AMSC 664 Final Report: Escaping from Saddle Points Using Asynchronous Coordinate Descent

Marco Bornstein
Advisor: Dr. Furong Huang

May 13, 2021

1 Introduction

First-order gradient based methods are widely used in large-scale optimization problems due to their computational efficiency compared with higher-order methods, such as Hessian-based methods. In modern machine learning, and beyond, the objective functions we aim to optimize are usually non-convex. First-order gradient based methods are known to converge to first-order stationary points [20], i.e., points with a gradient value of zero. This includes local optima, global optima and saddle points in non-convex functions. Therefore, when applied in non-convex settings, gradient descent may converge to saddle points as well as local or global optima.

Dissimilar from local or global optima, saddle points are to be avoided. For many non-convex problems, all local minima are close in value to the global minimum (dictionary learning [26], matrix sensing and completion [5, 12, 23], and tensor decomposition [11] problems are examples of this). Therefore, convergence to local minima is sufficient for finding a solution to these problems [8]. In comparison, saddle points may produce highly sub-optimal solutions to a given optimization problem, such as training a deep neural network [10]. No research has quantified the frequency of saddle points in deep neural networks. However, the presence of scaling and perturbation symmetries in the parameter space is directly correlated to the proliferation of saddle points on the error surface in deep neural-networks [3, 10, 25].

Important optimization problems, such as training deep neural-networks, are increasing in dimension. To solve such high-dimensional optimization problems, it is necessary to create and apply an efficient algorithm. Coordinate-descent algorithms are built to handle large-scale optimization problems efficiently [21, 28, 30]. Within coordinate descent, the parameters of the model are split up into coordinate blocks between multiple workers. To train the model, each worker must compute its corresponding block of gradients. The full gradient of the model is attained only after each worker's gradient block is computed and returned.

1.1 Problem Formulation

Converging to second-order stationary points (local or global minima) is a necessary condition to minimize non-convex optimization problems. Previously, only algorithms utilizing Hessian information were thought to converge to second-order stationary points (such as cubic regularization or trust region algorithms). These algorithms require computing the inverse of the Hessian in each iteration, which is computationally expensive

[13, 22]. With high-dimensional problems, it is computationally infeasible to use second-order methods to minimize them. To remain computationally feasible, minimization must be done via first-order methods.

The sheer size of large-scale optimization problems requires multiple workers to solve it. Even with the help of parallel computing, where data is broken up between multiple workers to be worked on simultaneously, the increase in computational speed comes with the corresponding brake of synchronization issues. Commonly used optimization algorithms, such as stochastic gradient descent, operate under synchronization.

The requirement of synchronization, however, is impractical. Each worker is required to send its computed gradient values back to a global server before the global solution is updated. Only when the global solution is updated can each worker receive its next batch of data. Therefore, the speed of parallel computing is limited by the weakest link in the computational chain: by the slowest worker and its consequent longest communication delay. Worse, if a worker stops working or experiences a network connection failure, then the parallel computing process pauses. In many applications, an asynchronous process can be implemented to train large-scale problems efficiently.

The goal of this project is to implement a first-order asynchronous-coordinate-descent algorithm that I have constructed in parallel, and analyze its convergence. This algorithm is entitled the Saddle Escaping Asynchronous Coordinate Descent (SEACD) algorithm. I will be implementing SEACD in a parallel manner using Python, and compare it to another common algorithm: serial gradient descent (GD). I aim to show that parallel SEACD successfully minimizes non-convex functions, and does so in a quick and efficient manner.

1.2 Related Works

Convergence of First-order Algorithms. As mentioned in Section 1.1, it was previously believed that only second-order optimization methods would converge to second-order stationary points. More recent work has shown that there are lower-order, and more efficient, methods capable of converging to second-order stationary points [11, 17]. In these works, stochastic gradient descent is shown to converge to second-order stationary points, but with a large dependence on the dimension (high degree polynomial dependence). This result was greatly improved upon in the work of Jin et al. [14, 15, 16]. In both [14, 15], perturbed stochastic and regular gradient descent converge to second-order stationary points in poly-logarithmic steps with respect to dimension. Impressively, this result mirrors the convergence rate to first-order stationary points (disregarding logarithmic factors). My work matches their result, as SEACD also relies poly-logarithmically on dimension.

Another key aspect from [14, 16, 15], is that their results relied upon a new characterization of the geometry around saddle points. It is shown that the region where points get stuck around a saddle point during the course of gradient descent is quite thin. My work takes inspiration from this geometric technique to bound the volume of this region and show that a random perturbation from a saddle point is unlikely to fall in this stuck region.

Other simple first-order procedures that efficiently find escaping direction from saddle points include NEON within [1] and [32]. The work of [32] is inspired by the perturbed gradient method proposed in [14] and its connection with the power method for computing the largest eigenvector of a matrix starting from a random noise vector. In [1], negative-curvature-search subroutines are converted into first-order processes. My algorithm is complementary of this line of work, as one could combine these careful estimations of negative curvature into my algorithm. One focus of my work, rather than designing sophisticated procedures to improve the rate of escaping from saddle point, is to provide a simple algorithm that is resilient to delayed gradients.

Convergence of Coordinate-Descent Algorithms. First mentioned by [24], coordinate descent (CD) has convergence difficulties in non-convex settings. Powell's finding that cyclic CD fails to converge to a stationary point illustrates that a general convergence result for non-convex functions cannot be expected. However, Powell also mentions that the cyclic behavior of CD is unstable with respect to small perturbations. To solve this issue, I follow the work of [27] in using inexact line searches to lead to convergence of CD.

The main theorem of [27] states that a convex or *non-convex* L-gradient Lipschitz function (defined below), with a finite minimum value, converges to a stationary point at a rate inversely proportional to the square root of the total number of asynchronous coordinate descent iterations (asynchronous coordinate descent is detailed in Section 2.1). It is assumed in this theorem that there is a deterministic block rule. This means that each block of coordinates (detailed in Section 2.1) is guaranteed to be updated at least once within a given number of iterations. My work follows this theorem closely, as I also assume that the function is L-gradient Lipschitz and has a finite minimum value. Further, I assume a deterministic block rule with a window size equal to the maximum bounded delay τ (this is also detailed further below in Section 2.1).

Other methods and assumptions for attaining convergence results are discussed in [30]. These include assuming unique minimizers along any coordinate direction [4] and using functions that satisfy the Kurdyka-Łojasiewicz (KL) property [2, 31].

Convergence of Asynchronous Algorithms. This paper follows the theory behind asynchronous coordinate descent. A major portion of the theory, on asynchronous algorithms, has been built from [9, 18, 19]. In [19], the convergence properties of an asynchronous-stochastic-coordinate-descent algorithm is nailed down for a convex function. Similar work is applied in [18], with convergence results for an asynchronous-stochastic-proximal-coordinate-descent algorithm. This algorithm has an improved complexity compared to similar asynchronous algorithms. Convergence analysis of asynchronous block coordinate descent in [18, 19] relied upon the independence assumption as well as bounded delays. This work also incorporates the use of bounded delays. The work in [9] uses a similar asynchronous, but accelerated, stochastic-coordinate-descent algorithm. This work provides an even further speed up in convergence for convex functions than in [18, 19].

The work of [9, 18, 19] led to expanding research about asynchronous-coordinate-descent convergence within non-convex settings. In more closely related research to mine, the convergence properties of asynchronous coordinate descent in non-convex settings has been studied in [7, 27]. Within the important work of [27], asynchronous-coordinate-descent convergence results are determined for bounded, stochastic unbounded, and deterministic unbounded delays. These results are provided for both convex and non-convex functions. Furthermore, within [27] it is shown that there is sufficient descent for bounded delays by using a Lyapunov function for a non-convex function. As stated above, [27] proved that asynchronous coordinate descent converges to stationary points for L-gradient Lipschitz non-convex functions (defined below). This work utilizes the sufficient descent and Lyapunov function improvements found in [27].

Definition 1. *A differentiable function f is **L-smooth (or L-gradient Lipschitz)** if:*

$$\forall x_1, x_2,\ ||\nabla f(x_1) - \nabla f(x_2)|| \leq L||x_1 - x_2||$$

Definition 2. *A twice-differentiable function f is **ρ-Hessian Lipschitz** if:*

$$\forall x_1, x_2,\ ||\nabla^2 f(x_1) - \nabla^2 f(x_2)|| \leq \rho||x_1 - x_2||$$

Definition 3. *For a ρ-Hessian Lipschitz function f, we call x a **second-order stationary point** if:*

$$||\nabla f(x)|| = 0, \text{ and } \lambda_{min}(\nabla^2 f(x)) \geq 0$$

*We call x an **ϵ-second-order stationary point** if:*

$$||\nabla f(x)|| \leq \epsilon, \text{ and } \lambda_{min}(\nabla^2 f(x)) \geq -\sqrt{\rho\epsilon}$$

1.3 Project Goals

- Implement SEACD (as well as its subroutines SWACD, GACD, and PACD) in Python from scratch
  - Optimize the selection of hyperparameters
  - Compare the convergence results of SEACD to gradient descent (GD) and perturbed gradient descent (PGD) algorithms defined in [14] and [16]
- Implement an asynchronous-like serial version of SEACD
- Test and analyze the convergence of serial-asynchronous SEACD
  - Show that serial-asynchronous SEACD converges to second-order stationary points
  - Compare the convergence results of serial-asynchronous SEACD to gradient descent
- Create a Python Module for parallel SEACD from scratch
- Test and analyze the convergence of parallel SEACD
  - Show that parallel SEACD converges to second-order stationary points
  - Compare the convergence results of parallel SEACD to serial gradient descent (GD) and display a speed-up in runtime of the SEACD algorithm compared to GD

2 Approach

As mentioned in the Introduction, my approach to this project is split into two pieces: asynchronous coordinate descent and escaping saddle points. Below I will briefly describe each piece.

