Continuous matrix completion

Sean Soon

August 4, 2025

Contents

1 Introduction

Matrix completion is the problem of recovering a matrix from a sampling of its entries. This problem is inherently ill-posed since, with fewer samples than entries, there are infinitely many valid completions. It is impossible to determine which of these candidate completions is indeed the 'correct' solution without additional information. Fortunately, the matrix to be recovered is often (approximately) low-rank. For example, in recommendation systems, users rate only a subset of entries in a database, and vendors aim to recommend entries based on inferred preferences [20]. To make accurate recommendations, vendors must estimate a user's rating for unseen items based on their limited observed ratings.

A well-known instance of this problem is the Netflix problem [3]. Each row of the data matrix represents a user while each column represents a movie. Each entry in the matrix corresponds to a user's rating of a movie; however, this matrix is sparse as users typically rate only a few movies. To provide relevant suggestions to users, Netflix would like to complete this matrix. It can be assumed that the full matrix is approximately low-rank since it is commonly believed that only a few factors govern a user's preferences.

Low-rank matrix completion also appears in a range of other applications such as model reduction [17], pattern recognition [19], and machine learning [1, 2]. It has been extensively studied over the past few decades, with many algorithms specifically designed for low-rank matrix completion. For example, alternating steepest descent (ASD) [26] is a popular completion algorithm with several variants such as ScaledASD [26] and LoopedASD [27] that improve its performance. However, each iteration of ASD requires computing the gradient of the loss function over all the known entries, which has a high computational cost for recovering large data sets. Stochastic optimisation methods such as stochastic gradient descent (SGD) [4] and Adam [16] have a lower computational per-iteration cost, which makes them faster than standard gradient descent methods. Furthermore, both algorithms often achieve better minima on non-convex objectives in practice. As such, they are widely used in general machine learning problems and neural net-

works [31].

In imaging applications, images are traditionally stored as a grid of pixels. For example, x-ray spectromicroscopy data can be represented by a low-rank matrix if the specimen only contains a few elements [27], similar to the Netflix problem. Each entry of the data matrix corresponds to the intensity of a pixel. The scanning times and the radiation dosage on the specimen can be reduced using sparse scans from which the full image can be reconstructed. However, various factors, such as limitations in machine precision and fluctuations in the ambient temperature, can distort the image. The specimen also heats up during scans by absorbing the radiation, which causes it to expand. These distortions inflate the rank of the matrix, which increases the reconstruction error.

In this paper, we present two variants of ASD that utilise mini-batching to achieve a lower computational per iteration cost than the standard ASD. We also benchmark SGD and Adam against the standard ASD in the context of low-rank matrix completion. Additionally, we propose an alternative image representation model where the data set is stored as a piecewise bilinear interpolation, which can be used to recover the traditional pixel-based image more accurately when the actual positions of the pixels are distorted.

The paper is organised as follows. We begin in Section 2 with a background of matrix completion. SGD, Adam, ASD, and its mini-batch variants are presented in the context of low-rank matrix completion in Section 3. We then propose the continuous image representation model based on bilinear interpolation in Section 4. Finally, the numerical experiments presented in Section 5 contrast the aforementioned algorithms.

2 Low-rank matrix completion

Low-rank matrix completion aims to find a matrix with the lowest possible rank that is consistent with the known entries. This problem can be explicitly expressed as

$$\min_{A\in\mathbb{R}^{n_1\times n_2}} \operatorname{rank}(A), \quad \text{subject to} \quad P_\Omega(A) = P_\Omega(A_0), \tag{2.1}$$

where $A_0 \in \mathbb{R}^{n_1\times n_2}$ is the underlying matrix to be reconstructed, $\Omega \subset \{1,\dots n_1\}\times\{1,\dots,n_2\}$ is the set of indices of the known entries in A_0, and P_Ω is the associated *sampling operator* defined by

$$P_\Omega(A) = \begin{cases} A_{ij}, & (i,j) \in \Omega, \\ 0, & (i,j) \notin \Omega. \end{cases} \tag{2.2}$$

Equation (2.1) reframes the original matrix completion problem as seeking the simplest explanation that fits the observed data. However, this problem is non-convex and generally NP-hard [13] because of the rank objective. A widely studied approach to circumvent these issues is to replace rank function with the nuclear norm, which is defined as

$$\|A\|_* := \sum_{k=1}^{n} \sigma_k(A), \tag{2.3}$$

where $\sigma_k(A)$ denotes the kth largest singular value of X. Equation (2.1) can then be rewritten as its *convex* relaxation,

$$\min_{A \in \mathbb{R}^{n_1 \times n_2}} \|A\|_*, \quad \text{subject to} \quad P_\Omega(A) = P_\Omega(A_0). \tag{2.4}$$

Unlike the rank function, the nuclear norm is convex, which guarantees that any solution to Equation (2.4) is indeed the most optimal solution. Moreover, Equation (2.4) has been proven to be a highly structured semidefinite program [6, 11] for which many efficient solvers like SDPT3 [28] are available.

Instead of seeking the reconstruction with the lowest possible rank, suppose that we aim to recover the underlying matrix A_0 for a given rank r, which may differ from the true rank of A_0. Any matrix $A \in \mathbb{R}^{n_1 \times n_2}$ with rank r can be factorised as

$$A = XY^\top \quad \text{with} \quad X \in \mathbb{R}^{n_1 \times r}, Y \in \mathbb{R}^{n_2 \times r}. \tag{2.5}$$

Based on this factorisation, the problem of matrix completion for a given rank r is equivalent to minimising the scaled MSE (mean square error) between the known entries in the underlying matrix A_0 and those in the reconstruction $XY^\top$. This can be formulated as

$$\min_{X \in \mathbb{R}^{n_1 \times r}, Y \in \mathbb{R}^{n_2 \times r}} L_\Omega(X,Y), \quad L_\Omega(X,Y) = \frac{1}{2}\|P_\Omega(A_0) - P_\Omega(XY^\top)\|_F^2, \tag{2.6}$$

where the Frobenius norm $\|\cdot\|_F$ is defined for a matrix $A \in \mathbb{R}^{n_1 \times n_2}$ by

$$\|A\|_F^2 := \sum_{i,j} A_{ij}^2. \tag{2.7}$$

While this problem is non-convex with respect to both factors X and Y unlike nuclear norm minimisation in Equation (2.4), it can often be solved much more efficiently. Additionally, the loss function $L_\Omega(X, Y)$ can be minimised using general optimisation algorithms such as SGD and Adam. These benefits will be further explored in Section 3.

2.1 Well-posedness

It is not generally possible to recover a low-rank matrix from a subset of its entries. For example, consider the rank-1 matrix

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \in \mathbb{R}^{n \times n}.$$

If the only non-zero entry (in the top-left) is not sampled, then the matrix appears to be entirely zero and it is impossible to determine otherwise. Under uniform random sampling, the probability of observing that entry is $1/n^2$. Therefore, to guarantee its inclusion, one would need to sample all n^2 entries, which is equivalent to observing the entire matrix.

It has been proven that any low-rank matrix can be recovered by solving the convex relaxation problem in Equation (2.4) if its singular vectors are weakly correlated to the standard basis [6]. Given a matrix $A \in \mathbb{R}^{n_1 \times n_2}$ of rank r, its SVD (singular value decomposition) is defined as

$$A = \sum_{k=1}^{r} \sigma_k(A) \mathbf{u}_k \mathbf{v}_k^\top, \tag{2.8}$$

where $\mathbf{u}_k \in \mathbb{R}^{n_1}$ and $\mathbf{v}_k \in \mathbb{R}^{n_2}$ are the corresponding left and right singular vectors, respectively, of the kth largest singular value of A.

