efficacy of the algorithm is the upper bound on the regular gradient descent algorithm.

3. Results

In our findings, the SAM algorithm performed relatively similar to stochastic gradient descent in the best tests. $\rho = 0.05$ performed marginally better than the values above and below. Changing the value of p also did not seem to change the outcome by much, however, p = 2 seems to be the optimal value. We used a step size of 1 as we used that for SGD and it works well. For batch size, we tried relatively larger and smaller batch sizes but the smaller batch sizes work better, which matches with the fact that, according to our research, SGD performs best with mini batches.

To continue from the rest of the project, and for ease of comparison with the other optimizers, we ran SGD on the logistic regression for the breast cancer dataset. Here, training did not have much noise so the main benefit of the SAM algorithm is mostly unused. If we were training on a loss function with many "sharp" points, then we would have seen greater improvements. There are plots of our results attached below but more insight on our methodology and results can be gained from looking at the appendix with our code.

In Figure 1, a comparison of the SAM algorithm and the SGD algorithm is made and it can be seen that, despite not sharing the same paths, both algorithms converge in roughly the same number of steps.

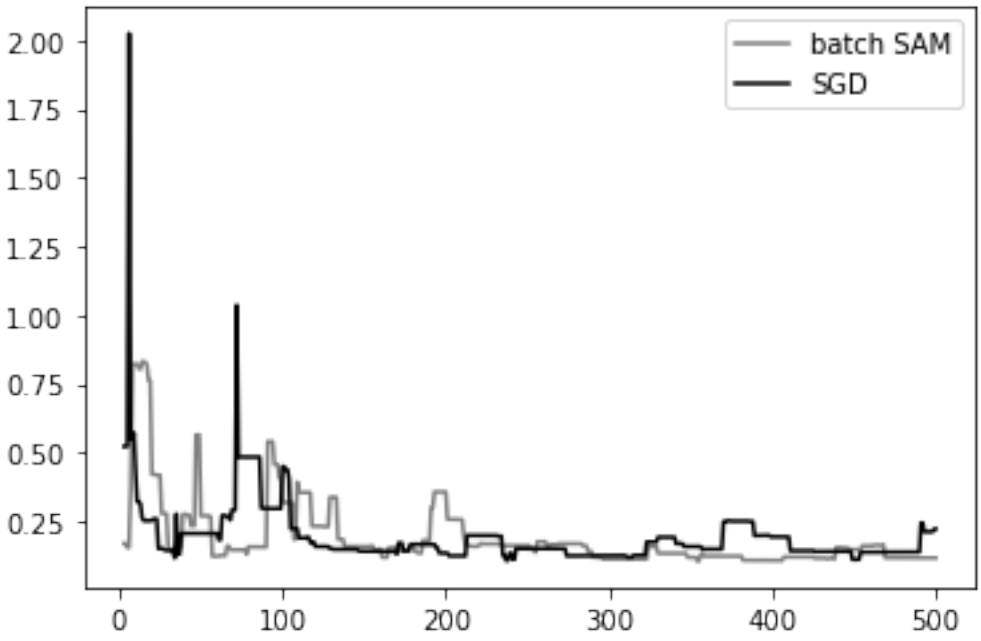


Figure 1. Comparison of SAM and SGD Algorithm on a Logistic Regression

In Figure 2, a comparison of the SAM algorithm that was run on the entire training set and not on batches against multiple different algorithms ran on whole training sets.

