AMSC 664 Final Presentation: Escaping From Saddle Points Using Asynchronous Coordinate Descent

Marco Bornstein
Advisor: Dr. Furong Huang

May 11th, 2021

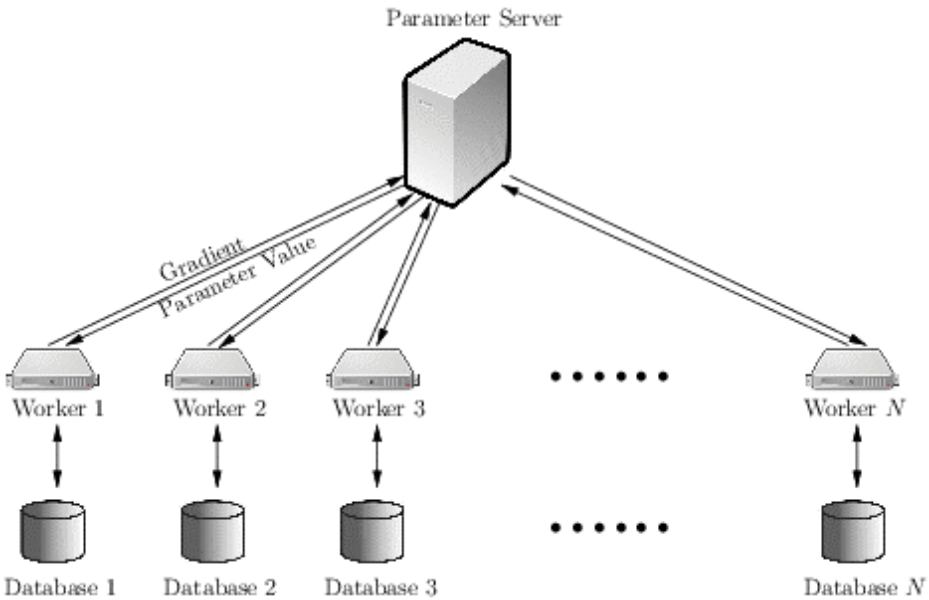
Table of Contents

Project Goal

The goal of my project is to implement and analyze my saddle escaping asynchronous coordinate descent (SEACD) algorithm. I aim to show that it efficiently minimizes a non-convex function with numerous parameters!

Parallel Computing Issues: Synchronization

- Parallel computing breaks data up and processes it simultaneously by multiple workers
- Some algorithms require all computed gradients be returned to the global server before the global solution is updated
- The speed of parallel computing is limited by the weakest link in the computational chain: the slowest worker and its consequent longest communication delay [3]



Solution: Asynchronous Coordinate Descent

- Asynchronous coordinate descent replaces the weakest link in the computational chain
- To accomplish this, each worker's computed gradient is no longer necessary to update the global solution
  - Instead, the global solution is updated, in real time, every time a worker finishes its computed gradient
  - Workers send back their computed gradient values and immediately use the updated global solution to continue its gradient update process
- Allowing each worker to update the global solution after its gradient is computed can potentially speed up parallel coordinate descent [2, 5]

Solution: Asynchronous Coordinate Descent

Asynchronous coordinate descent is defined by the following update rule:

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

- x^j is the current iterate
- η is the step-size
- i is the selected block of coordinates for a specific worker
- $\hat{x}^j$ is the delayed iterate

For all other non-updating blocks $e \neq i$, $x_e^{j+1} = x_e^j$.

$$\hat{x}^j = \left(x_1^{j-D(j,1)}, \; x_2^{j-D(j,2)}, \; \ldots, \; x_d^{j-D(j,d)} \right)$$

$$D(j) = \max_{1 \leq n \leq d} \{D(j,n)\} \leq \tau$$

Table of Contents

AMSC 664: Asynchronicity and Parallelization

The overarching themes of AMSC 664 for my project were:

- Asynchronicity: Creating a truly asynchronous algorithm
- Parallelization: Creating a fully parallel algorithm

My main goal for AMSC 664 was to incorporate both into the SEACD algorithm I implemented in AMSC 663.