2.1.1 Sampling pattern

A matrix clearly cannot be reconstructed if a row or column is not sampled. There is also empirical evidence that the completion is more consistent if the known entries are evenly spread across the matrix [27]. To satisfy both criteria, *Bernoulli sampling* can be used: each entry is sampled i.i.d. (identically and independently distributed) with probability p. p corresponds to the *undersampling ratio* δ: the proportion of known entries from $A_0 \in \mathbb{R}^{n_1 \times n_2}$.

$$\delta = \frac{|\Omega|}{n_1 n_2} \tag{2.9}$$

This sampling method is often used in practice since it is usually impossible to predict which entries will be known. In some settings like spectromicroscopy, the user has complete control over the data acquisition, and it may be more efficient to collect data using different sampling methods like *robust raster sampling* [27].

3 Completion algorithms

There are many computationally efficient algorithms that successfully solve the problem of low-rank matrix completion in practice. These algorithms can generally be classified as either directly targeting the *non-convex* problem of Equation (2.1) or solving its *convex* relaxation in Equation (2.4). Many of the algorithms specifically designed for solving either of these problems utilise *iterative thresholding*. Methods that target the non-convex problem often use iterative *hard* thresholding: a hard thresholding operator sets all but a specified number of singular values to zero. In contrast, methods for solving the convex relaxation are usually based on iterative *soft* thresholding: a soft thresholding operator shrinks the singular values towards zero by a specified amount [25].

The most direct implementation of these algorithms requires computing a partial SVD in each iteration. For any $n_1 \times n_2$ matrix, this has computational complexity $O(n_1^3)$, assuming that n_1 and n_2 are directly proportional to each other. As such, computing this SVD dominates the computational per-iteration cost, which limits the applicability of these methods for recovering large matrices. Since this SVD computation is needed to minimise the rank of the reconstruction, some algorithms avoid computing a SVD entirely by ignoring the rank objective in Equation (2.1). They instead focus on re-

covering the underlying matrix for a given rank r as per Equation (2.6), making them much faster.

3.1 Data-splitting

Training an algorithm on all the known entries could lead to *over-fitting*: the reconstruction error is small over the training data, but large for new, unseen entries. A standard data-splitting scheme was used to mitigate this. The observed index set Ω was randomly shuffled, then split into a training set Ω_{train} and test set Ω_{test}. This splitting was done such that Ω_{train} can be partitioned into B mini-batches, denoted $\Omega_1, \ldots, \Omega_B$, of equal size, and Ω_{train} was approximately 90% the size of Ω. The remaining entries were then allocated to Ω_{test}.

3.2 Early stopping

Some of the algorithms have slow convergence rates as the iterates approach the optimal solution. Thus, two stopping conditions were set on all the algorithms: a maximum number of iterations, and a tolerance on the change in the test loss between iterations t and $t-\Delta t$. This change can be expressed as

$$\frac{L_{\Omega_{\text{test}}}(X_t, Y_t)}{L_{\Omega_{\text{test}}}(X_{t-\Delta t}, Y_{t-\Delta t})}, \quad t > \Delta t.$$

The iterates X_{best}, Y_{best} with the lowest test loss (not the last iterates) were returned as the final output since the algorithms may diverge or oscillate after reaching an optimal solution.

3.3 Stochastic gradient descent

Let $\nabla_X L_\Omega(X, Y)$ denote the gradient of $L_\Omega(X, Y)$ with respect to X, and similarly for Y. Gradient descent is an iterative algorithm in which the solution is improved by taking a step from the current point along the negative of the gradient of the objective function [24]. For Equation (2.6), the update step for each iteration t of gradient descent can be written as

$$X_t = X_{t-1} - \eta \cdot \nabla_X L_\Omega(X_{t-1}, Y_{t-1}), \tag{3.1}$$
$$Y_t = Y_{t-1} - \eta \cdot \nabla_Y L_\Omega(X_{t-1}, Y_{t-1}), \tag{3.2}$$

where η is the learning rate. Computing the gradient over the whole data set often has a high computation cost. As such, gradient descent can be incredibly slow and inapplicable for data sets that are too large for memory [22], which limits its applicability in practice. Stochastic gradient descent (SGD) is a simple yet effective algorithm that circumvents these drawbacks by taking a step along a random direction instead. It is only required that the expected value of this direction be the negative of the gradient of the risk function. Let $\ell_{i,j}(X, Y)$ denote the scaled pointwise MSE between the entries with index (i, j) in A_0 and $XY^\top$.

$$L_\Omega(X, Y) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} \ell_{i,j}(X, Y) \tag{3.3}$$

$$\ell_{i,j}(X, Y) = \frac{|\Omega|}{2} \left((A_0)_{i,j} - \sum_{k=1}^{r} X_{i,k} Y_{j,k} \right)^2 \tag{3.4}$$

The gradient of this pointwise loss function can be taken as the random direction for SGD since it can be shown that

$$\mathbb{E}\big(\nabla \ell_{i,j}(X, Y)\big) = \nabla L_\Omega(X, Y). \tag{3.5}$$

Computing the gradient over one training example is much cheaper and faster than doing so over the whole training set. Moreover, this also allows SGD to *learn online*: learn incrementally from data in a sequential manner. Compared to traditional batch learning methods like gradient descent, online learning models are highly scalable and responsive to new data, making them well-suited for dynamic, large-scale applications [14].

Although gradient descent is a slow optimisation method, it offers the most stable and smoothest convergence to the nearest local minimum within a few iterations. In contrast, SGD updates the parameters much more frequently and with a higher variance since significantly less training data is used in each update. This variance causes the iterates to fluctuate heavily, enabling SGD to escape poor local minima and discover new, potentially better local minima when navigating a non-convex loss surface. This behaviour is particularly relevant when optimising both X and Y simultaneously with respect to $L_\Omega(X, Y)$. However, this high variance also slows the convergence of SGD to an exact minimum.

Mini-batching is commonly used to counteract this. Instead of a single data point, the gradient is computed over a mini-batch of training examples to better estimate $\nabla L_\Omega(X, Y)$. Mini-batching allows SGD to strike a balance between the speed of stochastic updates and the stability of standard gradient descent. Algorithm 1 outlines our implementation of SGD with mini-batching for solving Equation (2.6).