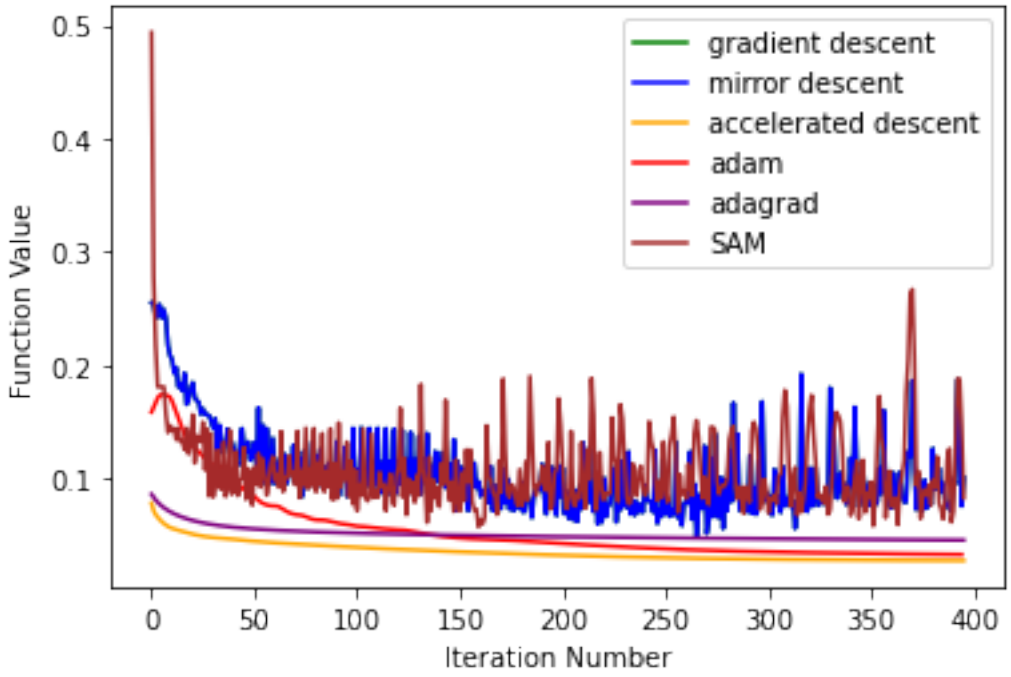


FIGURE 2. Comparison of SAM and Other Algorithms ran on a Logistic Regression Ran on Entire Training Sets

Appendix: Extension Code Implementation

May 2, 2023

[69]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

import random

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import animation

from scipy.optimize import minimize, OptimizeResult
from collections import defaultdict
from itertools import zip_longest
from functools import partial

from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.inspection import DecisionBoundaryDisplay

from copy import deepcopy
```

0.1 Control Algorithms Implementation

0.1.1 Methods

Here, I implemented five optimization algorithms (descriptions available here). - Gradient descent (`gd`) - MirrorDescent (`mirrD`) - Accelerated gradient method (`acc`) - Adaptive gradient method (`adagrad`) - Adaptive moment estimation (`adam`) so that I can compare them to my implementation of the paper's SAM and SGD.

Make note of the function headers: `def gd(func, x, lr, num_iters, grad):`. - `func`: [type: function] The loss function. Takes in a point of type np.ndarray (n,) for some n and returns a float representing the value of the function at that point. - `x`: [type: np.ndarray (n,)] The starting point of the optimization. - `lr`: [type: real] Learning rate. - `num_iters`: [type: int] The number of iterations of the optimization method to run. - `grad`: [type: function] The gradient of the loss function. Takes in a point of type np.ndarray (n,) and returns an np.ndarray (n,) representing the gradient of the function at that point.

Each function returns a `np.ndarray` containing all the iterates over the course of the optimization.

[4]:
```
def mirrorstep_entropy(x, lr, grad):
    temp = x*np.exp(-lr*grad(x))
    return temp/np.sum(temp)

#same as above except x lies in R^n inside of the unit simplex.
def mirrorstep_ell_2(x, lr, grad):
    return x - lr*grad(x)

def gd(func, x, lr, num_iters, grad):
    iterates = []
    x = deepcopy(x)
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        update = -lr*grad(x)
        x += update
        iterates.append(deepcopy(x))

    return np.array(iterates)

def mirrorD(func, x, lr, num_iters, grad):
    iterates = []
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        x = mirrorstep_ell_2(x, lr, grad)
        iterates.append(deepcopy(x))
    return np.array(iterates)