AMSC 664: Accomplishments

I have been able to accomplish the following milestones during the course of AMSC 664:

1. Optimize hyperparameters for the SEACD algorithm
2. Implement and validate an asynchronous-like serial version of SEACD
3. Construct my first Convolutional Neural Network from scratch!
4. Create an efficient Parallel SEACD Module in Python and test it on a high-dimensional problem

Table of Contents

Optimizing Hyperparameters

I began this semester strengthening the theoretical results of my SEACD algorithm by better selecting its hyperparameters. This improvement is behind the scenes and difficult to illustrate, but includes:

- Explicitly selecting the step-size η, whereas before the user had to search for a feasible value
- Clearly prove that convergence only depends poly-logarithmically with dimension d
- Concretely determine that the convergence depends sub-linearly on the maximum delay τ

Implementing Serial-Asynchronous SEACD

I took the following steps to alter Algorithms 1 and 2 into a serial-asynchronous process:

1. Split up x^j into even chunks amongst the provided workers
2. Designate the worker with the final piece of x^j as the "slow" one

Algorithm 1: $(n, x^{j+n}, E_{j+n}) = \text{GACD}(x^j, f, \eta, \tau, M, L)$

$\gamma \leftarrow j + \tau + 1$
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
  $j \leftarrow j + 1$
end
$n \leftarrow (j + \tau + 1 - \gamma)$
$E_j = f(x^j) + \frac{L}{2} \sum_{k=j-\tau}^{j-1} (k - (j - \tau) + 1) ||x^{k+1} - x^k||_2^2$
return n, x^j, E_j

In Parallel

Implementing Serial-Asynchronous SEACD

3. For $\tau - 1$ iterations, update all workers except the "slow" one
4. On the τ^{th} iteration, update the "slow" worker
5. Repeat this process until convergence

Algorithm 2: $(x^{j+1}, E_{j+1}) = \text{PACD}(x^j, f, \eta, \tau, r, T, L)$

$\xi \leftarrow$ uniformly $\sim \mathbb{B}_{x^j}(r)$
$y^0 \leftarrow x^j + \xi,\ t \leftarrow 0$
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}

Validating Serial-Asynchronous SEACD Implementation

To test my implementation of the serial-asynchronous SEACD algorithm, I used it within a gradient-descent-based matrix factorization model. Given a solution matrix $S \in \mathbb{R}^{mxn}$ and two random matrices $U \in \mathbb{R}^{mxk}$ and $F \in \mathbb{R}^{kxn}$ (where k, m, n can be equal to one another), I use gradient descent to minimize the Mean Squared Error (MSE) between UF and S.

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

Matrix Factorization

An example of computing the MSE and gradient for an element of S for $S \in \mathbb{R}^{3x3}$, $U \in \mathbb{R}^{3x2}$, $F \in \mathbb{R}^{2x3}$ [4] is given as:

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

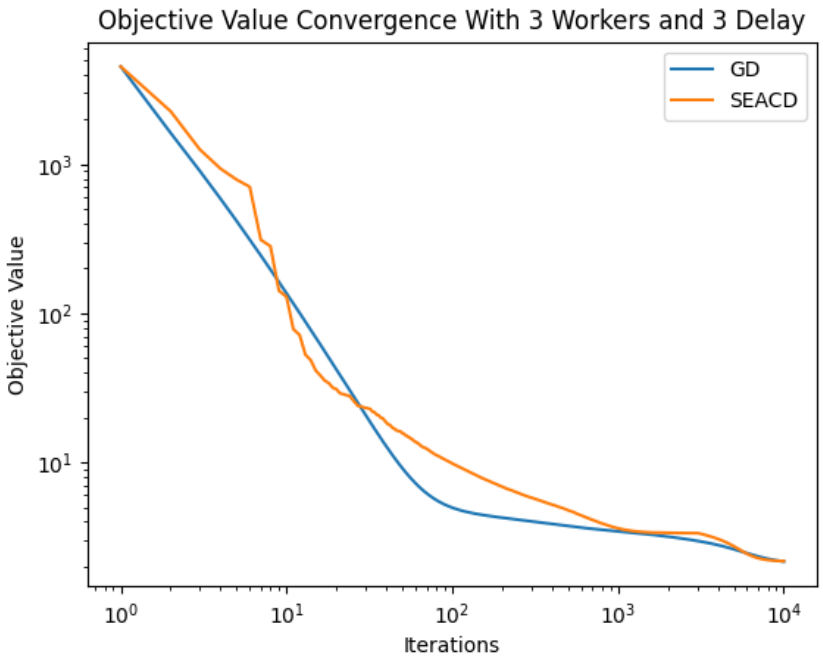
The gradient of $S_{1,1}$ with respect to one of the elements $U_{1,1}$ we are searching for the optimal value of is:

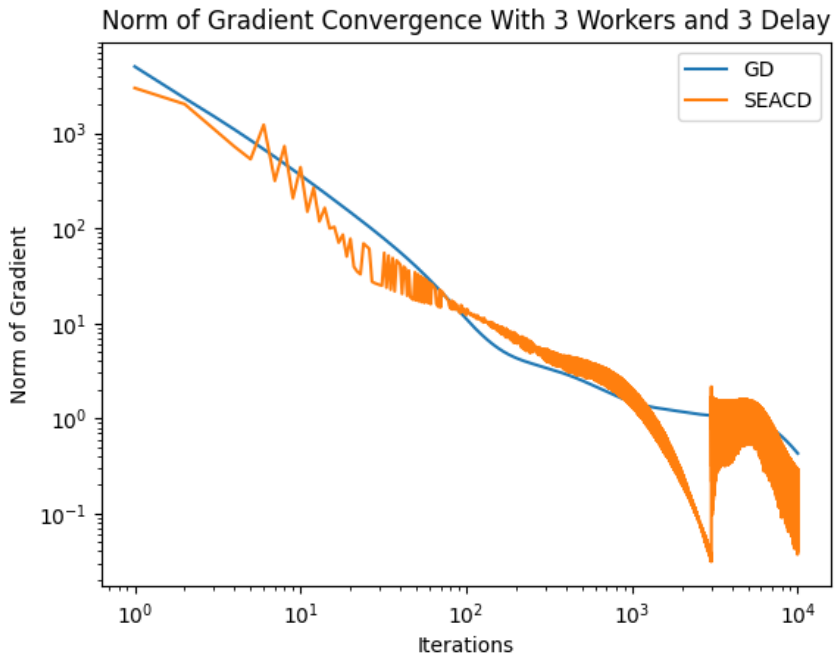
$$\frac{\partial MSE(S_{1,1})}{\partial U_{1,1}} = -2F_{1,1} * (S_{1,1} - [(U_{1,1} * F_{1,1}) + (U_{1,2} * F_{2,1})])$$

In this example, the elements $S_{1,2}$ and $S_{1,3}$ are also affected by the value of $U_{1,1}$. Thus, to update $U_{1,1}$, I average its gradient values with respect to the elements it affects within the solution matrix S.

Matrix Factorization Results

Here are results from the matrix factorization problem comparing the convergence of GD and SEACD. The dimensions of the matrices are $U \in \mathbb{R}^{10x2}$, $F \in \mathbb{R}^{2x10}$, and $S \in \mathbb{R}^{10x10}$ with 3 asynchronous workers and $\tau = 3$ for SEACD.





Convolutional Neural Network: MNIST Classification

One of the goals I aimed to accomplish this semester was to train a neural network with my SEACD algorithm. I built a simple CNN in Python (using TensorFlow), based on a live-script MATLAB example, with 1 convolution layer, 1 max pooling layer, and 2 densely connected layers. Using this CNN, I attempted to classify handwritten digits in the MNIST dataset.