Algorithm 1 Stochastic gradient descent (SGD) with mini-batching

Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: number of mini-batches B
Require: training mini-batches $\Omega_1, \ldots, \Omega_B$
Require: test set Ω_{test}
Require: constant learning rate $\eta > 0$
Require: maximum number of iterations $K \in \mathbb{N}$
$b \leftarrow 1$ ▷ Mini-batch counter.
for $t = 1$ **to** T **do**
  Compute $X_t = X_{t-1} - \eta \cdot \nabla_X L_{\Omega_b}(X_{t-1}, Y_{t-1})$
  Compute $Y_t = Y_{t-1} - \eta \cdot \nabla_Y L_{\Omega_b}(X_{t-1}, Y_{t-1})$
  if $L_{\Omega_{\text{test}}}(X_t, Y_t) < L_{\Omega_{\text{test}}}(X_{\text{best}}, Y_{\text{best}})$ **then**
    $X_{\text{best}} \leftarrow X_t$ ▷ Update the best iterates.
    $Y_{\text{best}} \leftarrow Y_t$
  end if
  if stopping conditions are reached **then**
    break
  end if
  if $b < B$ **then** ▷ Update the mini-batch counter.
    $b \leftarrow b + 1$
  else
    $b \leftarrow 1$
  end if
end for
return $X_{\text{best}} \approx X^*, Y_{\text{best}} \approx Y^*$

3.4 Adam

SGD in Algorithm 1 uses a fixed learning rate η to update both X and Y in every iteration. However, the optimal learning rate typically changes across iterations. A common solution to improve the convergence of SGD is to

Algorithm 2 Adam with mini-batching

Require: step size $\alpha > 0$
Require: exponential decay rates for moment estimates $\beta_1, \beta_2 \in [0, 1)$
Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: number of mini-batches B
Require: training mini-batches $\Omega_1, \ldots, \Omega_B$
Require: test set Ω_{test}
Require: maximum number of iterations $T \in \mathbb{N}$

$b \leftarrow 1$
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
for $t = 1$ **to** T **do**
  $g_t \leftarrow \begin{pmatrix} \nabla_X L_{\Omega_b}(X_{t-1}, Y_{t-1}) \\ \nabla_Y L_{\Omega_b}(X_{t-1}, Y_{t-1}) \end{pmatrix}$
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ $g_t^2 := g_t \odot g_t$
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Initialisation bias correction for m_t
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Initialisation bias correction for v_t
  $\begin{pmatrix} X_t \\ Y_t \end{pmatrix} \leftarrow \begin{pmatrix} X_{t-1} \\ Y_{t-1} \end{pmatrix} - \alpha \cdot \dfrac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$ ▷ $\varepsilon = 10^{-8}$
  if $L_{\Omega_{\text{test}}}(X_t, Y_t) < L_{\Omega_{\text{test}}}(X_{\text{best}}, Y_{\text{best}})$ **then**
    $X_{\text{best}} \leftarrow X_t$ ▷ Update the best iterates.
    $Y_{\text{best}} \leftarrow Y_t$
  end if
  if stopping conditions are reached **then**
    break
  end if
  Update b.
end for
return $X_{\text{best}} \approx X^*, Y_{\text{best}} \approx Y^*$

gradually shrink the learning rate as the iterates approach a local minimum, but this introduces additional hyper-parameters that require further tuning. Moreover, the parameters X and Y often do not share the same optimal learning rate.

Adam [16] is a widely used algorithm that computes adaptive learning rates for each parameter using the first and second moments of their respective gradients. The algorithm updates exponential moving averages, denoted m_t and v_t, of the gradient and squared gradient, respectively. Since m_t and v_t are initialised with zeros, they are biased estimates of first (mean) and second (uncentred variance) raw moments, respectively, of the gradient. This initialisation bias can be easily corrected by computing the bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$ as outlined in Algorithm 2.

Adam offers several advantages over SGD. Unlike SGD, the magnitude of the parameter updates is invariant to the scale of the gradient and often approximately bounded by the step-size hyper-parameter α. Additionally, Adam is suitable for problems with sparse gradients, similar to AdaGrad [8] which also utilises adaptive learning rates. Adam also naturally performs a form of step-size annealing. As Adam converges towards a minimum, the ratio $\frac{\hat{m}_t}{\sqrt{\hat{v}_t}}$ tends towards 0, which leads to progressively smaller parameter updates. This automatic annealing reduces the need for manually tuning a decaying learning rate schedule and enhances stability during optimisation.

3.5 Alternating steepest descent

Algorithm 3 Power Factorisation

Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: index set Ω
1: **for** t = 1, ... **do**
2: Fix Y_{t-1} and solve $X_t = \arg\min_{X \in \mathbb{R}^{n_1 \times r}} \|P_\Omega(A_0) - P_\Omega(XY_{t-1}^\top)\|_F^2$
3: Fix X_t and solve $Y_t = \arg\min_{Y \in \mathbb{R}^{n_2 \times r}} \|P_\Omega(A_0) - P_\Omega(X_t Y^\top)\|_F^2$
4: **if** stopping conditions are reached **then**
5: **break**
6: **end if**
7: **end for**

Many algorithms developed specifically to solve Equation (2.6) usually adopt an alternating minimisation scheme since $L_\Omega(X, Y)$ is convex with respect to each factor while the other is held fixed. In each iteration, X is optimised for

while Y is fixed, then Y is optimised while X is fixed. Such an approach is widely used in optimisation due to its simplicity, low memory requirements, and flexibility [10]. A well-known example is PowerFactorisation [12] as outlined in Algorithm 3. It applies the alternating minimisation scheme directly to Equation (2.6).

Each sub-problem in Algorithm 3 is a standard least squares problem, which gives PowerFactorisation a high computational per-iteration cost. To improve computational efficiency, each least squares sub-problem can be replaced with a single step of simple line-search along the gradient descent directions. This gives the alternating steepest descent (ASD) algorithm [26] outlined in Algorithm 4. It applies gradient descent to $L_\Omega(X, Y)$ alternatively with respect to X and Y. The exact learning rates η_X and η_Y for updating X and Y, respectively, in each iteration of ASD are given by

$$\eta_X = \frac{\|\nabla_X L_\Omega(X,Y)\|_F^2}{\|P_\Omega(\nabla_X L_\Omega(X,Y)Y^\top)\|_F^2}, \quad \eta_Y = \frac{\|\nabla_Y L_\Omega(X,Y)\|_F^2}{\|P_\Omega(X[\nabla_Y L_\Omega(X,Y)]^\top)\|_F^2}. \tag{3.6}$$

Algorithm 4 Alternating steepest descent (ASD)

Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: index set Ω
Require: maximum number of iterations $T \in \mathbb{N}$
for $t = 1$ **to** T **do**
  ▷ Fix Y and update X.
  Compute $\eta_X = \dfrac{\|\nabla_X L_\Omega(X_{t-1}, Y_{t-1})\|_F^2}{\|P_\Omega(\nabla_X L_\Omega(X_{t-1}, Y_{t-1})Y_{t-1}^\top)\|_F^2}$
  Compute $X_t = X_{t-1} - \eta_X \cdot \nabla_X L_\Omega(X_{t-1}, Y_{t-1})$
  ▷ Fix X and update Y.
  Compute $\eta_Y = \dfrac{\|\nabla_Y L_\Omega(X_t, Y_{t-1})\|_F^2}{\|P_\Omega(X_t[\nabla_Y L_\Omega(X_t, Y_{t-1})]^\top)\|_F^2}$
  Compute $Y_t = Y_{t-1} - \eta_Y \cdot \nabla_Y L_\Omega(X_t, Y_{t-1})$
  if stopping conditions are reached **then**
    break
  end if
end for

3.5.1 Mini-batch ASD 1

ASD still has a high computational per-iteration cost since the gradient must be computed over the whole data set in each iteration, similar to standard

gradient descent. However, similar to how gradient descent can benefit from mini-batching, so too can ASD. We propose a variant of ASD that utilises mini-batching called *mini-batch ASD 1*, which incorporates a standard mini-batching scheme into ASD. As outlined in Algorithm 5, the gradients and learning rates in each iteration are computed over a mini-batch of training data instead of the whole data set. This would reduce the computational per-iteration cost while retaining the core structure of the ASD algorithm.