2.1 Asynchronous Coordinate Descent

In asynchronous coordinate descent, the solution vector is split up into a block of coordinates and worked on by a set of workers. Each worker continually updates the solution vector, one block at a time, leaving all other blocks unchanged. In this project we assume that each worker works on the same block of coordinates of the solution vector. Each block update is a read-compute-update cycle. This process begins with a worker reading the current global solution vector from a global server and saving it in a local cache as $\tilde{x}$. Then, all workers begin the asynchronous coordinate descent process. The asynchronous coordinate descent process revolves around the following update rule for updating block i:

$$x_i^{j+1} = x_i^j - \eta \nabla_i f(\hat{x}^j) \tag{1}$$

For all other non-updating blocks $e : e \neq i$, the update rule becomes $x_e^{j+1} = x_e^j$. The solution vector in this update formula is represented as $x \in \mathbb{R}^N$. The variable j represents the jth global iterate. Thus, x^j is the jth global point during the ACD algorithm. The step-size, or learning rate, is denoted as η. The difference (or delay) between $\hat{x}$ and the global iterate x^j is defined as:

$$\hat{x}^j = \left(x_1^{j-d(j,1)},\ x_2^{j-d(j,2)},\ \ldots,\ x_N^{j-d(j,N)} \right) \tag{2}$$

The term $d(j, n)$ is defined as the number of iterations elapsed since the n^{th} coordinate was first updated during the course of Algorithm 2 (detailed in Appendix A). Thus, $d(j, n)$ denotes the delay at the n^{th} coordinate during the j^{th} iteration. The maximum delay is denoted as:

$$d(j) = \max_{1 \leq n \leq N} \{d(j, n)\} \leq \tau \tag{3}$$

As shown above, the maximum delay is bounded above by the constant τ. The delay bound is always greater than or equal to 1, $\tau \geq 1$. Delays arise from blocks that are more expensive to update than others (larger blocks, poor data locality, more non-zero entries, etc.) as well as slower workers. Bounded delays apply when a user is familiar with a hardware platform and can provide the delay bound from experience (either prior usage or by running a pilot test before).

With the presence of delays, the update rule presented in Equation 1 above cannot preserve its monotonically decreasing property. A Hamiltonian (sum of potential energy and kinetic energy) is developed to fight this issue. Following a similar process in [27], I modeled an energy function (Hamiltonian) to have a non-positive first derivative value in continuous time. This equation is defined as follows:

$$E(t) = f(x(t)) + \gamma \int_{t-c}^{t} (s - (t-c))||\dot{x}(s)||_2^2 \, ds \tag{4}$$

The term γ is the weighting for the kinetic energy term. I proved that for a step-size $\eta < \frac{1}{2L\sqrt{\tau}}$ and $\gamma = \frac{L}{2\sqrt{\tau}}$, that Equation 4 is in fact monotonically decreasing. Using these findings, I was able to model a discrete energy function based on the following Lyapunov function:

$$E_j := f(x^j) + \frac{L}{2\sqrt{\tau}} \sum_{i=j-\tau}^{j-1} (i - (j-\tau) + 1)||x^{i+1} - x^i||_2^2 \tag{5}$$

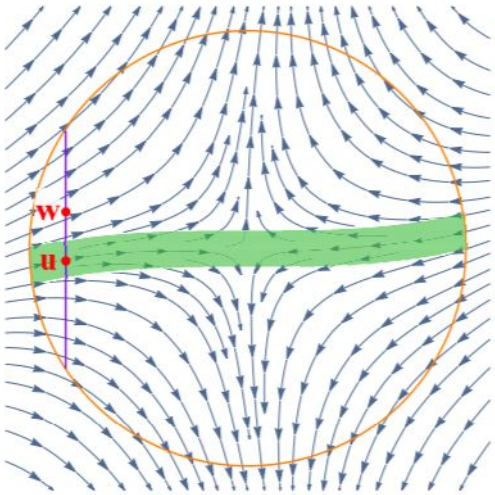
2.2 Escaping Saddle Points

Converging to second-order stationary points is a necessary condition to reach local or global optima within non-convex settings. While asynchronous coordinate descent descends towards first-order stationary points, it does not guarantee convergence to second-order stationary points. My research last summer revolved around theoretically proving that an iterate of asynchronous coordinate descent stuck at a saddle point can be perturbed in a manner which dislodges it from the saddle point (thus escaping it). The proof builds on a key characterization of the geometry around saddle points exploited in the Jin et al. papers [14, 16].

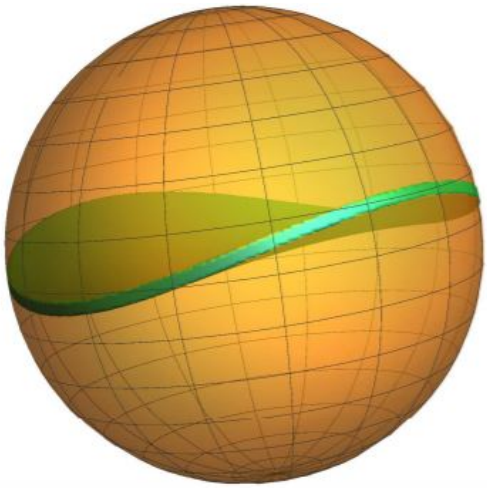
One of the main results from [14, 16] is that adding a carefully selected perturbation to a global iterate x^j stuck at a saddle point will have a high probability of escaping. The probability of failing to escape all saddle points during the course of the algorithm δ is a parameter specified by the user, so the probability of escaping all potential saddle points is defined as $1 - \delta$. The step-size η is proportional to δ, thus selecting an extremely small value of δ, to ensure a high probability of success, will greatly slow, or stop, convergence. In practice I set $\delta = 0.01$. The basis of the results in [14, 16] starts from selecting a perturbation ξ uniformly sampled from a d-dimensional ball and adding it to the global iterate $x^j \in \mathbb{R}^N$. The resulting perturbed point $y^0 = x^j + \xi$ can be viewed as coming from a uniform distribution over $\mathbb{B}_{x^j}^{(N)}(\eta r)$, which we call the perturbation ball. This perturbation ball is centered at the saddle point x^j with a carefully specified radius ηr (r is a hyperparameter). It is shown that the region of this perturbation ball, where iterates of gradient descent get *and remain* stuck after a specified number of iterations T (another hyperparameter), is *extremely small*. Furthermore, this region is only a very small proportion of the total volume of the perturbation ball. Below are diagrams from [14, 16] that display this phenomena.

Within my work, as well as [14, 16], the fraction of the total perturbation ball volume that the stuck region consumes is proven to be no larger than δ. Therefore, there is only a δ probability that the perturbed point $y^0 = x^j + \xi$ ends up in the stuck region. To determine whether the perturbed point escapes, it undergoes T iterations of asynchronous coordinate descent $\{y^t\}_{t=0}^{T}$. The final iterate, y^T, is tested to see if it escapes the saddle point (does it makes sufficient progress decreasing the objective function). If it does, then it becomes the next global iteration $x^{j+1} = y^T$. If not, then x^j is outputted as a second-order stationary point.

Figure 1a depicts a perturbation ball centered at a saddle point in two dimensions. The green region within Figure 1a portrays the *stuck region*: the region where iterates of gradient descent remain stuck even after T



(a) Narrow Stuck Region Within 2-Dimensional Perturbation Ball



(b) 3-Dimensional Perturbation Ball and Thin Stuck Region

Figure 1: Perturbation Ball Diagrams (Reproduced From [14])

iterations. As one can see, this region is quite small compared to the rest of the perturbation ball. Therefore, a perturbed point coming uniformly from this perturbation ball would very likely escape the saddle point after T iterations. The small size of the stuck region is also depicted in Figure 1b. Once again, the stuck region is extremely small relative to the total volume of the perturbation ball.

It is stated within [14] that "although we do not know the explicit form of the stuck region, we know it must be very 'thin', therefore it cannot have a large volume". Taking a cue from the work of Jin et al. I am able to bound the thickness ηr_0 of this stuck region. I prove that if a point is stuck anywhere within the stuck region, then another point at least $\eta r_0 = \frac{\eta r \delta \sqrt{\pi}}{2\sqrt{N}}$ away in the direction of the minimum eigenvalue (escaping direction) will not fall in the stuck region. This is depicted in Figure 1a with the point u stuck in the stuck region while w is outside the stuck region ($w - u = \eta r_0 e_1$, where e_1 is the direction of the minimum eigenvalue). Using this proof, I am able to determine an upper bound on the volume of the stuck region (with the help of the thickness ηr_0). Using this upper bounded volume, I can now show that the ratio of the stuck region volume to the total perturbation ball volume is extremely small, less than or equal to δ. Therefore, with a high probability of at least 1-δ, a uniformly sampled perturbed point from the perturbation ball will not fall within the stuck region. This is stated in the following Theorem (I moved the proof into the appendix to save space).