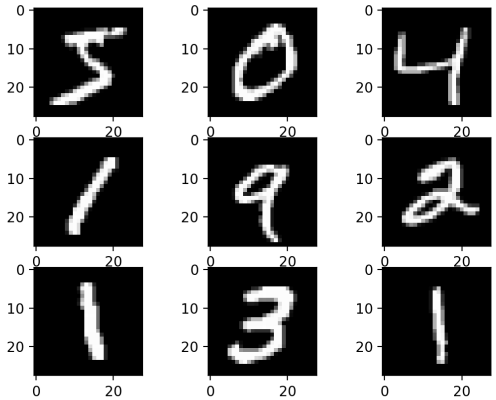
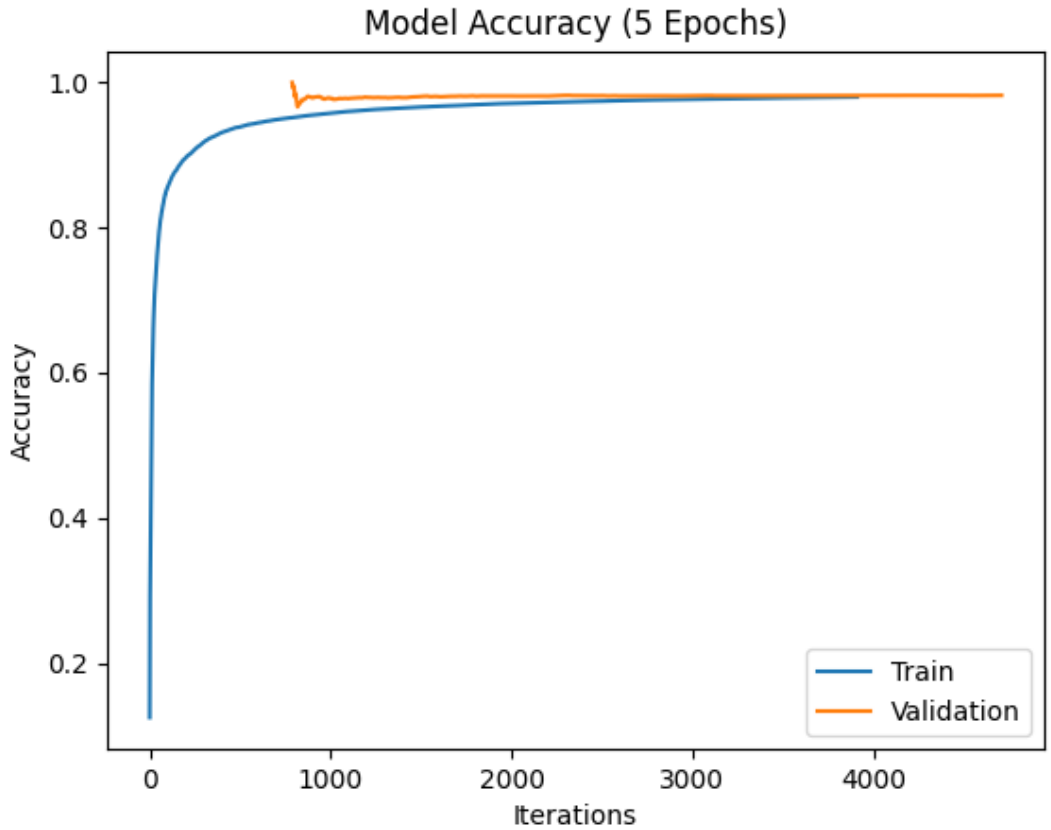


Figure: Example Images from MNIST

The MNIST dataset is quite large, containing 60,000 28 x 28 pixel images.

Convolutional Neural Network: MNIST Classification

Due to the nature of my algorithm (requiring the function f) I was not able to implement it within TensorFlow as an optimizer to train the CNN. Instead, I used ADAM and achieved the following results:



Implementing Parallel-Asynchronous SEACD

My final goal for AMSC 664 was to successfully implement my SEACD algorithm in parallel. Specifically, I aimed to attain the following:

1. Achieve convergence to local optima
2. Obtain a run-time speed-up compared to serial gradient descent methods
3. Create a Python Module for the Parallel-Asynchronous SEACD algorithm

Issues Implementing Parallel-Asynchronous SEACD

Parallelization is usually accomplished in a straightforward manner using the Multiprocessing package. The issue that made this process difficult for SEACD was that the processes, or workers, need to communicate with each other throughout the course of my algorithm. This is easier to accomplish in a language like C/C++ (using MPI).