Algorithm 5 Mini-batch ASD 1

Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: number of mini-batches B
Require: training mini-batches $\Omega_1, \ldots, \Omega_B$
Require: test set Ω_{test}
Require: maximum number of iterations $T \in \mathbb{N}$
$b \leftarrow 1$
for $t = 1$ **to** T **do**
    ▷ Fix Y and update X using mini-batch b.
    Compute $\eta_X = \dfrac{\|\nabla_X L_{\Omega_b}(X_{t-1}, Y_{t-1})\|_F^2}{\|P_{\Omega_b}(\nabla_X L_{\Omega_b}(X_{t-1}, Y_{t-1}) Y_{t-1}^\top)\|_F^2}$
    Compute $X_t = X_{t-1} - \eta_X \cdot \nabla_X L_{\Omega_b}(X_{t-1}, Y_{t-1})$
    ▷ Fix X and update Y using mini-batch b.
    Compute $\eta_Y = \dfrac{\|\nabla_Y L_{\Omega_b}(X_t, Y_{t-1})\|_F^2}{\|P_{\Omega_b}(X_t[\nabla_Y L_{\Omega_b}(X_t, Y_{t-1})]^\top)\|_F^2}$
    Compute $Y_t = Y_{t-1} - \eta_Y \cdot \nabla_Y L_{\Omega_b}(X_t, Y_{t-1})$
    if $L_{\Omega_{\text{test}}}(X_t, Y_t) < L_{\Omega_{\text{test}}}(X_{\text{best}}, Y_{\text{best}})$ **then**
        $X_{\text{best}} \leftarrow X_t$ ▷ Update the best iterates.
        $Y_{\text{best}} \leftarrow Y_t$
    end if
    if stopping conditions are reached **then**
        break
    end if
    Update b.
end for
return $X_{\text{best}} \approx X^*, Y_{\text{best}} \approx Y^*$

3.5.2 Mini-batch ASD 2

Unlike SGD and Adam, ASD does not update both factors X and Y simultaneously. Consequently, it is not restricted to using the same mini-batch

Algorithm 6 Mini-batch ASD 2

Require: initial parameters $X_0 \in \mathbb{R}^{n_1 \times r}, Y_0 \in \mathbb{R}^{n_2 \times r}$
Require: known entries $P_\Omega(A_0) \in \mathbb{R}^{n_1 \times n_2}$
Require: number of mini-batches B
Require: training mini-batches $\Omega_1, \ldots, \Omega_B$
Require: test set Ω_{test}
Require: maximum number of iterations $T \in \mathbb{N}$

$b_X \leftarrow 1$
$b_Y \leftarrow \frac{B}{2}$
for $t = 1$ **to** T **do**
    ▷ Fix Y and update X using mini-batch b_X.
    Compute $\eta_X = \dfrac{\|\nabla_X L_{\Omega_{b_X}}(X_{t-1}, Y_{t-1})\|_F^2}{\|P_{\Omega_{b_X}}(\nabla_X L_{\Omega_{b_X}}(X_{t-1}, Y_{t-1}) Y_{t-1}^\top)\|_F^2}$
    Compute $X_t = X_{t-1} - \eta_X \cdot \nabla_X L_{\Omega_{b_X}}(X_{t-1}, Y_{t-1})$
    ▷ Fix X and update Y using mini-batch b_Y.
    Compute $\eta_Y = \dfrac{\|\nabla_Y L_{\Omega_{b_Y}}(X_t, Y_{t-1})\|_F^2}{\|P_{\Omega_{b_Y}}(X_t[\nabla_Y L_{\Omega_{b_Y}}(X_t, Y_{t-1})]^\top)\|_F^2}$
    Compute $Y_t = Y_{t-1} - \eta_Y \cdot \nabla_Y L_{\Omega_{b_Y}}(X_t, Y_{t-1})$
    if $L_{\Omega_{\text{test}}}(X_t, Y_t) < L_{\Omega_{\text{test}}}(X_{\text{best}}, Y_{\text{best}})$ **then**
        $X_{\text{best}} \leftarrow X_t$ ▷ Update the best iterates.
        $Y_{\text{best}} \leftarrow Y_t$
    end if
    if stopping conditions are reached **then**
        break
    end if
    Update b_X and b_Y.
end for
return $X_{\text{best}} \approx X^*, Y_{\text{best}} \approx Y^*$

to update both factors within a given iteration. We leverage this flexibility to propose another variant of ASD with mini-batching, referred to as *mini-batch ASD 2*. As outlined in Algorithm 6, each factor is updated using a distinct mini-batch of the training data in the same iteration.

More specifically, let b_X and b_Y denote the mini-batch counter for X and Y, respectively. These are initialised as 1 and $\frac{B}{2}$, respectively, where B is the number of mini-batches. This offset between b_X and b_Y was chosen arbitrarily. In each iteration, the learning rate and gradient for X are computed using mini-batch Ω_{b_X}, while those for Y are computed using Ω_{b_Y}. b_X and b_Y are then updated as they are for SGD in Algorithm 1 and Adam in Algorithm 2, ensuring that $b_X \neq b_Y$ throughout the whole process. This second approach allows more training data to be utilised in each iteration, improving convergence as shown in Section 5.2.2.

3.6 Optimising the rank

All the algorithms discussed were originally designed to solve Equation (2.6), which omits the rank objective in Equation (2.1). Nevertheless, they can be adapted to address Equation (2.1) by progressively increasing the rank from 1 until an optimal solution is identified. This incremental-rank strategy was first introduced to improve the performance of PowerFactorisation. Although it requires multiple runs to identify the best rank, empirical results show that this approach consistently outperforms SDP-based algorithms for nuclear norm minimisation, both in terms of computational efficiency and recovery accuracy [12].

Inspired by this, we propose a similar rank-incrementing approach for SGD, Adam, and ASD. Starting from rank 1, the algorithms complete the matrix for each rank up to a maximum rank set arbitrarily. Unlike PowerFactorisation, which uses factors for the rank-1 completion as the initial factors for future completions, our implementations for SGD, Adam, and ASD initialise both factors randomly at each rank. This design choice, while arbitrary, simplifies implementation.

4 Continuous image representation model

In the traditional image representation model, an image is discretised into a uniform, rectangular spatial grid [15]. This representation allows the image to be modelled as a matrix, where the (i, j)-th entry denotes the intensity or energy level at the centre of the corresponding pixel. A key assumption

in this model is that the recorded intensity is accurately sampled at the exact centre of each pixel. However, this assumption often does not hold in practice. This inflates the rank of the matrix representation, thereby increasing the reconstruction error from a sampling of its entries.

To preserve the low-rank structure of the complete data set, we propose an alternative continuous model that extends the traditional discrete representation. Specifically, we assume that the image is generated by some unknown continuous function $f(x, y)$ which describes the intensity at any spatial location within the image domain. This function can be approximated using piecewise bilinear interpolation $\hat{f}(x, y)$, which enables the estimation of intensity values at arbitrary points aside from pixel centres. This interpolation-based model is particularly beneficial when the locations of the measurements are known to be off-centre since the actual intensity at the pixel centres can be approximated using this information. Consequently, the completed matrix reflects the underlying image more accurately, thereby maintaining its approximate low-rank structure and improving the performance of matrix completion algorithms.