Theorem 1 (**Saddle Point Scenario**). *Let f be a L-smooth and ρ-Hessian Lipschitz function with a bounded minimum value f^*, $\delta \in (0,1)$ be the failure of escape probability, and $\epsilon \leq \frac{L^2}{\rho}$. By Definition 3, if $\|\nabla f(x^j)\|_2 \leq \epsilon$ and $\lambda_{\min}(\nabla^2 f(x^j)) \leq -\sqrt{\rho\epsilon}$, then $x^j \in \mathbb{R}^N$ is located at a saddle point. Let $\{y^t\}_{t=0}^T$ be the iterates of asynchronous coordinate descent starting from the perturbed point $y^0 = x^j + \xi$ (where ξ is uniformly sampled from a ball with radius ηr) with $T = \frac{\log_2(\frac{16\sqrt{N}\epsilon}{r\delta\sqrt{\pi}})}{\eta\sqrt{\rho\epsilon}}$. Then, with at least probability $1-\delta$, the subsequent global iterate $x^{j+1} = y^T$ escapes the saddle point.*

2.3 Main Algorithm (SEACD)

Below I reveal the Saddle Escaping Asynchronous Coordinate Descent (SEACD) algorithm. To save space, the three inner algorithms, Single Worker Asynchronous Coordinate Descent (SWACD), Global Asynchronous Coordinate Descent (GACD), and Perturbed Asynchronous Coordinate Descent (PACD), have

been moved to the appendix of this report. The SEACD algorithm is shown below.

Algorithm 1: $(x_\epsilon^*) = \text{SEACD}(x^0, f, \eta, r, \tau, T, \mathscr{F}, M, L)$

Input: An initial point $x^0 \in \mathbb{R}^N$, objective function f, learning rate (step-size) η, perturbation radius r, delay bound τ, escaping time bound T, function change threshold $\mathscr{F}$, momentum threshold M, gradient-Lipschitz L

Output: Returns an ϵ-second-order stationary point x_ϵ^*

```
1  E_0 ← f(x^0);
2  j ← 0;
3  for s = 1, 2, 3, ... do
4  |   n, x^{j+n}, E_{j+n} ← GACD(x^j, f, η, τ, M, L);
5  |   j ← j + n;
6  |   if (E_j − E_{j−n}) > −ℱ then
7  |   |   x^{j+1}, E_{j+1} ← PACD(x^j, f, η, τ, r, T, L);
8  |   |   j ← j + 1;
9  |   |   if (E_j − E_{j−1}) > −ℱ then
10 |   |   |   break;
11 |   |   end
12 |   end
13 end
14 return x^j
```

The SEACD algorithm shown above is broken down into the following steps. In line 4, the global asynchronous coordinate descent (GACD) sub-algorithm is called to asynchronous coordinate descent on point x^j. During this sub-algorithm, the point undergoing coordinate descent will either decrease the objective function by a specified function threshold $\mathscr{F}$ (a hyperparameter), and thus make improved progress, or will not and be flagged as a potential second-order stationary point. This flagging of the function threshold occurs in line 6. If the point is flagged as a potential second-order stationary point, then the perturbed asynchronous coordinate descent (PACD) sub-algorithm is performed. This sub-algorithm perturbs that point and then performs asynchronous coordinate descent on the perturbed point. After this sub-algorithm finishes, there is a check to determine whether the perturbed point is dislodged from the saddle. Dislodging the point would mean that the perturbed point decreases the objective function by the specified function threshold $\mathscr{F}$. If the point is dislodged, then the function is decreased by $\mathscr{F}$ and the algorithm will return to line 4. If not, then a second-order stationary point is found and the algorithm breaks at line 10 before returning the current point. All the sub-algorithms within SEACD are displayed in Appendix A.

3 AMSC 664 Final Results

All of my results are found in my GitHub repository. My repository includes all Python files and figures included in this report. The link to my repository is: `https://github.com/Marcob1996/AMSC663_664`.

At the end of last semester and the beginning of this semester, I had implemented a serial version of SEACD that lacked full asynchronicity. I compared SEACD to gradient descent and perturbed gradient descent on a high-dimensional t-SNE test case. I aimed to visualize the MNIST database of handwritten digits via t-SNE to successfully cluster MNIST numbers into separate groups in two dimensions. My expectation was that after reducing the data dimensions from 784 to 2, one can still differentiate between the hand-written numbers

in the embedded two-dimensional space. I used SEACD, GD, and PGD to power the t-SNE algorithm and the results are shown in the figure below.

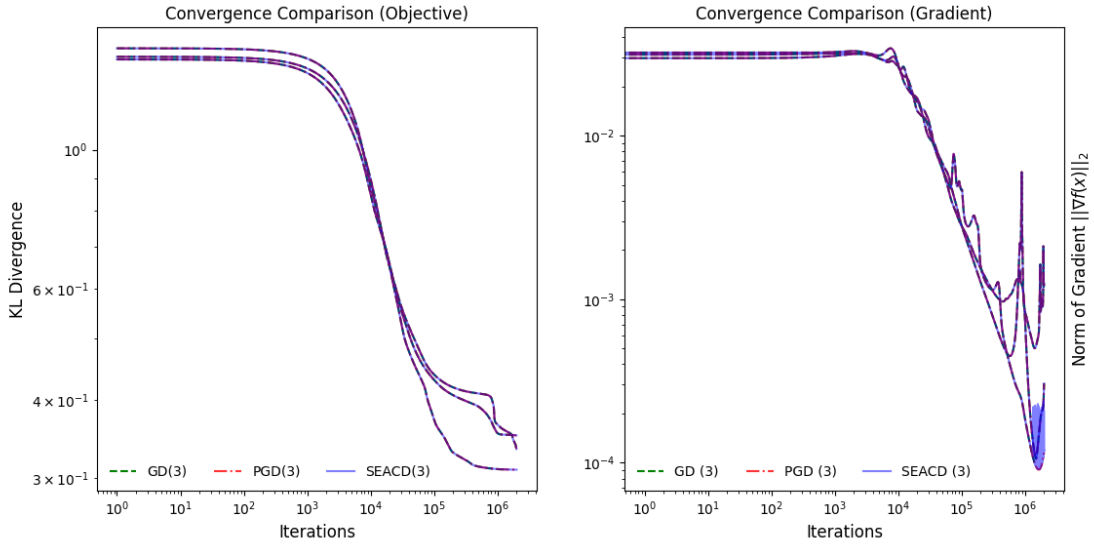


Figure 2: t-SNE Convergence Comparison

In both sub-figures of Figure 2, one can see that all three algorithms successfully minimized the KL divergence in the t-SNE algorithm. In the final run, SEACD and PGD did trigger perturbations near the end of the two million iteration limit. The perturbations were triggered as the local minimum was discovered. Even so, SEACD was able to end on a slightly better minimum KL divergence value than PGD and GD.

What is important to notice is that in the left sub-figure of Figure 2, GD, PGD, and SEACD all followed the same paths except for the final run. There are two reasons for this result. The first is that the objective function did not encounter any first-order stationary points during the course of all my runs. In fact, every run was heading towards a similar local minimum value of 0.309 (this too is the minimum value determined by the open source t-SNE package created by [29]). This is due to many flat regions of small gradient values, yet not too small to trigger any perturbations for SEACD or PGD. Because no perturbations were triggered, except for the last run, the non-asynchronous serial SEACD is not able to differentiate from PGD or GD.

The t-SNE example was a success, in that it showed SEACD does converge to local optima in a manner similar to other methods like PGD, which converge to second-order stationary points. Furthermore, the t-SNE example showcases the necessity to implement a serial-asynchronous, and parallel-asynchronous, SEACD algorithm to differentiate from PGD and GD (in the case that saddle points are unlikely to be experienced). This is where I began my progress for AMSC 664, implementing and validating a serial-asynchronous version of SEACD.

3.1 Implementation and Validation of Serial-Asynchronous SEACD

As mentioned above, my first goal within AMSC 664 was to implement SEACD in a serial-asynchronous manner, and validate that it converges to second-order stationary points. Below, I first detail how I implemented this serial-asynchronous process. Then, I dive into the test case I used to validate my implementation of serial-asynchronous SEACD.

3.1.1 Serial-Asynchronous SEACD Implementation

Previously, my implementation of SEACD consisted of alternating between coordinate blocks of the solution vector and updating them with no delay. This implementation can be described as one main worker consecutively updating all the coordinate blocks. Therefore, this process was not asynchronous at all. Even further, the use of a hyperparameter τ is useless if blocks are updated consecutively (there is no real delay, only waiting for ones turn to update). To fix this, and create a serial-asynchronous SEACD algorithm, I altered both the Global Asynchronous Coordinate Descent (GACD) and Perturbed Asynchronous Coordinate Descent (PACD) algorithms (defined in the Appendix A as Algorithms 4 and 5) with the same following steps.

First, I split the solution vector x^j into equal chunks amongst the set number of asynchronous workers. Afterwards, I designate one of these workers as the "slow" worker, which means that this worker takes τ iterations to update its coordinate block. During both GACD and PACD, for the first $\tau - 1$ iterations I continually update all other workers in a random order (I ensure each worker is updated once and then uniformly at random select workers after). Then, on the τth iteration, I update the designated "slow" worker. I repeat this process until both GACD and PACD terminate.

I implemented this serial-asynchronous SEACD algorithm within Python using NumPy. After implementation I pivoted towards validating my new SEACD algorithm. To validate, I used the serial-asynchronous SEACD algorithm to minimize the Matrix Factorization problem.