# we will not use a learning rate but rather use a parameter L which denotes the␣
 ↪Lipshitz constant of the function to be optimized.
def acc(func, x, L, num_iters, grad):
    y = deepcopy(x)
    z = deepcopy(x)
    iterates = []
    iterates.append(deepcopy(x))
    for k in range(num_iters):
        t = 2/(k+2)
        x = t*z + (1-t)*y
        y = x - (1/L)*grad(x)
        z -= grad(x)/(L*t)
        iterates.append(deepcopy(x))
    return np.array(iterates)

def adagrad(func, x, lr, num_iters, grad, eps=1e-5):
    iterates = []
    iterates.append(deepcopy(x))
    G = 0
    for itr in range(num_iters):
```

```
        G += grad(x)**2
        update = -lr*np.linalg.inv(np.diag(np.sqrt(np.diag(G + eps*np.eye(*x.
↪shape)))))@grad(x)
        x += np.reshape(update, newshape = x.shape)
        iterates.append(deepcopy(x))
    return np.array(iterates)

def adam(func, x, lr, num_iters, grad, beta1=0.9, beta2=0.999, eps=1e-5):
    m = 0
    v = 0
    b1 = beta1
    b2 = beta2
    iterates = []
    iterates.append(deepcopy(x))
    for i in range(1,num_iters):
        g = grad(x)
        m = beta1*m + (1-beta1)*g
        v = beta2*v + (1-beta2)*(g**2)
        mh = m/(1-b1)
        vh = v/(1 - b2)
        b1 = beta1**i
        b2 = beta2**i
        x = x - lr*mh/(vh**0.5 + eps)
        iterates.append(deepcopy(x))
    return np.array(iterates)
```

0.2 Testing your code

The cell below tests the accelerated gradient descent on the function $f(x) = x^2$ as a sanity check.

[5]:
```
x_squared_fval = lambda x: x**2
x_squared_grad = lambda x: 2*x
iterates = acc(x_squared_fval,np.array([2.0]),1.9,100,x_squared_grad)
res=[(i,x,x_squared_fval(x)) for (i,x) in enumerate(iterates)]
print([x[2] for x in res])
```

```
[array([4.]), array([4.]), array([0.01108033]), array([0.00110496]),
array([2.17659863e-05]), array([1.54525879e-06]), array([5.91094574e-08]),
array([3.80497322e-09]), array([1.94368141e-10]), array([1.23903412e-11]),
array([7.28716454e-13]), array([4.73378344e-14]), array([3.00528511e-15]),
array([2.00212986e-16]), array([1.33131958e-17]), array([9.08501578e-19]),
array([6.23142428e-20]), array([4.34174382e-21]), array([3.04581605e-22]),
array([2.15954976e-23]), array([1.54144065e-24]), array([1.10895806e-25]),
array([8.02618895e-27]), array([5.84528136e-28]), array([4.27930205e-29]),
array([3.14891217e-30]), array([2.32763342e-31]), array([1.72799324e-32]),
array([1.28788074e-33]), array([9.63427449e-35]), array([7.23188208e-36]),
array([5.4460942e-37]), array([4.11365092e-38]), array([3.11603536e-39]),
array([2.36665685e-40]), array([1.80202757e-41]), array([1.37536888e-42]),
```