4.0.1 One-dimensional piecewise linear interpolation

To formulate the continuous model, we first introduce one-dimensional piecewise linear interpolation. Suppose that a univariate function $f : \mathbb{R} \to \mathbb{R}$ is specified only by its values $a_1, a_2, \ldots, a_{n-1}, a_n$ at n distinct points (or *knots*) $x_1, x_2, \ldots, x_{n-1}, x_n$ [18]. Furthermore, suppose that $x_1 < x_2 < \ldots x_{n-1} < x_n$. The piecewise linear interpolant $\hat{f}$ of the function f is defined as

$$\hat{f}(x) = a_i + \frac{a_{i+1} - a_i}{x_{i+1} - x_i}(x - x_i), \quad x \in [x_i, x_{i+1}] \tag{4.1}$$

for $i = 1, \ldots, n-1$ [7]. Essentially, the data points $a_1, \ldots, a_n$ are connected by straight line segments, and the value of $f(x)$ for $x \in [x_1, x_n]$ can be approximated by the line segment between the two knots that contain x. Now, suppose that the knots are given by $x_i = i$ for $i = 1, \ldots, n$. Then, Equation (4.1) can be written as

$$\hat{f}(x) = a_i(1 + i - x) + a_{i+1}(x - i), \quad x \in (i, i + 1]. \tag{4.2}$$

For $i = 1, \ldots, n$, we define the **basis function** φ_i as

$$\varphi_i(x) = \begin{cases} x - i + 1 & , x \in [i - 1, i], \\ 1 + i - x & , x \in (i, i + 1], \\ 0 & , x \notin [i - 1, i + 1]. \end{cases}$$

Additionally, let

$$\mathbf{a} := \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix}, \quad \boldsymbol{\varphi}_n(x) := \begin{pmatrix} \varphi_1(x) \\ \varphi_2(x) \\ \vdots \\ \varphi_{n-1}(x) \\ \varphi_n(x) \end{pmatrix}.$$

Then, the knots $1, \ldots, n$ are the indexes of the values in $\mathbf{a}$, and Equation (4.2) can be rewritten as

$$\hat{f}(x) = \sum_{i=1}^{n} a_i \cdot \varphi_i(x) = \boldsymbol{\varphi}_n(x) \cdot \mathbf{a}, \tag{4.3}$$

where $(\cdot)$ denotes the vector dot product. Naturally, this representation of the interpolant $\hat{f}(x)$ as a vector dot product can be extended to the matrix-vector product to approximate multiple univariate functions simultaneously. The vector-valued piecewise linear interpolant $\hat{\mathbf{f}} : \mathbb{R} \to \mathbb{R}^r$ can be defined as

$$\hat{\mathbf{f}}(x) = \boldsymbol{\varphi}_n(x)^\top A, \tag{4.4}$$

where each column of $A \in \mathbb{R}^{n \times r}$ stores the known values of a different function at the knots $1, \ldots, n$.

4.0.2 Bilinear interpolation model

We now extend the concept of piecewise linear interpolation to a bivariate function $f(i, j)$ that is only specified for $(i, j) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\}$. Following a similar approach for approximating univariate functions, let $A \in \mathbb{R}^{n_1 \times n_2}$ contain the known values of the function f such that

$$A_{ij} = f(i, j), \quad (i, j) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\}. \tag{4.5}$$

The piecewise bilinear interpolant $\hat{f}$ can then be defined as

$$\hat{f}(x, y) = \boldsymbol{\varphi}_{n_1}(x)^\top A \, \boldsymbol{\varphi}_{n_2}(y). \tag{4.6}$$

Suppose that A has rank r. Using the decomposition $A = XY^\top$ from Equation (2.5), Equation (4.6) can be written as

$$\hat{f}(x,y) = \boldsymbol{\varphi}_{n_1}(x)^\top \left(XY^\top\right) \boldsymbol{\varphi}_{n_2}(y) \tag{4.7}$$

$$= \underbrace{\left(\boldsymbol{\varphi}_{n_1}(x)^\top X\right)}_{:=\hat{\mathbf{f}}_X(x)} \underbrace{\left(\boldsymbol{\varphi}_{n_2}(y)^\top Y\right)^\top}_{=:\hat{\mathbf{f}}_Y(y)} \tag{4.8}$$

where $\hat{\mathbf{f}}_X$ and $\hat{\mathbf{f}}_Y$ are the vector-valued piecewise linear interpolants of X and Y, respectively. Clearly, $\hat{f}(i',j') = A_{i'j'}$ for $i' = 1,\dots n_1$ and $j' = 1,\dots,n_2$ as $\varphi_i(i') = 1 = \varphi_j(j')$ if, and only if, $i' = i$ and $j' = j$; otherwise, $\varphi_i(i') = 0$ or $\varphi_j(j') = 0$.

Equation (2.6) can be modified to utilise this continuous model. Let $\Omega \subset [1,n_1] \times [1,n_2]$ be the set of positions for which the values of the bivariate function f are known. Instead of $L_\Omega(X,Y)$, we now aim to minimise the rescaled MSE between the interpolated values from the reconstruction and the known values. This new loss function can be expressed as

$$\tilde{L}_\Omega(X,Y) = \frac{1}{2} \sum_{(i,j)\in\Omega} \left(\hat{\mathbf{f}}_X(i) \cdot \hat{\mathbf{f}}_Y(j) - f(i,j)\right)^2. \tag{4.9}$$

5 Numerical experiments

The performance of the algorithms can be evaluated using several metrics. The original and reconstructed data sets were compared directly by computing the *average completion error*, often denoted e_c. This is the relative norm of the difference between the true matrix A and the output of the algorithm $A^* = X^*Y^*$, defined as

$$e_c = \frac{\|A - A^*\|_F}{\|A\|_F}. \tag{5.1}$$

Besides accuracy, the speed of the algorithms was evaluated by comparing the time taken to minimise the average completion error. All the algorithms tested were implemented in Python using the PyTorch library. Gradients were computed using the back-propagation algorithm implemented in PyTorch's automatic differentiation engine [19]. All the experiments were conducted on a computer with Intel Core i7-1065G7 CPUs @ 1.30GHz (1.50GHz boost) and 8GB of RAM, running Windows, and executed using Python 3.12.

5.1 Recovering synthetic data using standard matrix completion

The data set, labelled *SD1*, for this experiment was a rank-2 square matrix constructed as follows:

$$A_0 = XX^\top, \quad X \in \mathbb{R}^{n\times 2}, \quad X_{ij} = \begin{cases} \frac{i}{n} & , j = 1 \\ \left(1 - \frac{i}{n}\right)^2 & , j = 2 \end{cases} \tag{5.2}$$

For conciseness, this matrix was square as is typical in literature. Additionally, $n = 175$ was chosen to give this data set the same number of total entries as the real data set in Section 5.2. In the following experiments, all the algorithms aimed to reconstruct *SD1* as a rank-2 matrix. Unless stated otherwise, the undersampling ratio was fixed at $p = \frac{1}{3}$ in all the experiments.

5.1.1 Tuning the learning rate

The learning rates for SGD and Adam in all the experiments were tuned by comparing the number of iterations to achieve a minimal test loss within the desired precision. For SGD, learning rates of 1, 2, 4, ..., 128 were tested. Figure 1 shows the tuning results for SGD with 100 mini-batches on the data set *SD1*. Note that the loss function for SGD in all the experiments is the standard MSE, written as

$$\bar{L}_\Omega(X, Y) = \frac{1}{|\Omega|}\|P_\Omega(A_0) - P_\Omega(XY^\top)\|_F^2. \tag{5.3}$$

This was chosen over $L_\Omega(X, Y)$ in Equation (2.6) for consistency in tuning the learning rate for data sets of different sizes. For Adam, 1, 0.1, 0.01, and 0.001 were initially tested. Once an optimal learning rate from this set was found, finer tuning was performed by incrementing the learning within the same precision. For example, if the highest learning rate from the initial selection was 0.01, then the learning rate was increased to 0.02, 0.03, and so on until Adam no longer converged. Adam was observed to be more robust than SGD with respect to the choice of the learning rate. Across all experiments, Adam had an optimal learning rate of 0.02 up to 100 mini-batches, except without mini-batching in which case the optimal learning rate was 1. In contrast, the learning rate for SGD needed re-tuning, with the optimal learning rate varying between experiments and batch sizes.