3.1.2 Serial-Asynchronous SEACD Validation

As mentioned above, I validated the serial-asynchronous SEACD algorithm by applying it to the Matrix Factorization problem. In the Matrix Factorization problem, a matrix $S \in \mathbb{R}^{mxn}$ is factored into two other matrices $U \in \mathbb{R}^{mxk}$ and $F \in \mathbb{R}^{kxn}$. The parameters m, k, and n can all be equal to one another, but in many cases k is much smaller than m and n (in the case of low-rank matrix factorization). This is a non-convex optimization problem, with many local minima, and the objective function I am minimizing is the mean squared error (MSE) between S and UF. This is defined below as:

$$\underset{U,F}{\text{argmin}} \quad \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(S_{ij} - (UF)_{ij})^2 \tag{6}$$

Computing the gradient for each value of U and F is more complex. The reason for this, is that each element of U and F plays a role in the formation of *multiple* elements in the solution matrix S. For example, if $S, U, F \in \mathbb{R}^{2x2}$, then $U_{1,1}$ directly affects what the values $S_{1,1}$ and $S_{1,2}$ are due to matrix multiplication. This is shown in the following MSE calculations:

$$MSE(S_{1,1}) = (S_{1,1} - [(U_{1,1} * F_{1,1}) + (U_{1,2} * F_{2,1})])^2$$
$$MSE(S_{1,2}) = (S_{1,2} - [(U_{1,1} * F_{1,2}) + (U_{1,2} * F_{2,2})])^2$$

To compute the gradient of a single element of U or F, like $U_{1,1}$, the gradient of each solution matrix element it affects must be averaged together. For elements within the matrix U, this is defined as:

$$\frac{\partial MSE}{\partial U_{i,j}} = \frac{1}{n}\sum_{h=1}^{n}\frac{\partial MSE(S_{i,h})}{\partial U_{i,j}} \tag{7}$$

Similarly, for elements within the matrix F, the gradient is defined as:

$$\frac{\partial MSE}{\partial F_{i,j}} = \frac{1}{m}\sum_{h=1}^{m}\frac{\partial MSE(S_{h,j})}{\partial F_{i,j}} \tag{8}$$

To begin this optimization problem, I initialize random matrices for U and F. I then reshape these two matrices into an initial solution vector x^0. This vector is fed into the serial-asynchronous SEACD algorithm along with the objective function (MSE) and a function that will compute the gradient for each element of the solution vector (as this vector contains each element of U and F). From there, I let my new implementation of SEACD go to work. When testing, I used a solution matrix $S \in \mathbb{R}^{10x10}$ and two random matrices $U \in \mathbb{R}^{10x2}$ and $F \in \mathbb{R}^{2x10}$. This low-rank matrix factorization ensured that my algorithm would experience many local-minima and potential saddle points. I compared my new implementation of SEACD to gradient descent, and below are the convergence results of an example run (to save space I have left the figures from the other runs on my GitHub repository). I selected three asynchronous workers and a maximum bounded delay of $\tau = 3$ for SEACD. Figure 3 depicts how serial-asynchronous SEACD and gradient descent minimize the MSE objective function.

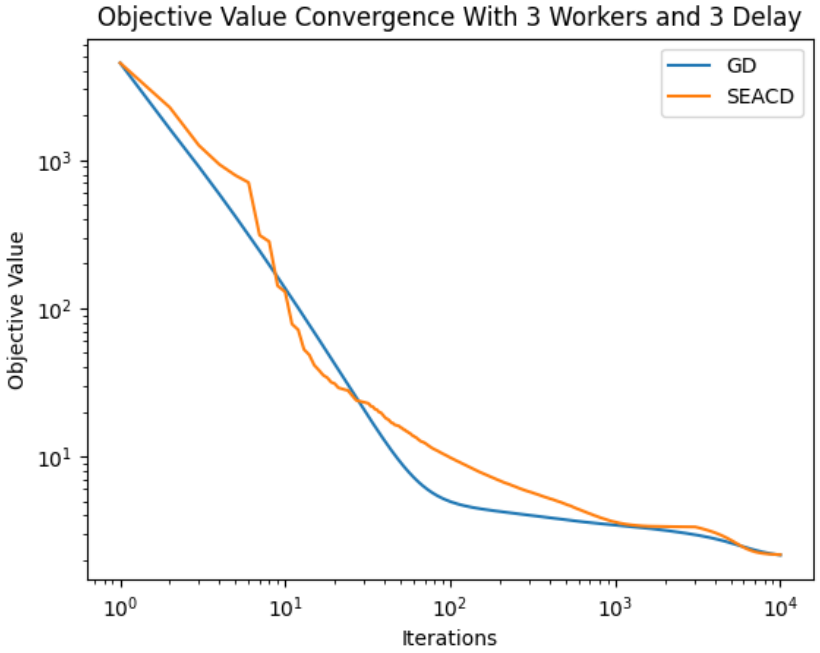


Figure 3: Objective Value Convergence $\tau = 3$ (One Run)

As one can see in Figure 3, both SEACD and gradient descent look to converge to a local minimum with an objective value ≈ 1.5. Compared to Figure 2, where SEACD and GD followed a similar path of convergence towards a local minima, SEACD and GD follow different paths. Due to the asynchronous nature of SEACD, some iterations will produce a decrease of the objective function that is greater or less than that for gradient descent (as some workers will experience a significant decrease of the objective function while others may not). As one can see, SEACD jaggedly decreases the objective function faster than GD at the beginning iterations before it slows and then catches back up to GD at the final iteration.

Along with Figure 3, Figure 4 displays the difference in convergence between SEACD and GD. Once again, SEACD produces a more jagged convergence result due to its asynchronous nature (each coordinate block can have a stark difference in its gradient values). What we see within Figure 4 is that SEACD reaches a potential local minima, which triggers a perturbation. This is why there is a large spike in the graph for SEACD. Once the norm of the gradient decreases after the perturbation, SEACD terminates as a local minimum has been reached (the perturbation did not escape the potential saddle point, thus it must be a second-order stationary point).

Overall, one can also see in Figures 3 and 4 that both algorithms converge to a similar local minima in a similar amount of iterations. Over the course of multiple test runs, I have found that the convergence rates of both algorithms are quite similar. This is important, as gradient descent is an extremely efficient algorithm. Aligning with gradient descent shows the convergence strength of serial-asynchronous SEACD.

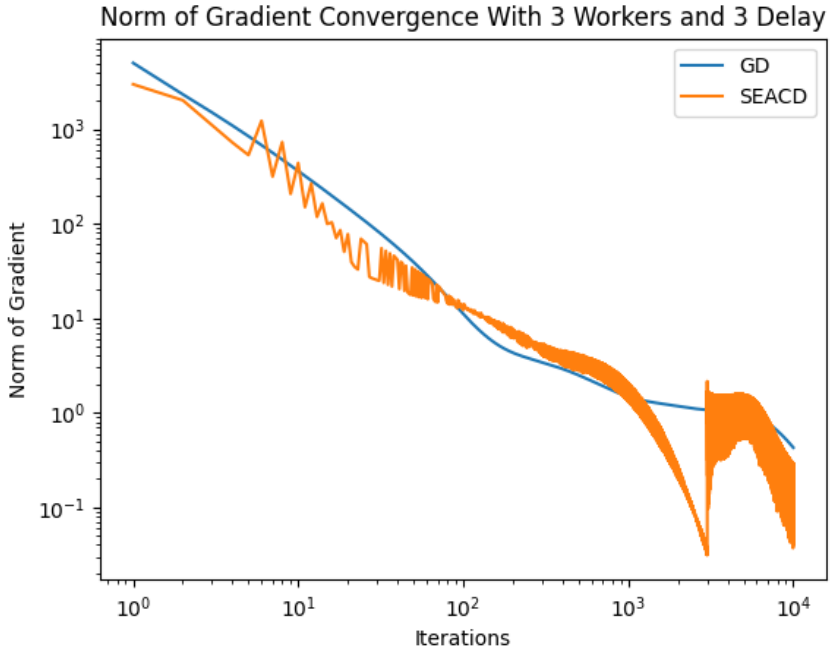


Figure 4: Norm of Gradient Convergence $\tau = 3$ (One Rune)

3.2 Classifying MNIST Images Using Convolutional Neural Networks

One of the goals during the middle of the semester that I sought to accomplish was to implement my serial-asynchronous SEACD algorithm to train a convolutional neural network. Dr. Cameron provided a MATLAB live script that built a simple convolutional neural network to classify MNIST images with high accuracy (94.2%). This live script can be accessed at this link: `https://www.mathworks.com/help/deeplearning/ref/trainnetwork.html`. The Modified National Institute of Standards and Technology (MNIST) database is a large database filled with thousands of 28x28 pixel images of handwritten numbers (0 through 9). MNIST is commonly used for training and testing algorithms within machine learning. Example MNIST images are shown in Figure 5.

The convolutional neural network created within the live script consists of the following layers: a two-dimensional convolutional layer with 20 filters of size 5x5 (no stride), a ReLu layer, a max pooling layer for size 2x2 with a stride of length 2 in both directions, a densely connected layer, and a softmax layer at the very end before classification.

During this semester, I constructed a similar convolutional neural network from scratch using Python and TensorFlow. Only at this point did I run into issues. I was unable to implement my SEACD algorithm as an optimizer within TensorFlow. The reasons for this are that my algorithm is complex compared to some of the simpler built-in optimizers, and there is not a lot of freedom and customization offered within TensorFlow when creating a novel optimizer. SEACD requires the function to be minimized f as an input, and using the classification layer as f is not possible to implement within the TensorFlow custom optimizer

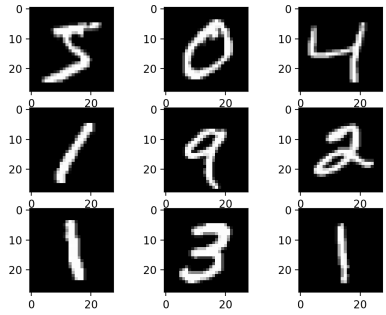


Figure 5: Example MNIST Images

framework (let alone that this choice of f causes many issues with computing the gradient as well as figuring out how to create one's own back-propagation process within TensorFlow). Due to all of these issues keeping me from implementing SEACD in TensorFlow, I was unable to train my convolutional neural network using it.

While I was unable to train my convolutional neural network using SEACD to classify MNIST images, I was able to train it using the ADAM. ADAM is an extremely powerful and commonly used optimization algorithm often used to train neural networks. Because I spent so much time constructing my convolutional neural network, I wanted to show that it does achieve similar accuracies to those found in the MATLAB live script. The results of my MNIST image classification are shown in Figure 6.