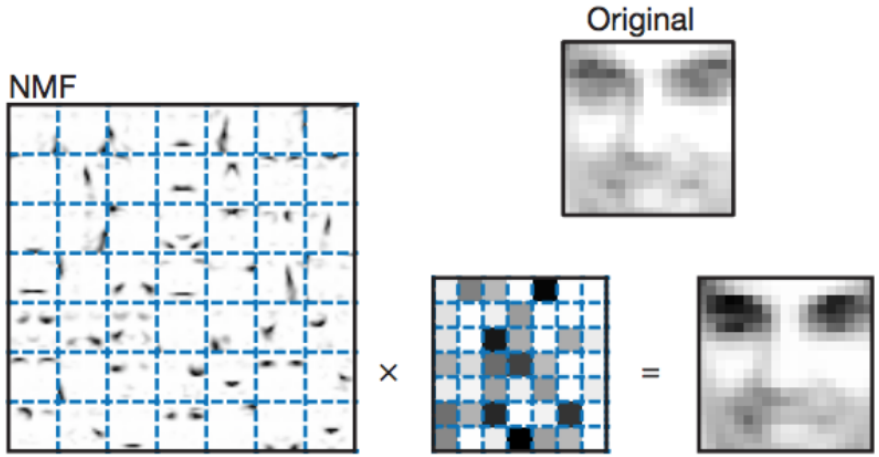
Solutions to Implementing Parallel-Asynchronous SEACD

Within the Multiprocessing package, these are the avenues I traversed to get SEACD up and running efficiently:

- Multiprocessing Managers: control a server process and allows other processes to manipulate them using proxies (too slow)
- Queues: first in first out (FIFO) data structure that allows processes to communicate safely (blocking occurs)
- ⋆ Shared Memory: allocate and manage shared memory that can be accessed by processes (new to Multiprocessing and complicated)

Validating Parallel-Asynchronous SEACD Implementation

To test my implementation of the parallel-asynchronous SEACD algorithm, I used it to solve a large-scale non-negative matrix factorization (NMF) problem. In the NMF problem, a non-negative matrix $A \in \mathbb{R}_+^{m \times n}$ is sought to be factorized into two non-negative, and often lower-ranked, matrices $W \in \mathbb{R}_+^{m \times k}$ and $H \in \mathbb{R}_+^{k \times n}$ [1].



Non-Negative Matrix Factorization

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 $P(x) = [x]_+$. The Projected GD update rule becomes:

$$x^{j+1} = P\left(x^j - \eta \nabla f(x^j)\right)$$

As provided in [1], the Projected GD gradient values for W and H are:

$$\nabla W = (WH - A)H^T$$
$$\nabla H = W^T(WH - A)$$

NMF Factorization With (Projected) GD and SEACD

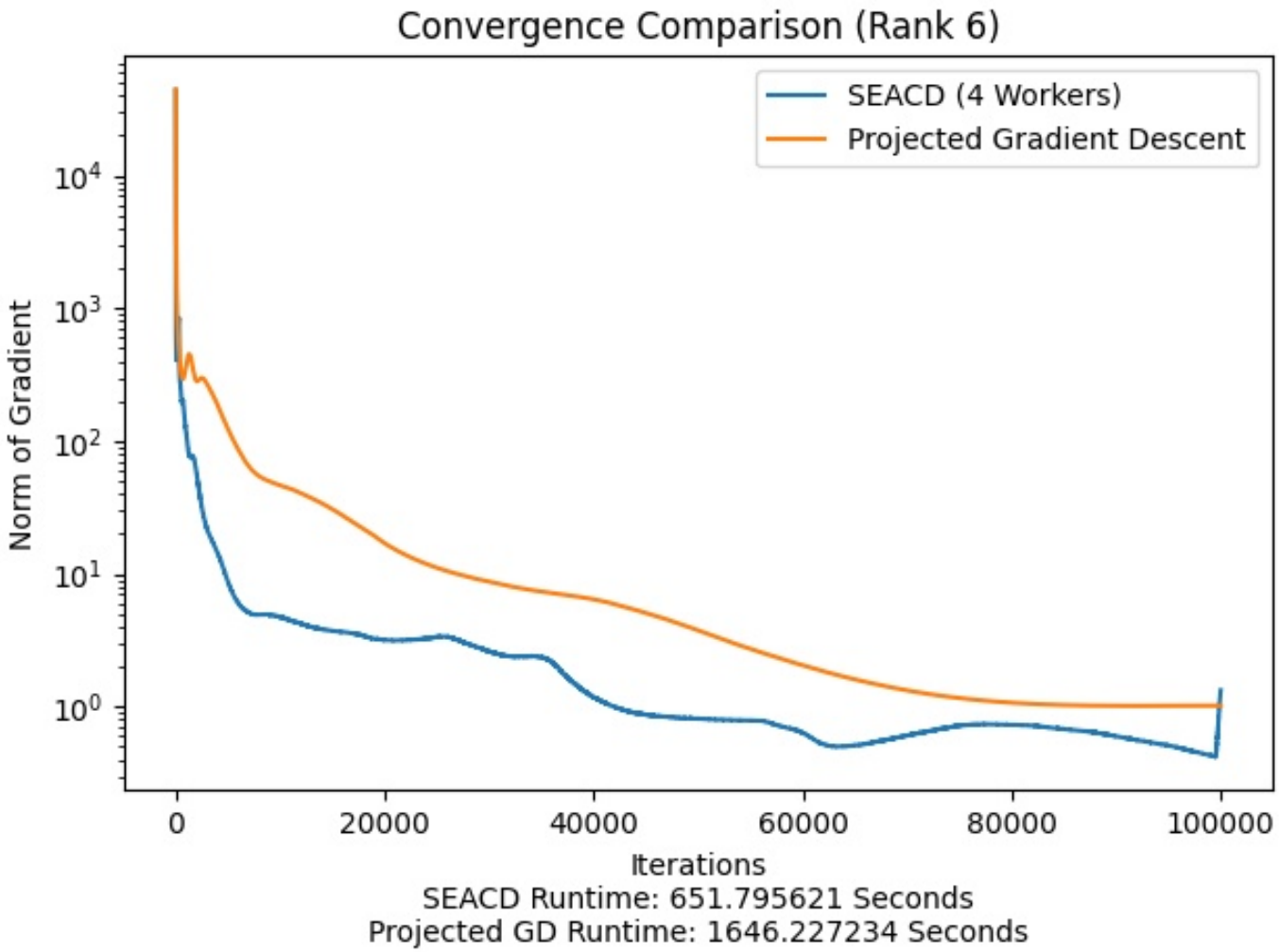
To begin, a randomized initial vector x^0 is inputted into Projected GD and Projected SEACD. An iterate x^j within Projected GD and Projected SEACD is a vector containing all elements of both W and H:

$$x^j = \left(x^j_{1W}, x^j_{2W}, \ldots x^j_{(mk)W}, x^j_{1H}, x^j_{2H}, \ldots x^j_{(kn)H}\right)$$