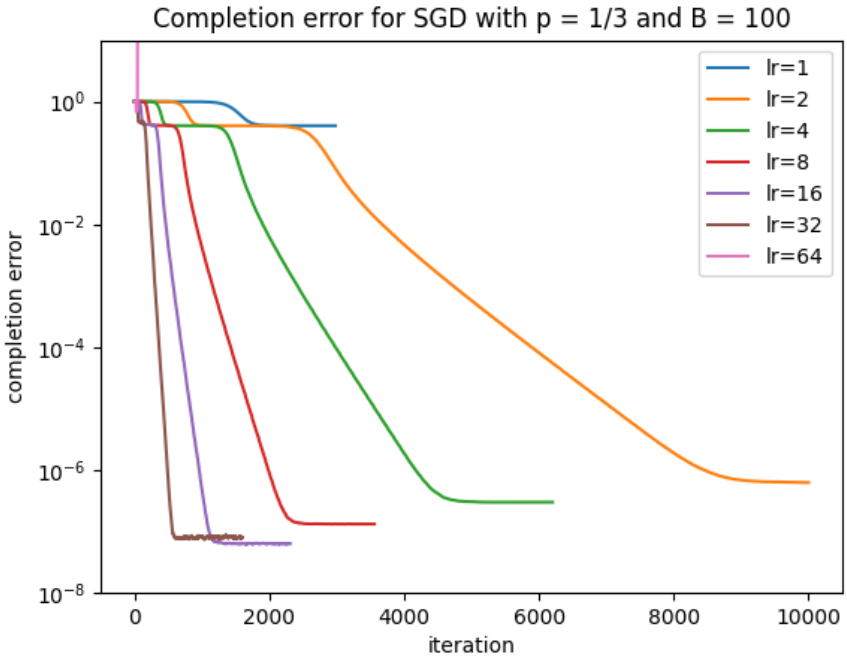

Figure 1: Plot of the mean completion error e_c on data set *SD1* for SGD with 100 mini-batches ($B = 100$) and various learning rates $\eta = 1, 2, 4, 8, 16, 32, 64$.

Similarly, the number of mini-batches was tuned in every experiment by testing all the algorithms with 1, 10, 20, 40, and 80 mini-batches. Each experiment was then conducted with the number of mini-batches that was optimal for all the algorithms.

5.1.2 Comparing the test loss and mean completion error

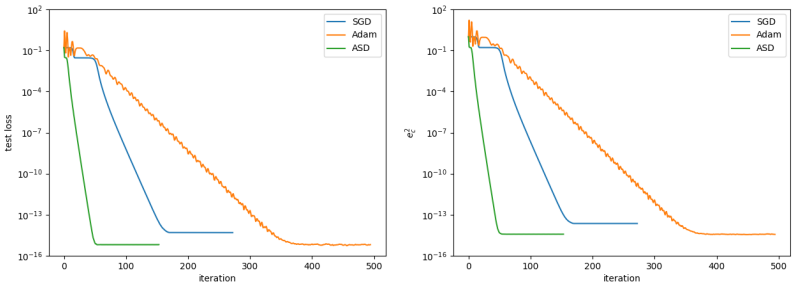

Figure 2: Plots of the test loss (left) and e_c^2 (right) against iteration for SGD, Adam, and ASD without mini-batching ($B = 1$) on data set *SD1*. The learning rates for SGD and Adam were 128 and 1, respectively.

In this experiment, the number of iterations required by each algorithm without mini-batching to converge to a minimum was compared. Figure 2 shows that ASD took the least iterations to converge, followed by SGD (equivalent to standard gradient descent), and lastly Adam. Notably, the plots of the test loss and squared mean completion error e_c^2 are visually identical, which shows that the test loss is a good indicator of the average completion error. Consequently, the test loss can be used to compare the speed of the algorithms since it avoids the high computational cost of calculating the mean completion error in every iteration.

5.2 Recovering real data using standard matrix completion

The data set for the following experiments was derived from the full data set *DS1* in [27]. The original data set is a three-dimensional tensor with $n_E = 149$ distinct energies, and spatial dimensions $n_1 = 101$ and $n_2 = 101$. It was reduced to a tensor of dimensions $n_E = 49$, $n_1 = 25$, and $n_2 = 25$ by partitioning it into equally sized chunks along each dimension, then computing the average energy level of each chunk. The resulting tensor was then flattened to obtain the full matrix A_0 of size 625×49. Unless stated

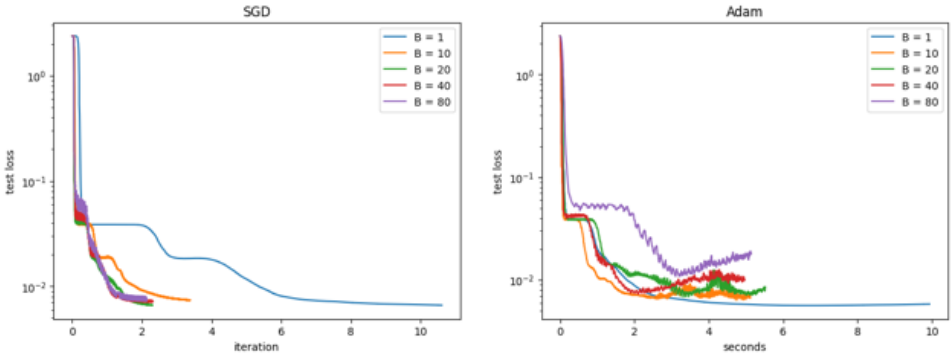

Figure 3: Plots of the test loss against time in seconds for SGD (left) and Adam (right) on data set *DS1* with various numbers of mini-batches $B = 1, 10, 20, 40, 80$. Adam and SGD had learning rates of 8 and 0.02, respectively, and run for 3000 iterations.

otherwise, all the algorithms aimed to reconstruct *DS1* as a rank-5 matrix in the following experiments.

5.2.1 Tuning the number of mini-batches

In this experiment, we investigated how mini-batching impacts the convergence and speed of the algorithms. Each algorithm was run for 3000 iterations for an easier comparison of their computing times. Figure 3 shows that SGD consistently converged to the same minimum test loss across all the values of B tested, whereas Adam began to deviate slightly for higher numbers of mini-batches. Additionally, there is not much improvement in the computational per-iteration cost beyond $B = 10$. This behaviour is expected since each mini-batch becomes increasingly smaller than the test set. As such, computing the test loss dominates the computational per-iteration cost for larger values of B. All the algorithms can be expected to perform much faster if the test loss was computed less frequently.

Figure 4 was included to compare the performance of all the algorithms for a given batch size. The time taken for each iteration of all the algorithms with 10 mini-batches is approximately half compared to without mini-batching ($B = 1$). Interestingly, SGD and Adam can achieve a lower test loss than standard ASD, and in less time, with mini-batching. While mini-batching reduces the overall computation time for ASD, it takes roughly the same time to converge to a minimal test loss.