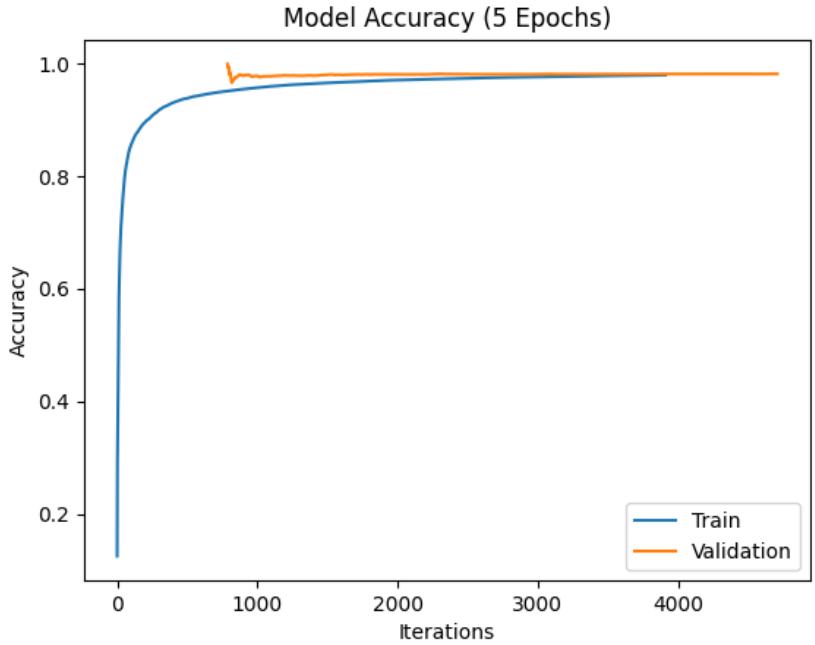


Figure 6: MNIST Classification Accuracy From My Constructed CNN

As one can see in Figure 6, my convolutional neural network trained using ADAM was able to achieve a slightly higher classification accuracy (>95%) after 5 epochs of training. On most test runs, I reached 98-99%

classification accuracy. A reason for the higher accuracy likely arises from the use of ADAM and a slightly larger step-size (no dropout is used in either case). While this entire process did not result in exactly what I planned, the achievements I made were invaluable. I learned how to implement neural networks in Python, and was able to successfully solve a large-scale classification problem.

3.3 Implementation and Validation of Parallel SEACD

The final, and biggest, milestone I accomplished in AMSC 664 was to implement a parallel-asynchronous version of SEACD within Python. Specifically, I created a Python Module from which a user can call my parallel-asynchronous SEACD algorithm. Below, I will discuss the issues I faced while implementing SEACD in parallel as well as provide an overview of the Module I created.

3.3.1 Parallel Implementation Issues

The parallel processing capabilities of Python have improved over the previous years. However, they still fall short of the capabilities found in other languages such as C or C++. While MPI and OpenMP (if using threads) are efficient and flexible libraries within C an C++, there lacks a perfect substitute in Python. Parallelization can be implemented within Python using the Multiprocessing package. Unlike MPI, Multiprocessing is limited and still developing. As mentioned in Section 2.1, asynchronous coordinate descent requires parallel workers, in this case processors, to communicate their updates to a global server. This communication is difficult and inefficient to implement within the Multiprocessing package (and thus Python in general). If I were more practiced with to C or C++, I would have tried to implement this parallel process using MPI where communication is quick and efficient between processors. Within the Multiprocessing package, these are the limited avenues I traversed to get SEACD up and running efficiently: Multiprocessing managers, queues, and shared memory.

I initially utilized Multiprocessing managers to allow communication between processes (workers). Multiprocessing managers control a server process and allows other processes to manipulate them using proxies. While this enabled communication between processes, it was extremely slow and actually caused a slow-down compared to my serial implementation of gradient descent and SEACD.

After using Multiprocessing managers proved infeasible due to their speed, I incorporated queues. Queues are a first in first out (FIFO) data structure that are revamped in the Multiprocessing package to allow processes to communicate safely with one another. Queues are the recommended method for communication between processes if needed. When using a queue, I instructed each process to place its updates into the queue and then immediately get the next value from the queue to work on. The issue with implementing a queue for asynchronous coordinate descent, is that the queue was rarely filled. The queue would usually have 0 or 1 global iterates available at a time (as the workers are quick with their work). Thus, blocking between workers was very common with this implementation. Many workers were waiting until there was an iterate placed in the queue. This caused unnecessary waiting and wasted computational power. While implementing a queue did cause a speed-up in runtime, it was minimal due to blocking.

Finally, I found the best solution to my communication problem: shared memory. The shared memory Python module is new to the Multiprocessing package (introduced in Python 3.8). Through this module, I am able to allocate and manage shared memory that can be accessed by processes. Utilizing this, I was able to create the current global iterate of global asynchronous descent as shared memory. Then, I allowed each process to read in and save their updates to the shared memory as soon as they were ready. This eliminated any waiting time, and the runtime greatly increased. As one can see in Table 1, the shared memory implementation led to speed-up of 2-2.5x in some examples.

3.3.2 Parallel-Asynchronous SEACD Python Module Overview

The parallel-asynchronous SEACD module that I implemented within Python contains the following functions:

Initialization: This function initializes all of the hyperparameters, shared memory for the initial solution, and lists which keep track of the norm of the gradient values.

Update: This function updates the Hamiltonian given the computed work of a single asynchronous worker. The computed work is the step that the global iterate will take next, which is the gradient multiplied by the step-size. Within this function, other hyperparameters necessary to compute the Hamiltonian are also updated.

Block Allocation: This simple function splits the blocks evenly amongst the inputted number of asynchronous workers. If there is an uneven split, one worker is given a smaller coordinate block than the others.

Perturb: The perturbation function simply adds a perturbation from a N-dimensional ball to a point stuck at a potential saddle point. This function is only called during the PACD algorithm.

Single Worker Asynchronous Coordinate Descent (SWACD): Within the SEACD algorithm, this function is the powerhouse. The SWACD function is run in parallel by GACD and PACD. A full overview of the algorithm is detailed in Algorithm 3 within Appendix A. Within this function, an asynchronous worker reads in the shared memory solution, computes its gradient computations for its coordinate block, records the norm of the gradient for the block update, and then updates the shared memory solution.

Global Asynchronous Coordinate Descent (GACD): This function runs asynchronous coordinate descent on the global iterate until a first-order stationary point is reached (a point with a small gradient). The detailed overview of GACD is described in Algorithm 4. It is within this function that I initialize parallel processes using the Multiprocessing package to run SWACD until a first-order stationary point is reached.

Perturbed Asynchronous Coordinate Descent (PACD): As detailed in Algorithm 5, this function perturbs an iterate that is potentially stuck at a saddle point. When called, an iterate is perturbed using the Perturb function and then parallel processes are initialized to run SWACD for a specified number of iterations T (a hyperparameter). If the iterate after T asynchronous coordinate descent update does not sufficiently decrease the Hamiltonian, then PACD breaks and the pre-perturbation iterate is returned as it is a second-order stationary point. If the Hamiltonian does decrease sufficiently, then a saddle point has been escaped and the SEACD algorithm continues.

Saddle Escaping Asynchronous Coordinate Descent (SEACD): Using all of the functions I define above, this function runs the parallel-asynchronous SEACD algorithm. All a user needs to do is call this function after initializing the hyperparameters to run the SEACD algorithm.

3.3.3 Parallel-Asynchronous SEACD Validation

To validate my parallel-asynchronous SEACD algorithm, I again applied it to a Matrix Factorization problem. However, this time I solved the Non-Negative Matrix Factorization (NMF) problem. In the Non-Negative Matrix Factorization problem, a non-negative matrix $A \in \mathbb{R}_+^{mxn}$ is sought to be factored into two non-negative, and often lower-ranked, matrices $W \in \mathbb{R}_+^{mxk}$ and $H \in \mathbb{R}_+^{kxn}$ [6]. The NMF problem is non-convex,

with potentially many local minima. To reach a local minima, I utilized a Projected GD algorithm. For the case of a simple non-negativity constraint, one can set the projection as $P(x) = [x]_+$. This projection is simply the maximum value between x and 0. The Projected GD update rule becomes:

$$x^{j+1} = P\big(x^j - \eta \nabla f(x^j)\big) \tag{9}$$

Thus, when applying gradient descent, if a coordinate becomes negative it is set to zero instead. For this example, I adjust my SEACD algorithm to become a projected version. This only took a couple tweaks, namely ensuring that no coordinate can be negative during each global iterate update and setting all negative values to zero. The gradient of the matrices W and H for Projected GD, derived in [6], are equal to the following:

$$\begin{aligned} \nabla W &= (WH - A)H^T \\ \nabla H &= W^T(WH - A) \end{aligned} \tag{10}$$

Once again, similar to my Matrix Factorization test problem in Section 3.1.2, I use MSE as the objective function to minimize.

$$\underset{W,H \in \mathbb{R}_+}{\operatorname{argmin}} \quad \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (A_{ij} - (WH)_{ij})^2 \tag{11}$$