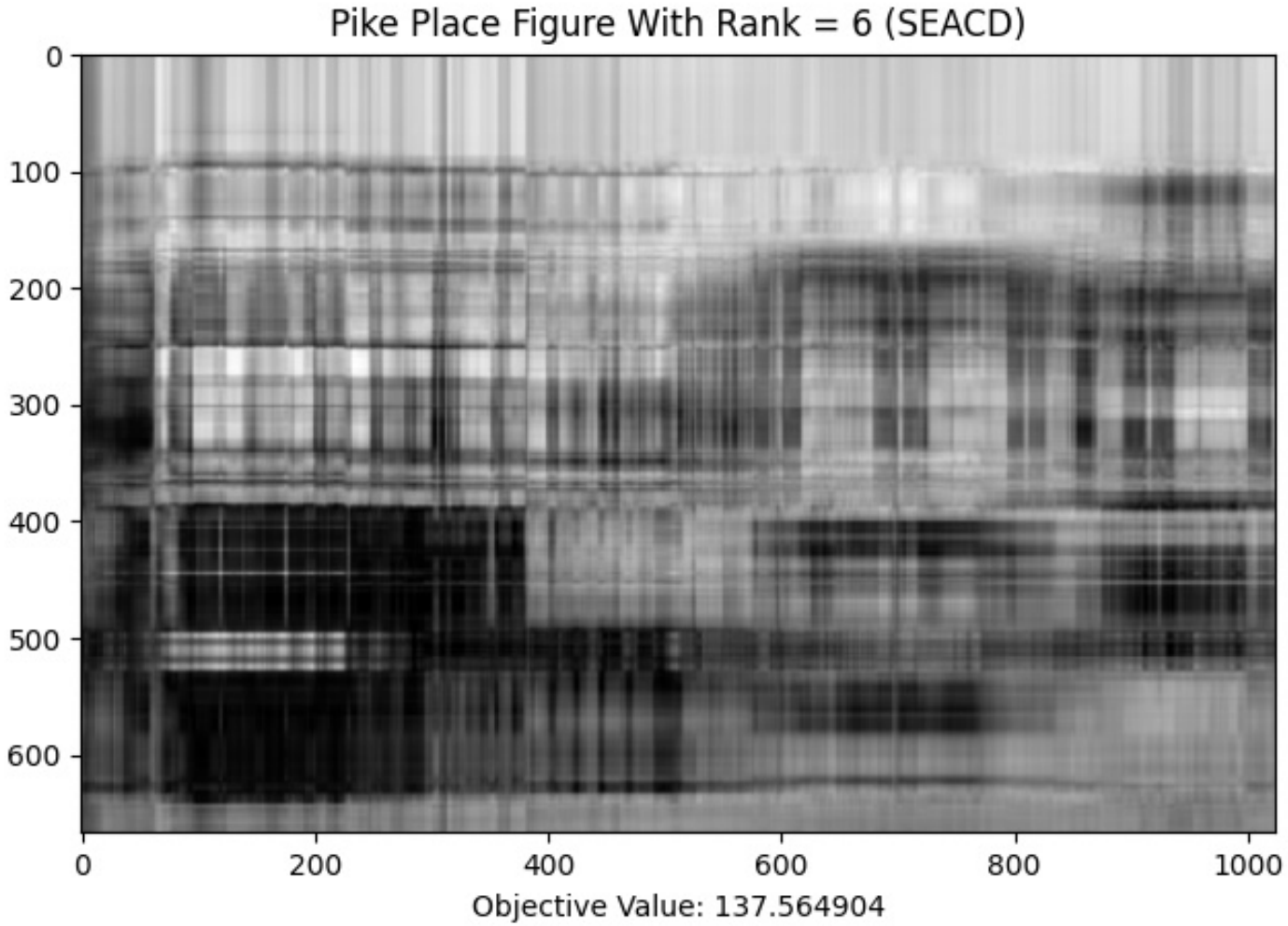
- The element x^j_{1W} is the top left element of W at iterate j and $x^j_{(mk)W}$ is the bottom right element of W at iterate j
- To reconstruct W and H within the algorithm, I simply reshape the left and right halves of x^j
- Besides hyperparameters, the only inputs into Projected GD and Projected SEACD are x^0, the gradient function, and the objective function (MSE)