```
array([1.05209124e-43]), array([8.06514794e-45]), array([6.19510707e-46]),
array([4.7678089e-47]), array([3.67605379e-48]), array([2.83922669e-49]),
array([2.19653639e-50]), array([1.70201788e-51]), array([1.3208293e-52]),
array([1.02649675e-53]), array([7.98858858e-55]), array([6.2252825e-56]),
array([4.85736891e-57]), array([3.79467005e-58]), array([2.9679532e-59]),
array([2.32397025e-60]), array([1.82169669e-61]), array([1.42947344e-62]),
array([1.12283027e-63]), array([8.82824739e-65]), array([6.94772194e-66]),
array([5.47272805e-67]), array([4.31464909e-68]), array([3.40450988e-69]),
array([2.68855658e-70]), array([2.1248475e-71]), array([1.68061966e-72]),
array([1.33025162e-73]), array([1.05368529e-74]), array([8.35201609e-76]),
array([6.6246962e-77]), array([5.25806803e-78]), array([4.17603135e-79]),
array([3.31872064e-80]), array([2.63900038e-81]), array([2.0997266e-82]),
array([1.67160484e-83]), array([1.33151289e-84]), array([1.0611862e-85]),
array([8.46186528e-87]), array([6.75092044e-88]), array([5.3886074e-89]),
array([4.30329664e-90]), array([3.43820633e-91]), array([2.74829543e-92]),
array([2.19781345e-93]), array([1.75836671e-94]), array([1.40739091e-95]),
array([1.12694391e-96]), array([9.02750835e-98]), array([7.23448199e-99]),
array([5.79985284e-100]), array([4.65149634e-101]), array([3.73190755e-102]),
array([2.99521487e-103]), array([2.40480879e-104]), array([1.93145787e-105]),
array([1.55181094e-106]), array([1.24720583e-107]), array([1.00272114e-108]),
array([8.06421284e-110]), array([6.48754998e-111]), array([5.22075857e-112]),
array([4.20259925e-113])]
```

0.3 SAM vs other algos

Here, I implemented SAM, mini-batch SAM and mini-batch SGD. Then, I implemented loss and loss gradient functions for a logistic regression. Then, I loaded a standard breast cancer classification dataset and fit a logistic regression to this using each of the optimizers implemented above (without batches). I kept iterations, learning rate, random initialization, etc. constant throughout to ensure fairness in comparison. Then, I made plots to compare the performance of SAM to all the other optimizers. However, it is clear from the paper that SAM was meant to be implemented with training in batches so I also used a mini-batch version and compared it to SGD at the end. In both cases, we see a first plot showing convergence from initialization first and then another plot skipping the first few iterates so we can see the later convergence comparisons more clearly.

[188]:
```python
def sam(func, x, lr, num_iters, grad, rho = 0.05, small = 1e-7, p = 2):
    iterates = []
    x = deepcopy(x)
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        g = grad(x)
        sign = g/(np.abs(g))
        q = p/(p-1)
        eps = rho*(sign)*(np.abs(g)**(q-1))/(np.linalg.norm(g, ord = q)**(q/p))
        samg = grad(x + eps)
        x = x - lr*samg
        iterates.append(deepcopy(x))
    return np.array(iterates)
```

```
def batchsam(func, x, lr, num_iters, grad, X=X, y = y, m = 32, rho = 0.05, small␣
↪= 1e-7, p = 2):
    iterates = []
    x = deepcopy(x)
    iterates.append(deepcopy(x))
    for itr in range(num_iters):
        rows_id = np.random.choice(X.shape[0]-1, m, replace = False)
        X_b = X[rows_id, :]
        y_b = y[rows_id]
        g = grad(X_b, y_b, x)
        sign = g/(np.abs(g) + small)
        q = p/(p-1)
        eps = rho*(sign)*(np.abs(g)**(q-1))/(np.linalg.norm(g, ord = q)**(q/p) +␣
↪small)
        samg = grad(X_b, y_b, x + eps)
        x = x - lr*samg
        iterates.append(deepcopy(x))
    return np.array(iterates)

def sgd(func, w, lr, num_iters, grad, X=X, y=y, m=32): #minibatch sgd
    iterates = []
    w = deepcopy(w)
    iterates.append(deepcopy(w))
    for itr in range(num_iters):
        rows_id = np.random.choice(X.shape[0]-1, m, replace = False)
        X_b = X[rows_id, :]
        y_b = y[rows_id]
        w = w - lr*grad(X_b, y_b, w)
        iterates.append(deepcopy(w))
    return np.array(iterates)
```

[189]:
```
x_squared_fval = lambda x: x**2
x_squared_grad = lambda x: 2*x
iterates = sam(x_squared_fval,np.array([2.0]),0.5,100,x_squared_grad)
res=[(i,x,x_squared_fval(x)) for (i,x) in enumerate(iterates)]
print([x[2] for x in res])
```