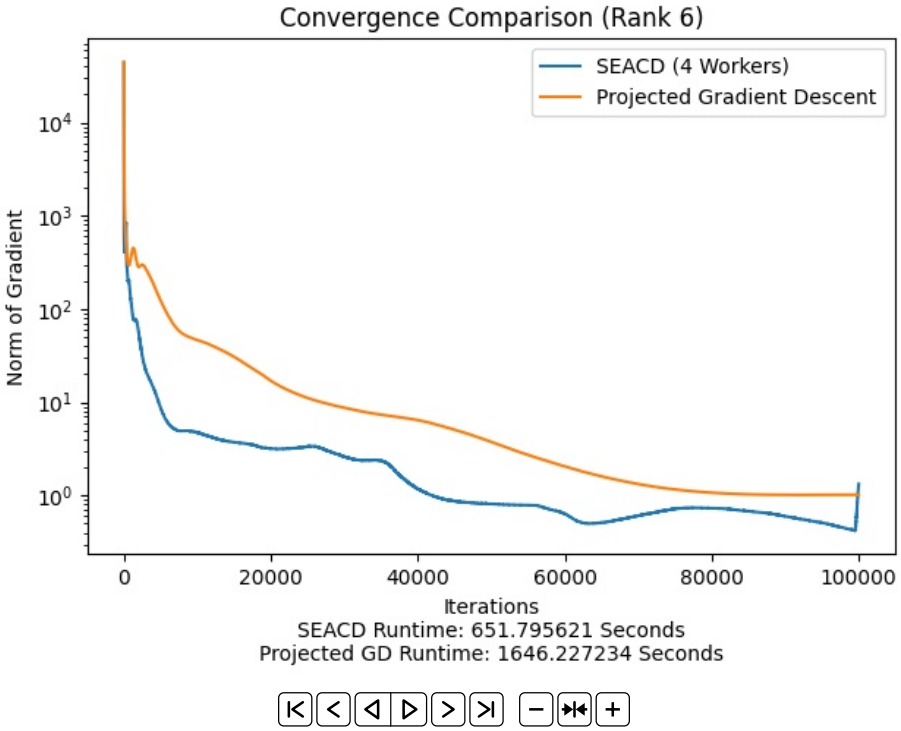
Like in the Matrix Factorization problem, I initialize random matrices for W and H. I reshape these two matrices into an initial solution vector x^0. This vector is fed into the parallel-asynchronous SEACD algorithm along with the objective function (MSE) and a function which computes the gradient of W and H. From there, the projected parallel-asynchronous SEACD algorithm does the rest.

The matrix A I chose to factorize is the following black and white image of Pike Place Market in my hometown of Seattle, Washington. I am hoping to factor this image into matrices W and H that contain the important features of the original Pike Place Market image. Further, I hope that multiplying W and H will result in an image that is nearly identical to the original image.

Figure 7: Original Pike Place Market Image To Be Factorized

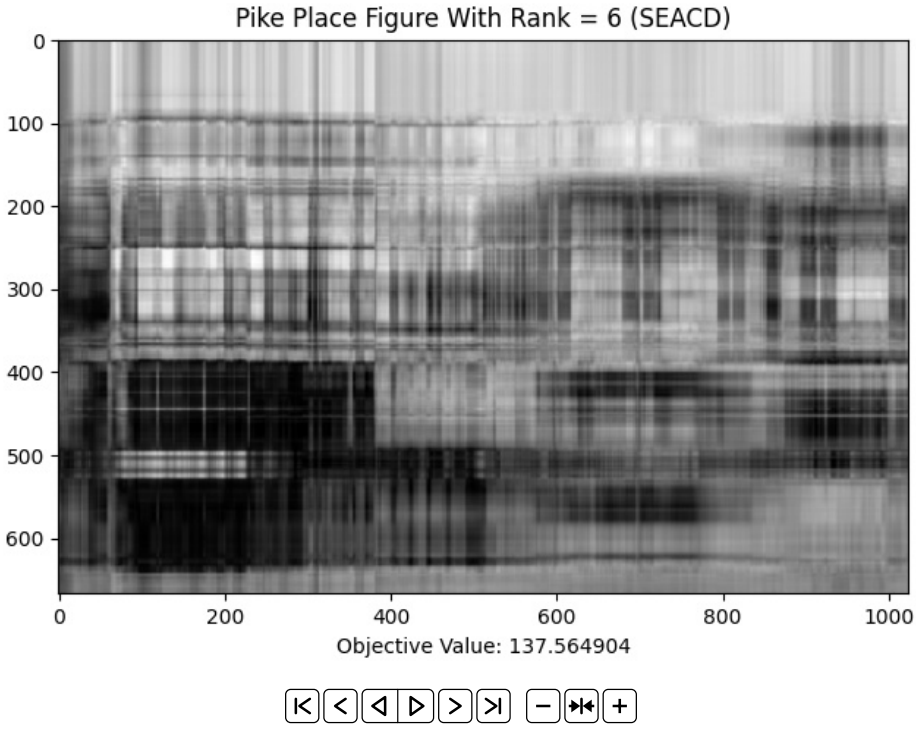
The Pike Place Market image is large, with dimensions of 667x1024 pixels. Using Projected GD and Projected SEACD to solve the Non-Negative Matrix Factorization problem, I will factorize the Pike Place Market image

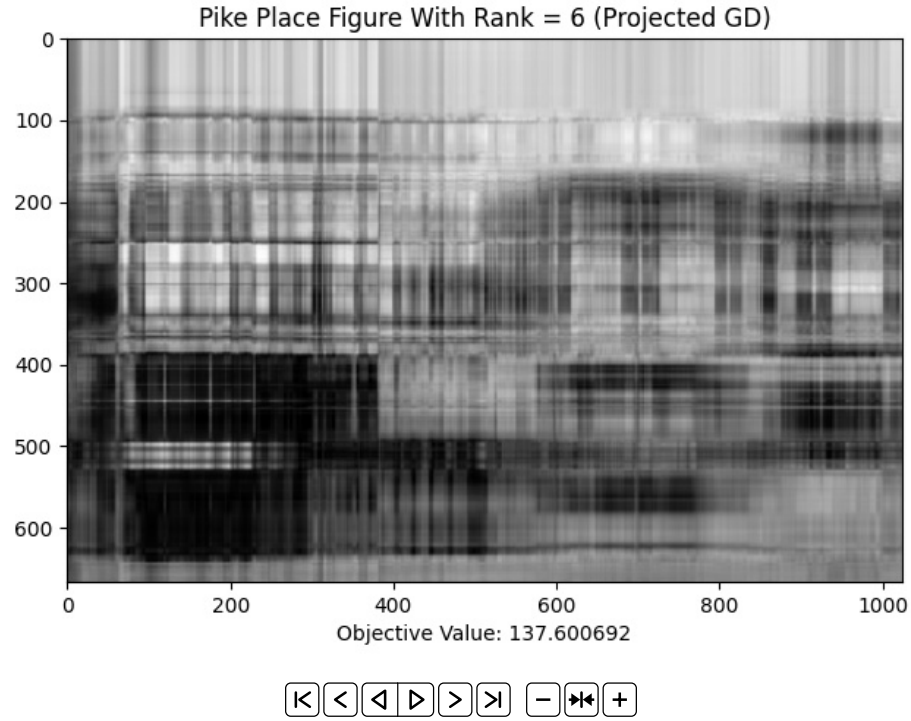
into matrices W and H which are lower rank ($m, n >> k$). I solved NMF for the following lower ranks: k = 6, 11, 21, 42, 84, 112, 167, and 334. Solving for all of these lower ranks is important to show that my parallel-asynchronous SEACD algorithm not only consistently converges to second-order stationary points, but does so with a speed-up in runtime compared to serial GD. In the animation below, the convergence comparison between Projected SEACD and Projected GD is shown for each low rank k I tested.



As one can see in each figure of the animation above, Projected SEACD decreases the norm of the gradient at a quicker rate than Projected GD. It is important to note that computing the norm of the gradient is a little tricky in NMF, as the coordinates set equal to zero due to the projection may have a large gradient values. This arises as the coordinates set to zero may have a strong pull to become negative (gradient is pulling them in this direction), but nothing comes of this as the projection keeps them at a value of zero. This large underlying gradient is the likely culprit that the norm of the gradient is not smaller for both Projected SEACD and Projected GD after 100,000 iterations. The spike at the end of some of the lower rank tests for Projected SEACD is an interesting occurrence. While it is likely caused by an added perturbation (as Projected SEACD has reach a first-order stationary point), another possible reason arises from the final process (asynchronous worker) finishing up the last few iterations by itself. This final process may be computing the norm of the gradient for a coordinate block containing a large number of zero-valued coordinates that may have a large gradient when in reality they will stay fixed at a zero value due to the projection. As one can see from all these images, however, is that Projected SEACD converges at a similar or slightly faster rate than Projected GD to a local minima.

This is backed up by the following two animations below. These two animations display the low-rank approximation to the Pike Place Market image, produced by multiplying the two factorized matrices WH. Both Projected SEACD and Projected GD are able to recapture most of the important aspect of the Pike Place Market image at extremely low ranks ($k = 6, 11, 21$) and produce an accurate low-rank approximation of the Pike Place Market image at higher values of k ($k = 112, 167, 334$).





These low-rank image animations confirm what is shown in the convergence comparison animation, that Projected SEACD and Projected GD have reached (or are close to reaching) a local minima. This too is seen in the difference in objective function (MSE) values for Projected SEACD and Projected GD in the animations above. The MSE for Projected SEACD is slightly that of Projected GD. Projected SEACD makes slightly more progress than Projected GD. The reason for this is two fold: the added perturbation

and coordinate descent effectiveness. The Non-Negative Matrix Factorization may generally contain many local minima [6] and so perturbations can help lead to better local minima. Finally, as shown in Section 3.1.2, asynchronous coordinate descent follows a different path towards convergence than a standard gradient descent method. Thus, Projected SEACD was able to either make more progress towards the same local minima compared to Projected GD, reach a separate and better local optima than Projected GD, or a combination of the two over the course of all the low-rank tests. What the NMF validation test problem showed, is that my parallel-asynchronous SEACD implementation converged to second-order stationary points and did so in a similar manner as gradient descent, an extremely efficient algorithm.