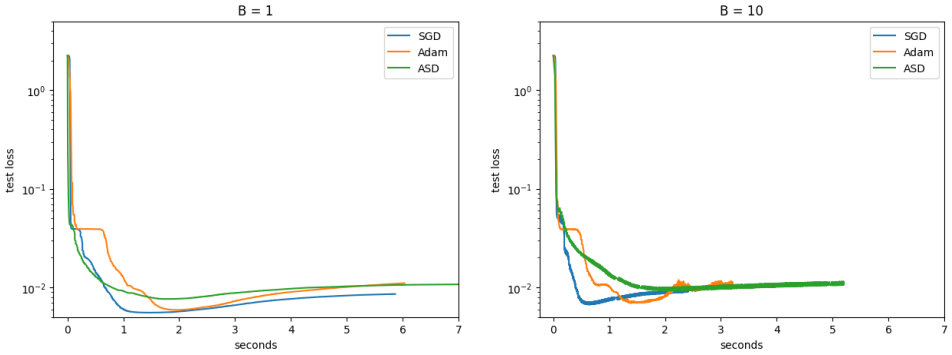

Figure 4: Plots of the test loss against iteration for SGD, Adam, and ASD on data set *DS1* for $B = 1$ (left) and $B = 10$ (right). Adam had a learning rate of 0.02 in both experiments. SGD had a learning rate of 32 for $B = 1$, and 8 for $B = 10$.

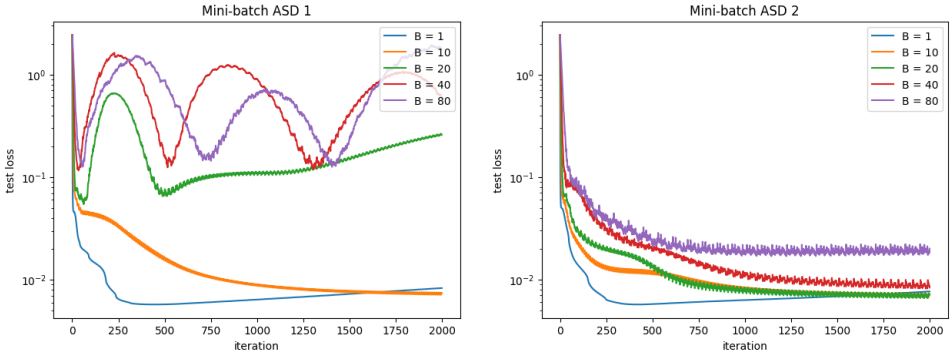

Figure 5: Plots of the test loss against iteration for mini-batch ASD 1 (left) and 2 (right) on data set *DS1* for various numbers of mini-batches $B = 1, 10, 20, 40, 80$.

5.2.2 Comparing mini-batch schemes for ASD

All the algorithms successfully recovered the synthetic data set *SD1* in Section 5.1 up to 80 mini-batches. However, Figure 5 shows that mini-batch ASD 1 is less consistent when applied to the data set *DS1*. While it converged close to the same minimum test loss as the standard ASD with less mini-batches, it fails to converge for $B \leq 20$. Note that mini-batch ASD 1 and 2 without mini-batching ($B = 1$) is standard ASD. In contrast, mini-batch ASD 2 converges to approximately the same minimal test loss across most values of B. This difference in performance is expected since each iteration in mini-batch ASD 2 uses more training examples per iteration than mini-batch ASD 1. For clarity, mini-batch ASD 1 will not be included in the following experiments.

5.2.3 Determining the optimal rank

The purpose of this experiment was to test the effectiveness of the methodology proposed in Section 3.6 for determining the optimal rank. Starting from rank 1, the factors X and Y were optimised for five random sampling sets Ω starting from different, random initialisations. The sample mean μ of the average completion error across these five experiments for each rank was computed and plotted in Figure 6. Additionally, the 95% asymptotic confidence interval was calculated and plotted in Figure 6. This confidence interval is given by

$$\mu \pm z_{0.975} \cdot \sqrt{\frac{\sigma^2}{5}},$$

where σ^2 is the sample variance, and $z_{0.975}$ is the 97.5^{th}-percentile of the standard normal distribution with mean 0 and variance 1 [21]. The red line at rank 15 in Figure 6 marks where the completion error is expected to start increasing. Beyond this rank, there would be more unknown rows in X and Y to optimise than there are known entries from the underlying matrix A_0. Based on this, the estimated critical rank is given by

$$\tilde{r} = \frac{n_1 n_2 p}{n_1 + n_2}.$$

Based on Figure 6, rank 5 appears to be the most optimal for all the algorithms since increasing the rank further does not lower average completion error. The error remains approximately constant for all the algorithms as the

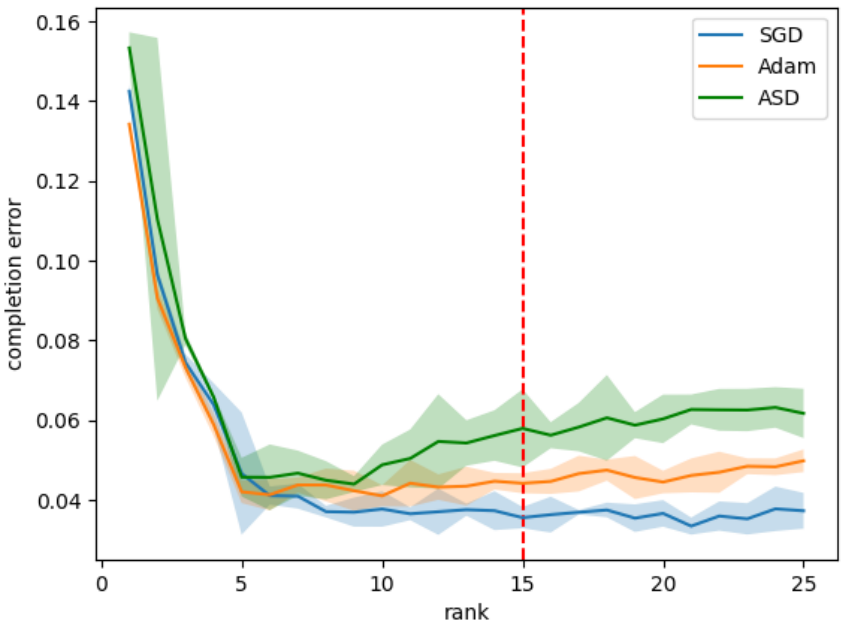

Figure 6: Plot of the mean completion error and the 95% asymptotic confidence interval as a function of the rank of SGD, Adam, and mini-batch ASD 2 on data set *DS1*. SGD and Adam had learning rates of 8 and 0.02, respectively. All algorithms were run with 10 mini-batches ($B = 10$).

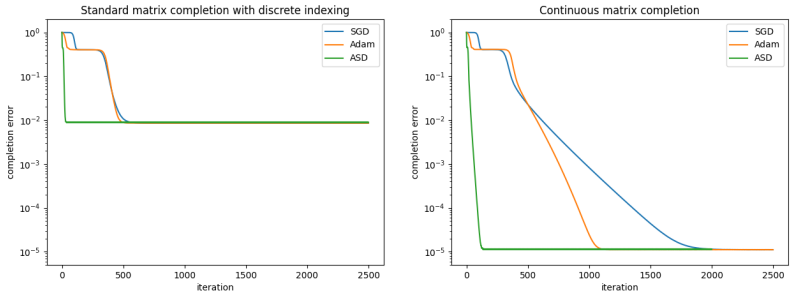

Figure 7: Plots of the mean completion error for standard matrix completion (left) and continuous matrix completion (right) with SGD, Adam, and mini-batch ASD 2 on data set *SD2*. All the algorithms were run with 10 mini-batches ($B = 10$). SGD and Adam had learning rates 16 and 0.01, respectively.

rank increases further above 5, with the error for ASD increasing slightly. Furthermore, SGD achieves the lowest completion error when the rank is overestimated, followed by Adam, and ASD.