```
[array([4.]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
```

```
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025]), array([0.0025]), array([0.0025]), array([0.0025]),
array([0.0025])]
```

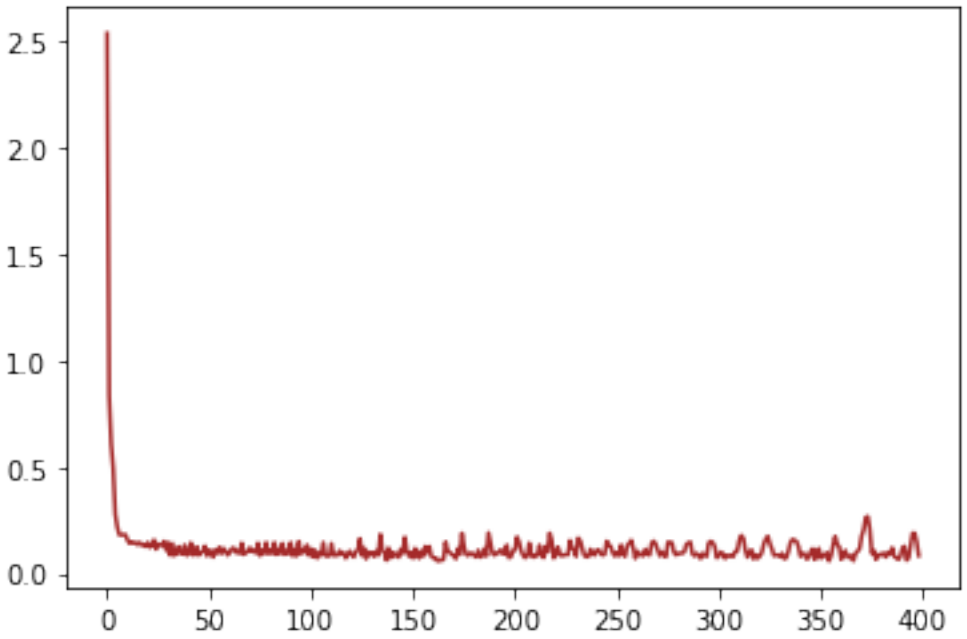
[190]:

```
#Logistic Regression Dataset
data=datasets.load_breast_cancer()
X=data.data
scaler=StandardScaler()
scaler.fit(X)
X=scaler.transform(X)
y=data.target
w0 = np.random.randn(X.shape[1])
maxitr = 400

iterates = gd(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resgd=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = mirrorD(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resmd=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = acc(lambda w:logistic_regression_fval(X,y,w),w0,200,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resacc=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = adam(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resadam=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]

iterates = adagrad(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
resadagrad=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in␣
 ↪enumerate(iterates)]
```