3.3.4 Parallel Speed-Up

What I did not mention in Section 3.3.3, is that one can see in the convergence comparison animation that my parallel-asynchronous SEACD algorithm achieves a sizeable speed-up in runtime over serial gradient descent. The runtimes displayed in the convergence comparison animation are reproduced in Table 1 below.

Table 1: Projected GD vs. Projected Parallel SEACD (4 Workers), Max Iterations: 100,000

Rank k	Projected GD Runtime (s)	Projected Parallel SEACD Runtime (s)
6	1,646.22	651.79
11	1,783.31	784.62
21	1,973.61	1,014.41
42	2,177.85	1,259.92
84	2,834.21	1,958.39
112	3,379.59	2,179.02
167	4,107.27	2,916.62
334	5,580.76	4,948.33

As one can see, my parallel implementation did indeed provide a speed-up in the runtime compared to serial gradient descent. For each rank, I achieve a reduction is about 1,000 seconds. I ran all of these low-rank tests consecutively (this took over 7 hours), starting with the lowest ranks first. Because of this, the last couple of low-rank tests likely produced runtimes that were affected by my computer's performance. My computer was using so much processing power for so long that the final results are likely off by a little bit. Regardless, my parallel implementation cuts down the total time by a sizable chunk. After so many issues with implementing SEACD in parallel, it was amazing to see the effects of parallelization when solving the NMF problem.

3.4 Optimization of Hyperparameters

I began this semester strengthening the theoretical results of my SEACD algorithm by better selecting its hyperparameters. Even into the middle of the semester, I was adjusting the hyperparameters to increase convergence results and ease of use for the user. These improvements are behind the scenes and difficult to illustrate, but include advancements in the step-size η, dimension-dependence, and convergence-dependence on the maximum delay τ.

The first major improvement I made this semester was explicitly selecting the step-size η as a hyperparameter. Previously, a user had to search for a feasible value of η and input it into SEACD. This would be a confusing process for a user not accustomed to SEACD, let alone optimization algorithms in general. Now, η is a hyperparameter that depends upon the gradient-Lipschitz constant L and maximum delay bound τ. This improvement increases ease of use for users of SEACD.

The second improvement is that I was able to prove that the convergence of SEACD depends only poly-logartihmically with respect to the dimension. This is important because the algorithm matches up with the dependence on dimension found within [14, 16]. Having only a poly-logarithmic dependence on dimension allows SEACD to remain efficient and useful in large-scale non-convex optimization problems. This result is an improvement over other first-order methods which reach second-order stationary points with a higher dependence on dimension [17].

Finally, the last major improvement was to reduce the dependence of the maximum delay τ on convergence. Previously, the convergence was quadratically dependent on the maximum delay. This is an issue if workers are extremely slow, and decreases the practicality in using an asynchronous algorithm. The incentive for using asynchronous algorithms are for cases where workers are slow to submit their work (allowing other workers to keep working instead of waiting), so why would one want to use an asynchronous algorithm with a convergence that has a poor dependence upon the maximum delay? This semester I was able to improve the convergence rate by now only depending sub-linearly on the maximum delay bound. This helped speed up the convergence of SEACD.

4 Deliverables

Below is an itemized list of all my deliverables for this semester. My code and documentation are found in my GitHub repository: `https://github.com/Marcob1996/AMSC663_664` Included within my repository are:

- Serial-Asynchronous SEACD module
- Matrix Factorization test code
- Convolutional Neural Network and MNIST test code
- Parallel-Asynchronous SEACD module
- Projected GD and Projected Parallel-Asynchronous SEACD modules
- Non-Negative Matrix Factorization test code
- Figures for Matrix Factorization convergence, CNN accuracy, and NMF convergence

5 Conclusion and Future Work

Over the course of both AMSC 663 and 664, I dove deeply into the depths of asynchronous coordinate descent. Building an algorithm founded on asynchronous coordinate descent provides satisfactory results when minimizing a convex or non-convex function, as depicted in my results above. I have shown that algorithms built on asynchronous coordinate descent, like SEACD, achieve convergence similar to other efficient gradient descent methods. Even further, asynchronous coordinate descent scales up to large problems well (my NMF validation problem shows this) and can be a quick algorithm when implemented in a parallel-asynchronous fashion.

While asynchronous coordinate descent and algorithms like SEACD that build off it are effective, gradient descent methods with momentum reign supreme. These momentum methods jump through flat region areas without wasting iterations, and are blazing fast at finding local minima. When comparing SEACD to accelerated gradient descent, ADAM, or stochastic gradient descent with momentum, they converge in

a much quicker manner. This is evident in the usage of accelerated gradient descent in the t-SNE python and MATLAB packages. The creator of t-SNE, Laurens van der Maaten, used accelerated gradient descent due to its rapid convergence and efficiency in visualizing extremely high-dimensional data like MNIST. One of my main goals while continuing to develop SEACD, is to include acceleration in the algorithm. This is necessary to compete against the other super-powerful first-order methods available.

Although I was able to implement a parallel version of SEACD successfully, doing so in Python is sub-optimal compared to other languages such as C/C++. Python's "multiprocessing" package lacks functionality compared to packages like Open MPI on C/C++. The increased functionality and efficiency would greatly boost the parallel speed-up in runtime that is observed within my results. This would be one of the next major pieces of future work: implementing parallel-asynchronous SEACD in C/C++ to further increase the speed of the algorithm.

In summary, future areas that could be expanded upon and explored for SEACD include:

1. Implementing an accelerated version of SEACD
2. Implementing SEACD in parallel within C or C++ to boost parallel computing power and efficiency
3. Comparing the t-SNE results with parallel-asynchronous SEACD (or an accelerated version of SEACD) against GD, PGD, and accelerated gradient descent
    - To examine if the parallel-asynchronous SEACD algorithm (or accelerated version) is able to match the rapid and accurate results of the accelerated gradient descent algorithm that is usually used to solve the t-SNE problem
4. Testing the theoretical convergence result of SEACD numerically (to confirm that it matches)

Acknowledgements

Also, I wanted to thank both Dr. Balan and Dr. Cameron for their guidance, critiques, and advice on my project. You pushed me to build my computational and mathematical skills and accomplish so much this semester!

References

[1] Zeyuan Allen-Zhu and Yuanzhi Li. Neon2: Finding local minima via first-order oracles. *arXiv preprint arXiv:1711.06673*, 2017.

[2] Hédy Attouch, Jérôme Bolte, Patrick Redont, and Antoine Soubeyran. Proximal alternating minimization and projection methods for nonconvex problems: An approach based on the kurdyka-łojasiewicz inequality. *Mathematics of operations research*, 35(2):438–457, 2010.

[3] Vijay Badrinarayanan, Bamdev Mishra, and Roberto Cipolla. Understanding symmetries in deep networks, 2015.

[4] Dimitri P Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.

[5] Srinadh Bhojanapalli, Behnam Neyshabur, and Nati Srebro. Global optimality of local search for low rank matrix recovery. In *Advances in Neural Information Processing Systems*, pages 3873–3881, 2016.

[6] David Bindel. Non-negative matrix factorization (nmf), 2018.

[7] Loris Cannelli, Francisco Facchinei, Vyacheslav Kungurtsev, and Gesualdo Scutari. Asynchronous parallel algorithms for nonconvex big-data optimization. part i: Model and convergence. *arXiv preprint arXiv:1607.04818*, 2016.

[8] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Proceedings of Machine Learning Research*, pages 192–204, 2015.

[9] Richard Cole and Yixin Tao. An analysis of asynchronous stochastic accelerated coordinate descent. *arXiv preprint arXiv:1808.05156*, 2018.

[10] Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS'14*, page 2933–2941. MIT Press, 2014.

[11] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points online stochastic gradient for tensor decomposition. In *Conference on Learning Theory*, pages 797–842, 2015.

[12] Rong Ge, Jason D Lee, and Tengyu Ma. Matrix completion has no spurious local minimum. In *Advances in Neural Information Processing Systems 29*, pages 2973–2981. Curran Associates, Inc., 2016.

[13] Serge Gratton. Second-order convergence properties of trust-region methods using incomplete curvature information, with an application to multigrid optimization. *Journal of Computational Mathematics*, 24(6):676–692, 2006.

[14] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. How to escape saddle points efficiently. *arXiv preprint arXiv:1703.00887*, 2017.

[15] Chi Jin, Praneeth Netrapalli, Rong Ge, Sham M Kakade, and Michael I Jordan. On nonconvex optimization for machine learning: Gradients, stochasticity, and saddle points. *arXiv preprint arXiv:1902.04811*, 2019.

[16] Chi Jin, Praneeth Netrapalli, and Michael I Jordan. Accelerated gradient descent escapes saddle points faster than gradient descent. *arXiv preprint arXiv:1711.10456*, 2017.

[17] Kfir Y. Levy. The power of normalization: Faster evasion of saddle points. *CoRR*, 2016.

[18] Ji Liu and Stephen J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties, 2015.

[19] Ji Liu, Steve Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *International Conference on Machine Learning*, pages 469–477, 2014.

[20] Yurii Nesterov. Introductory lectures on convex programming volume i: Basic course, 1998.

[21] Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[22] Yurii Nesterov and Borris Polyak. Cubic regularization of newton method and its global performance. In *Mathematical Programming*, volume 108, pages 177–205, June 2006.

[23] Dohyung Park, Anastasios Kyrillidis, Constantine Carmanis, and Sujay Sanghavi. Non-square matrix sensing without spurious local minima via the burer-monteiro approach. In *Proceedings of Machine Learning Research*, volume 54, pages 65–74. PMLR, 20–22 Apr 2017.

[24] Michael JD Powell. On search directions for minimization algorithms. *Mathematical programming*, 4(1):193–201, 1973.