5.3 Recovering synthetic data using continuous matrix completion

In this experiment, the data set used, labelled *SD2*, was constructed as follows:

$$(A_0)_{ij} = \left(\frac{i+\varepsilon_1}{n}\right)\left(\frac{j+\varepsilon_2}{n}\right) + \left(1-\frac{i+\varepsilon_1}{n}\right)^2\left(1-\frac{j+\varepsilon_2}{n}\right)^2,$$

where $\varepsilon_1, \varepsilon_2$ are i.i.d. Unif$[0,1]$ random variables. *SD1* and *SD2* are images of the same surface, except that the entries in *SD2* do not correspond to their positions in the actual surface. As such, this data set is only approximately rank-2 because of the noisy positions. All the algorithms were given rank 2 in this experiment.

Figure 7 shows that similar to the real data set *DS1*, the completion error cannot be minimised further because the data set is not exactly low-rank. However, similar results to the experiments in Section 5.1 can be achieved with continuous matrix completion. The minimal completion error observed in this experiment is higher than that in Section 5.1, which is expected since the continuous image model is an approximation of the actual surface.

The key finding is that when the exact spatial coordinates of the known entries are available, continuous matrix completion can yield a more accurate reconstruction of the original matrix.

6 Conclusion and future directions

We have proposed two variants of ASD with mini-batching for matrix completion. Empirical evidence shows that mini-batching can improve the computational time of ASD. However, the mini-batching scheme can significantly impact the stability of convergence for smaller mini-batches. Other mini-batching schemes could be considered, such as computing each learning rate and gradient using different mini-batches. This could lead to even faster performance by utilising four mini-batches in each iteration. Alternatively, the gradients for each factor could be computed using one-mini-batch, and the learning rates with another mini-batch within the same iteration.

Empirical evidence indicates that SGD and Adam are competitive with ASD for matrix completion in terms of the recoverable rank and overall computational time. Furthermore, all these algorithms can be used to efficiently estimate the true rank of the underlying matrix by gradually incrementing the rank from 1. Future studies could examine the performance of these algorithms under smaller undersampling ratios, as empirical evidence suggests that ASD can still reconstruct matrices satisfactorily for undersampling ratios as low as $p = 0.15$ [27]. Additionally, it may be of interest to test whether Adam and SGD would benefit from the LoopedASD methodology, where the output from the previous completion is taken as the input for the current completion with an incremented rank.

The continuous matrix completion model was demonstrated to enable a more accurate reconstruction of the underlying matrix if the exact positions of the entries are known. However, this model still relies on the positions being measured and entered manually. This model could be improved by incorporating image registration techniques or learning the shifts in data positions by measuring factors such as changes in the ambient temperature, scanning time, and x-ray intensity.

References

[1] Y. Amit, M. Fink, N. Srebro, and S. Ullman. Uncovering shared structures in multiclass classification. In *Proceedings of the 24th international*

conference on Machine learning, pages 17–24, 2007.

[2] A. Argyriou, T. Evgeniou, and M. Pontil. Multi-task feature learning. *Advances in neural information processing systems*, 19, 2006.

[3] J. Bennett, C. Elkan, B. Liu, P. Smyth, and D. Tikk. Kdd cup and workshop 2007. *ACM SIGKDD explorations newsletter*, 9(2):51–52, 2007.

[4] L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

[5] J. F. Cai, E. J. Candes Candes, and Z. Shen. A singular value thresholding algorithm for matrix completion, 2008.

[6] E.J. Candès and B. Recht. Exact matrix completion via convex optimization. *Found Comput Math*, 9:717–772, 2009.

[7] B. Daasagupta. *Applied mathematical methods*. Always Learning. Pearson, 1st edition edition, 2006.

[8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. volume 12, pages 257–269, 01 2010.

[9] L. Eldén. *Matrix methods in data mining and pattern recognition*. SIAM, 2019.

[10] R. Escalante and M. Raydan. *Alternating projection methods*. SIAM, 2011.

[11] R. M. Freund. Introduction to semidefinite programming (sdp). *Massachusetts Institute of Technology*, pages 8–11, 2004.

[12] J.P. Haldar and D. Hernando. Rank-constrained solutions to linear matrix equations using powerfactorization. *IEEE Signal Processing Letters*, 16(7):584–587, 2009.

[13] N. J. A. Harvey, D. R. Karger, and S. Yekhanin. The complexity of matrix completion. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, page 1103–1111, USA, 2006. Society for Industrial and Applied Mathematics.

[14] S.C.H. Hoi, D. Sahoo, J. Lu, and P. Zhao. Online learning: a comprehensive survey. *Neurocomputing*, 459:249–289, 2021.

[15] V.M. Jimenex-Fernandex, H. Vazquez-Leal, U.A. Filobello-Nino, M. Jimenez-Fernandex, L.J. Morales-Mendoza, and M. Gonzalez-Lee. Exploring the use of two-dimensional piecewise-linear functions as an alternative model for representing and processing grayscale-images. *Journal of Applied Research and Technology*, 14(5):311–318, 2016.

[16] Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization. Technical report, 2014.

[17] Z. Liu and L. Vandenberghe. Interior-point method for nuclear norm approximation with application to system identification. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1235–1256, 2010.

[18] J. Maidens. An illustrated guide to interpolation methods, 08 2016.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[20] Jasson DM Rennie and Nathan Srebro. Fast maximum margin matrix factorization for collaborative prediction. In *Proceedings of the 22nd international conference on Machine learning*, pages 713–719, 2005.

[21] J.A. Rice. *Mathematical statistics and data analysis*. Duxbury advanced series. Thomson, Belmont, Calif, 3rd ed. edition, 2007.

[22] S. Ruder. An overview of gradient descent optimization algorithms, 2017.

[23] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011.

[24] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: from theory to algorithms*. Cambridge University Press, USA, 2014.

[25] J. Tanner and K. Wei. Normalized iterative hard thresholding for matrix completion. *SIAM Journal on Scientific Computing*, 35(5):S104–S125, 2013.

[26] J. Tanner and K. Wei. Low rank matrix completion by alternating steepest descent methods. *Applied and Computational Harmonic Analysis*, 40(2):417–429, 2016.

[27] O. Townsend, S. Gazzola, S. Dolgov, and P. Quinn. Undersampling raster scans in spectromicroscopy for reduced dose and faster measurements. *Optics Express*, 30(24):43237–43254, 2022.

[28] R. Tutuncu, K. Toh, and M. Todd. Sdpt3 - a matlab software package for semidefinite-quadratic-linear programming, version 3.0. Technical report, National University of Singapore, 08 2001.

[29] B. Vandereycken. Low-rank matrix completion by riemannian optimization. *SIAM Journal on Optimization*, 23(2):1214–1236, 2013.

[30] V. Vasudevan and M. Ramakrishna. A hierarchical singular value decomposition algorithm for low rank matrices. *arXiv preprint arXiv:1710.02812*, 2017.

[31] B. Zhou, C. Han, and T. Guo. Convergence of stochastic gradient descent in deep neural network. *Acta Mathematicae Applicatae Sinica, English Series*, 37(1):126–136, 2021.