```
iterates = sam(lambda w:logistic_regression_fval(X,y,w),w0,1,maxitr,lambda w:
 ↪logistic_regression_grad(X,y,w))
ressam=[(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]
plt.plot([x[2] for x in ressam[2:]], color = 'brown', label = 'SAM')
```

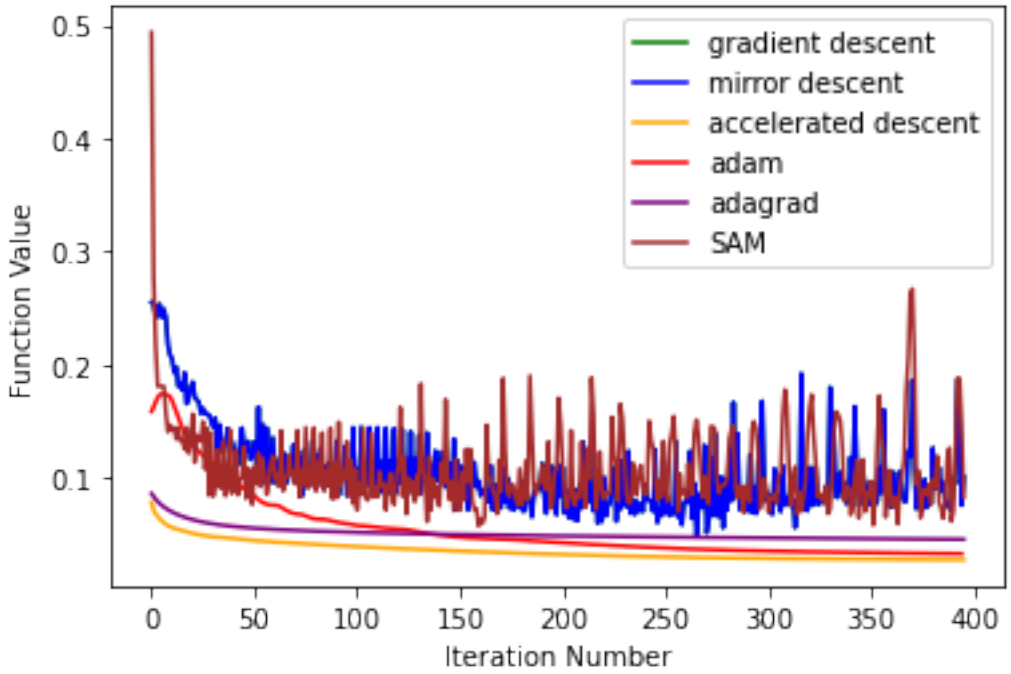
```
<ipython-input-6-790ea412211b>:6: RuntimeWarning: overflow encountered in exp
  p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
<ipython-input-6-790ea412211b>:3: RuntimeWarning: overflow encountered in exp
  p = np.array([1/(1+np.exp(-X[i].T@w)) for i in range(X.shape[0])])
```

[190]:
```
[<matplotlib.lines.Line2D at 0x7fc8c88ff6a0>]
```



[191]:
```
start = 5
a1, = plt.plot([x[2] for x in resgd[start:]], color = 'green', label = 'gradient␣
 ↪descent')
a2, = plt.plot([x[2] for x in resmd[start:]], color = 'blue', label = 'mirror␣
 ↪descent')
a3, = plt.plot([x[2] for x in resacc[start:]], color = 'orange', label =␣
 ↪'accelerated descent')
a4, = plt.plot([x[2] for x in resadam[start:]], color = 'red', label = 'adam')
a5, = plt.plot([x[2] for x in resadagrad[start:]], color = 'purple', label =␣
 ↪'adagrad')
a6, = plt.plot([x[2] for x in ressam[start:]], color = 'brown', label = 'SAM')
plt.legend(handles=[a1, a2, a3, a4, a5, a6])
```

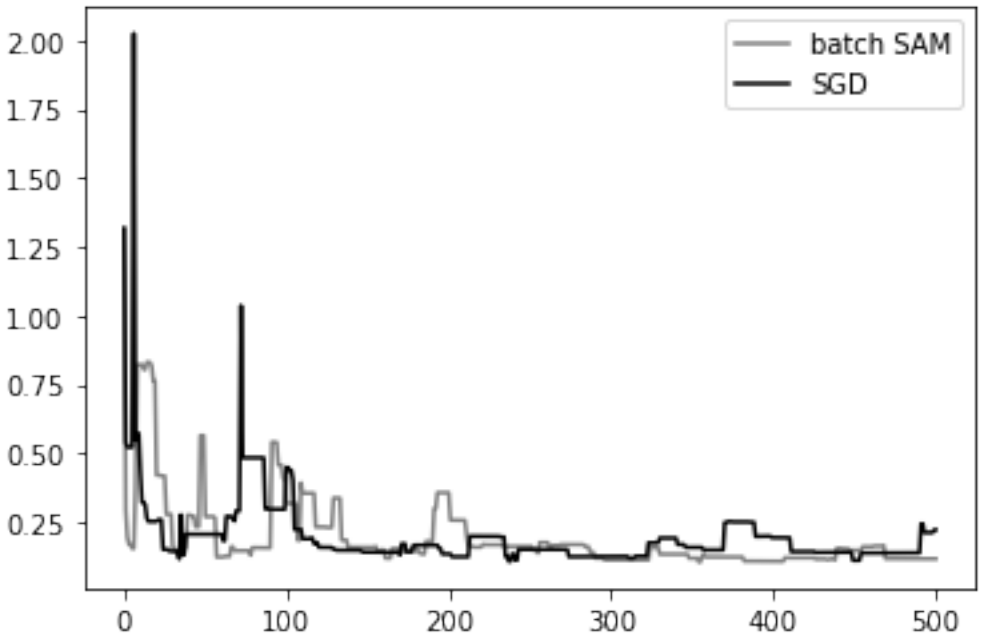
```
plt.xlabel('Iteration Number')
plt.ylabel('Function Value')
plt.show()
```



[202]:
```
#Logistic Regression Dataset
data=datasets.load_breast_cancer()
X=data.data
scaler=StandardScaler()
scaler.fit(X)
X=scaler.transform(X)
y=data.target
w0 = np.random.randn(X.shape[1])
maxitr = 500
batch_size = 4