[25] Adepu Ravi Sankar and Vineeth N Balasubramanian. Are saddles good enough for deep learning?, 2017.

[26] J. Sun, Q. Qu, and J. Wright. Complete dictionary recovery over the sphere i: Overview and the geometric picture. *IEEE Transactions on Information Theory*, 63(2):853–884, 2017.

[27] Tao Sun, Robert Hannah, and Wotao Yin. Asynchronous coordinate descent under more realistic assumptions. In *Advances in Neural Information Processing Systems*, pages 6182–6190, 2017.

[28] Eran Treister and Javier S Turek. A block-coordinate descent approach for large-scale sparse inverse covariance estimation. In *Advances in Neural Information Processing Systems 27*, pages 927–935, 2014.

[29] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[30] Stephen Wright. Coordinate descent algorithms. In *Mathematical Programming*, volume 151, pages 3–34, March 2015.

[31] Yangyang Xu and Wotao Yin. A globally convergent algorithm for nonconvex optimization based on block coordinate update. *Journal of Scientific Computing*, 72(2):700–734, 2017.

[32] Yi Xu, Rong Jin, and Tianbao Yang. First-order stochastic algorithms for escaping from saddle points in almost linear time. *arXiv preprint arXiv:1711.01944*, 2017.

6 Appendix A: Algorithms

Algorithm 3: s = SWACD($\hat{x}, f, \eta, i$)

Input: Shared point $\hat{x} \in \mathbb{R}^N$ (the read coordinate information that may be outdated by the end of the algorithm), objective function f, learning rate (step-size) η, updating block i (containing coordinates c)

Output: The update s to the shared solution (product of the gradient and step size)

$\bar{x} \leftarrow \hat{x}$;
for $c \in i$ **do**
  $\bar{x} \leftarrow \bar{x} - \eta \nabla_c f(\bar{x}) \mathbf{e}_c$;
end
$s \leftarrow \bar{x} - \hat{x}$;
return s

Algorithm 4: $(n, x^{j+n}, E_{j+n}) = \text{GACD}(x^j, f, \eta, \tau, M, L)$

Input: A starting point $x^j \in \mathbb{R}^N$, objective function f, learning rate (step-size) η, delay bound τ, momentum threshold M, gradient-Lipschitz L

Output: Total iterations performed n, the point x^{j+n}, energy function E_{j+n} at that point

$\gamma \leftarrow j + \tau$;
while $j < \gamma$ **do**
  Choose Block i;
  $x^{j+1} - x^j \leftarrow \text{SWACD}(x^j, f, \eta, i)$;
  if $||x^j - x^{j+1}||_2 \geq M$ **then**
    $j \leftarrow j + 1$;
    break;
  end
  $j \leftarrow j + 1$;
} In Parallel
end
$n \leftarrow (j + \tau - \gamma)$;
$E_j = f(x^j) + \frac{L}{2} \sum_{k=j-\tau}^{j-1} (k - (j - \tau) + 1)||x^{k+1} - x^k||_2^2$;
return n, x^j, E_j

Algorithm 5: $(x^{j+1}, E_{j+1}) = \text{PACD}(x^j, f, \eta, \tau, r, T, L)$

Input: A starting point $x^j \in \mathbb{R}^N$, objective function f, learning rate (step-size) η, delay bound τ, perturbation radius r, escaping time bound T, gradient-Lipschitz L

Output: The following point x^{j+1} (after T steps of perturbation), energy function E_{j+1} at that point

$\xi \leftarrow$ uniformly $\sim \mathbb{B}(0, r)$;
$y^0 \leftarrow x^j + \xi$;
$t \leftarrow 0$;
while $t < T$ **do**
  Choose Block i;
  $y^{t+1} - y^t \leftarrow \text{SWACD}(y^t, f, \eta, i)$;
  $t \leftarrow t + 1$;
} In Parallel
end
$E_{j+1} = f(y^T) + \frac{L}{2} \sum_{k=T-\tau}^{T-1} (k - (T - \tau) + 1)||y^{k+1} - y^k||_2^2$;
$x^{j+1} = y^T$;
return x^{j+1}, E_{j+1}

7 Appendix B: Proof of Saddle Point Escape

Theorem 1 Proof: As mentioned in Theorem 1, f is a L-smooth and ρ-Hessian Lipschitz function with a bounded minimum value f^*, $\delta \in (0, 1)$ is the failure of escape probability, and the error threshold is defined as $\epsilon \leq \frac{L^2}{\rho}$. Let the N-dimensional perturbation ball centered at $\mathbf{x}$ with radius ηr be denoted as $B_{\mathbf{x}}^{(N)}(\eta r)$,

where η is the step-size and r is a radius hyperparameter.

From previous work (which is not included in this report due to lack of space but follows Lemma 11 and Lemma 15 in [14]), it has been determined that the thickness of the stuck region R_{stuck} is ηr_0. This result states that for an escaping time bound $T = \frac{\log_2(\frac{16\sqrt{N}\epsilon}{r\delta\sqrt{\pi}})}{\eta\sqrt{\rho\epsilon}}$ (the number of iterations of asynchronous coordinate descent after perturbation necessary to escape the saddle), the term ηr_0 is bounded below by $\frac{\eta r\delta\sqrt{\pi}}{2\sqrt{N}}$. Thus, for any two points along the direction of the minimum eigenvalue e_1 that are at least $\frac{\eta r\delta\sqrt{\pi}}{\sqrt{N}}$ away from each other, one point must not be in the stuck region. This is the thickness of R_{stuck} along the e_1 direction. Using calculus, the thickness ηr_0 can be turned into an upper bound on the volume of the stuck region.

Following similar steps as in [14, 16] (Jin et. al), let $I_{stuck}(\cdot)$ be the indicator function of being inside the set R_{stuck}. Let $x = (x^{(1)}, \bar{x})$, where $x^{(1)}$ is in the e_1 direction and $\bar{x}$ is the remaining $N-1$ dimensional vector. Finally, let $\tilde{x}$ represent the potential saddle point. We can determine the volume of R_{stuck} as:

$$Vol(R_{stuck}) = \int_{B_{\tilde{x}}^{(N)}(\eta r)} dx \cdot I_{stuck}(x) = \int_{B_{\tilde{x}}^{(N-1)}(\eta r)} d\bar{x} \int_{x^{(1)}-\sqrt{(\eta r)^2 - ||\bar{\tilde{x}}-\bar{x}||^2}}^{x^{(1)}-\sqrt{(\eta r)^2 - ||\bar{\tilde{x}}-\bar{x}||^2}} dx^{(1)} \cdot I_{stuck}(x) \tag{12}$$

Using the previous result that $\eta r_0 \geq \frac{\eta r\delta\sqrt{\pi}}{2\sqrt{N}}$, we can compute an upper bound:

$$Vol(R_{stuck}) \leq \int_{B_{\tilde{x}}^{(N-1)}(\eta r)} d\bar{x} \cdot \left(\frac{2\eta r\delta\sqrt{\pi}}{2\sqrt{N}}\right) = Vol(B_{\tilde{x}}^{(N-1)}(\eta r))\left(\frac{\eta r\delta\sqrt{\pi}}{\sqrt{N}}\right) \tag{13}$$

The volume ratio of the stuck region is now computed as:

$$\frac{Vol(R_{stuck})}{Vol(B_{\tilde{x}}^{(N)}(\eta r))} \leq \frac{Vol(B_{\tilde{x}}^{(N-1)}(\eta r))\left(\frac{\eta r\delta\sqrt{\pi}}{\sqrt{N}}\right)}{Vol(B_{\tilde{x}}^{(N)}(\eta r))} \tag{14}$$

The volume of a N-dimensional ball of radius r is defined as:

$$V_d(r) = \frac{\pi^{\frac{N}{2}}}{\Gamma(\frac{N}{2}+1)} r^N \tag{15}$$

From this definition in Equation 8, the ratio of volumes is simplified to:

$$\frac{Vol(R_{stuck})}{Vol(B_{\tilde{x}}^{(N)}(\eta r))} \leq \frac{\left(\frac{\eta r\delta\sqrt{\pi}}{\sqrt{N}}\right)\frac{\pi^{\frac{N-1}{2}}}{\Gamma(\frac{N-1}{2}+1)}(\eta r)^{N-1}}{\frac{\pi^{\frac{N}{2}}}{\Gamma(\frac{N}{2}+1)}(\eta r)^N} = \frac{\left(\frac{\eta r\delta\sqrt{\pi}}{\sqrt{N}}\right)\Gamma(\frac{N}{2}+1)}{\eta r\sqrt{\pi}\,\Gamma(\frac{N-1}{2}+1)} = \left(\frac{\delta}{\sqrt{N}}\right)\frac{\Gamma(\frac{N}{2}+1)}{\Gamma(\frac{N}{2}+1/2)} \tag{16}$$

By property of the gamma function, the following inequality holds: $\frac{\Gamma(x+1)}{\Gamma(x+1/2)} \leq \sqrt{x+\frac{1}{2}}$. This inequality simplifies the expression above to become:

$$\frac{Vol(R_{stuck})}{Vol(B_{\tilde{x}}^{(N)}(\eta r))} \leq \left(\frac{\delta}{\sqrt{N}}\right)\sqrt{\frac{N}{2}+\frac{1}{2}} \leq \left(\frac{\delta}{\sqrt{N}}\right)\sqrt{N} = \delta \tag{17}$$

Therefore we see that the volume ratio of the stuck region is extremely small:

$$\frac{Vol(R_{stuck})}{Vol(B_{\tilde{x}}^{(N)}(\eta r))} \leq \delta \tag{18}$$

With probability of at least $1-\delta$, a uniformly perturbed point will not fall within the stuck region.