Non-Negative Matrix Factorization

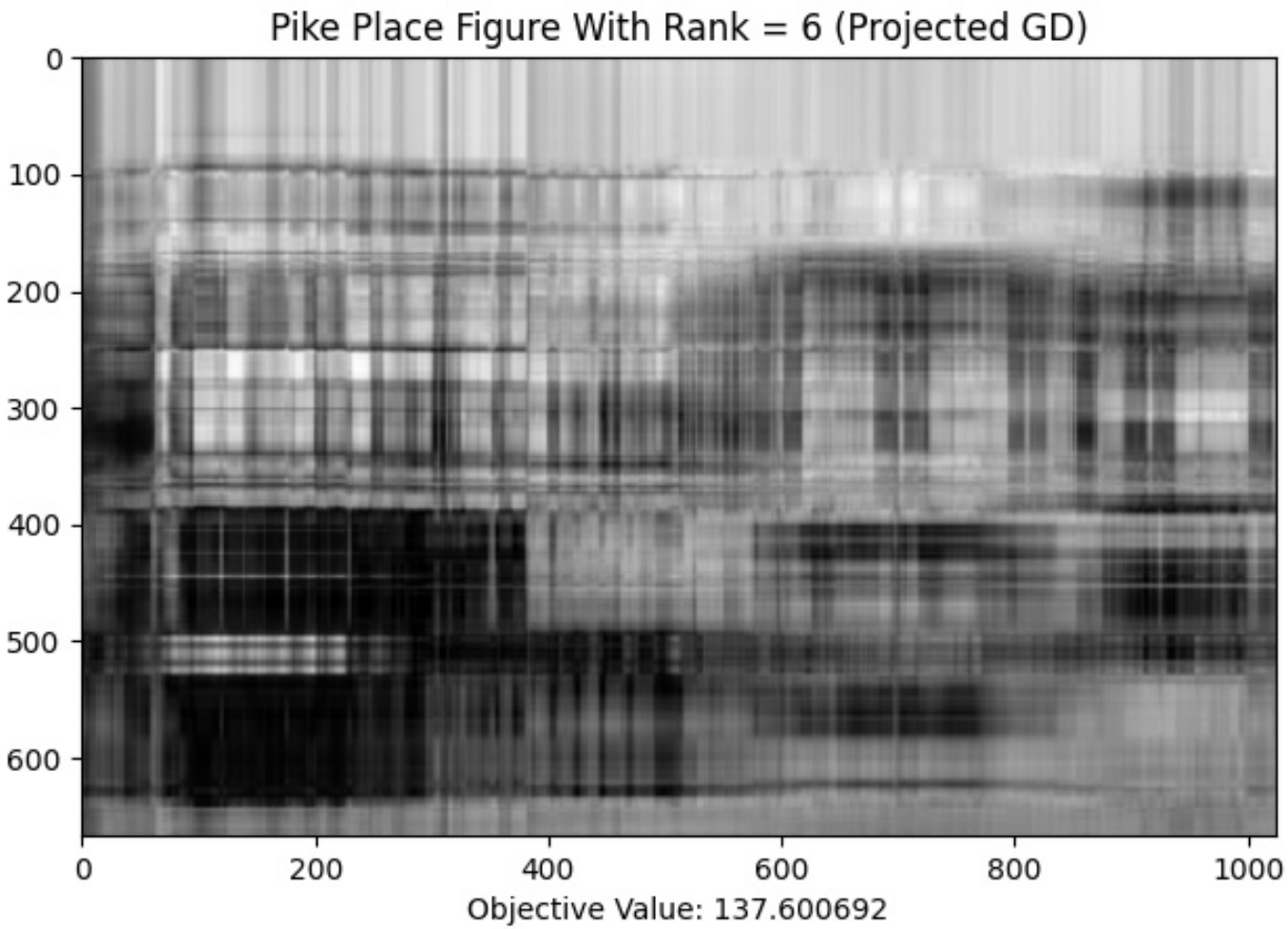
Convergence Comparison



Pike Place Animation SEACD



Pike Place Animation Projected GD



Parallel Success: Speed-Up Achieved

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

Rank k	Proj. GD Runtime (s)	Proj. 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

Table of Contents

Deliverables

All of my code and documentation can be found in my GitHub repository: `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
- Non-Negative Matrix Factorization test code
- Figures for Matrix Factorization convergence, CNN accuracy, and NMF convergence

Acknowledgements

Thank you so much to Dr. Balan and Dr. Cameron for their guidance, critiques, and advice on my project. Also, thanks to my advisor Dr. Huang for her support throughout both semesters. Finally, thank you to my close friend Michael Blankenship for his help navigating the Multiprocessing package in Python.

Table of Contents

References I

[1] David Bindel.

Non-negative matrix factorization (nmf), 2018.

[2] 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.

[3] Ji Liu and Stephen J. Wright.

Asynchronous stochastic coordinate descent: Parallelism and convergence properties, 2015.

[4] Jacob Moore.

Python: Implementing matrix factorization from scratch!, 2020.

[5] 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.