iterates =␣
 ↪batchsam(logistic_regression_fval,w0,1,maxitr,logistic_regression_grad, m =␣
 ↪batch_size)
resbsam = [(i,w,logistic_regression_fval(X,y,w)) for (i,w) in␣
 ↪enumerate(iterates)]
a7, = plt.plot([x[0] for x in resbsam],[x[2] for x in resbsam], color = 'grey',␣
 ↪label = 'batch SAM')
```

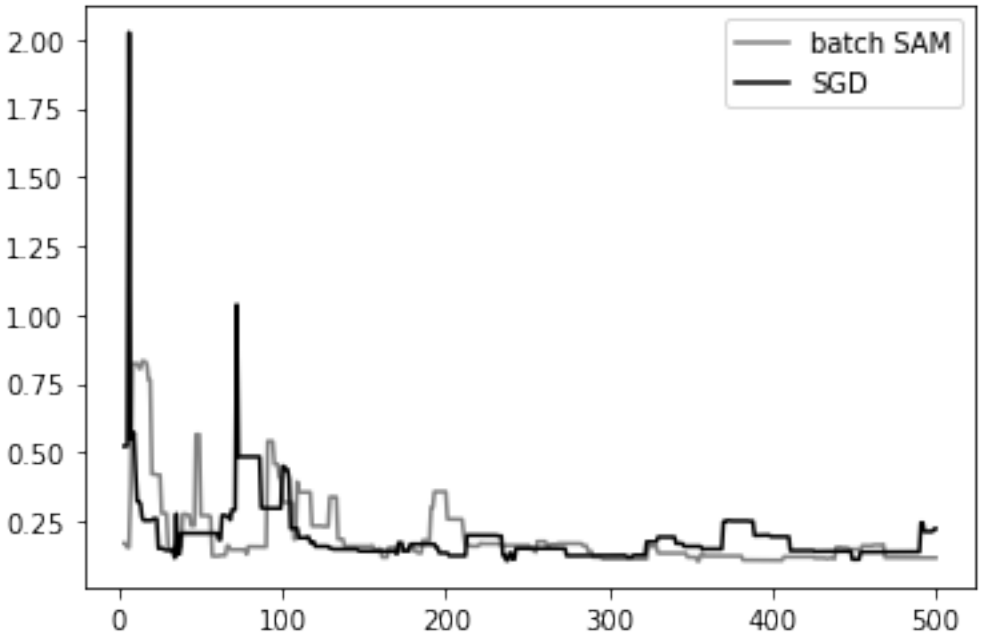
```
iterates = sgd(logistic_regression_fval,w0,1,maxitr,logistic_regression_grad, m␣
↪= batch_size)
ressgd = [(i,w,logistic_regression_fval(X,y,w)) for (i,w) in enumerate(iterates)]
a8, = plt.plot([x[0] for x in ressgd],[x[2] for x in ressgd], color = 'black',␣
↪label = 'SGD')
plt.legend(handles=[a7, a8])
```

[202]: <matplotlib.legend.Legend at 0x7fc8b0f6d040>



[203]:
```
start = 3
a9, = plt.plot([x[0] for x in resbsam[start:]],[x[2] for x in resbsam[start:]],␣
↪color = 'grey', label = 'batch SAM')
a10, = plt.plot([x[0] for x in ressgd[start:]],[x[2] for x in ressgd[start:]],␣
↪color = 'black', label = 'SGD')
plt.legend(handles=[a9, a10])
```

[203]: <matplotlib.legend.Legend at 0x7fc8c8eb2d30>


batch SAM
SGD
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
100
200
300
400
500