Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Klara Schlüter and Jon Riege

# Stochastic Multiplicative Updates for Symmetric Nonnegative Matrix Factorization

Master's thesis in Informatics and Computer Science
Supervisor: Professor Zhirong Yang
May 2021

NTNU
Norwegian University of
Science and Technology

Klara Schlüter and Jon Riege

# Stochastic Multiplicative Updates for Symmetric Nonnegative Matrix Factorization

Master's thesis in Informatics and Computer Science
Supervisor: Professor Zhirong Yang
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

# Abstract

In the field of machine learning, symmetric nonnegative matrix factorization (NMF) is commonly used for clustering. NMF is often performed by multiplicative update (MU) algorithms, because they naturally maintain nonnegativity. Inspired by the success of stochasticity in gradient descent, we develop a novel stochastic MU algorithm for symmetric NMF. We call the algorithm Stochastic Bound-and-Scale Multiplicative Updates (SBSMU). To the best of our knowledge, this is the first time stochasticity has been introduced to MU for symmetric NMF. We provide a theoretical analysis of SBSMU, including a proof which provides insights into the conditions under which the algorithm converges. Furthermore, we present the results of three empirical experiments. The data suggests that a standard configuration of SBSMU achieves a relatively good performance across datasets. Moreover, we find that that SBSMU is able to factorize the large datasets we apply it to, although it is not on par with the best benchmarks.

# Sammendrag

Innen maskinlæring brukes symmetrisk ikkenegativ matrisefaktorisering (NMF) i forbindelse med clusteranalyse. NMF utføres ofte ved hjelp av algoritmer som tar i bruk multiplikative oppdateringer (MU). Disse har den fordelen at de automatisk opprettholder ikkenegativitet i faktormatrisen. Inspirert av suksessen til stokastisk gradient descent, utvikler vi en ny stokastisk MU-algoritme. Vi kaller algoritmen Stochastic Bound-and-Scale Multiplicative Updates (SBSMU). Så langt vi vet, er dette den første gangen en stokastisk MU-algoritme har blitt utviklet for symmetrisk NMF. Vi presenterer en teoretisk analyse av SBSMU, inkludert et bevis som gir innsikt i betingelsene som avgjør om SBSMU konvergerer. Videre presenterer vi resultatene fra tre empiriske eksperimenter. Dataene tilsier at en standardkonfigurasjon for SBSMU gir relativt god ytelse på tvers av datasett. I tillegg viser vi at SBSMU kan faktorisere store datasett vi tester den på, selv om algoritmen ikke når opp til de beste referansealgoritmene.

# Preface

This thesis is a part of our master's degrees at the Norwegian University of Science and Technology. Klara's degree is a Master of Science in Informatics, while Jon's degree is a Master of Technology in Computer Science. Our work has been conducted at the Artificial Intelligence Group in the Department of Computer and Information Science. The thesis is the result of a year-long collaboration project which started in the fall of 2020. We have worked together on both the research and the writing, and have contributed equally to the thesis.

We would like to thank our supervisor, Professor Zhirong Yang, for providing the initial idea and for invaluable support throughout the project.

Klara Schlüter and Jon Riege
Trondheim, May 31, 2021

# Contents

# Abbreviations

| | |
|---|---|
| ASAG-MU | Asymmetric Stochastic Average Gradient MU |
| ASG-MU | Asymmetric Stochastic Gradient MU |
| EGD | Exponentiated Gradient Descent |
| GSAG-MU | Greedy Stochastic Average Gradient MU |
| GSG-MU | Greedy Stochastic Gradient MU |
| MM | Majorize-Minimize |
| MU | Multiplicative Updates |
| NMF | Nonnegative Matrix Factorization |
| PSGD | Projected Stochastic Gradient Descent |
| RQ | Research Question |
| SAGMU | Stochastic Average Gradient MU |
| SBSMU | Stochastic Bound-and-Scale MU |
| SGD | Stochastic Gradient Descent |
| SVRMU | Stochastic Variance Reduced MU |

# 1 Introduction

Nonnegative matrix factorization (NMF) is the decomposition of a nonnegative matrix into a set of nonnegative factor matrices. In symmetric NMF, the purpose is to find a matrix factor $\boldsymbol{W}$ so that $\boldsymbol{W}\boldsymbol{W}^\top \approx \boldsymbol{X}$. This can be used for cluster analysis, which has applications in many different fields.

Multiplicative updates (MU) are a commonly used class of algorithms for performing NMF. These algorithms perform multiplicative update steps, as opposed to the additive update steps used in gradient descent. This has the key advantage that it naturally maintains nonnegativity.

Stochastic gradient descent (SGD) is a variant of gradient descent that uses a randomly selected subset of the data in each iteration, thus reducing computational cost. SGD and its various extensions are the leading optimization algorithms in machine learning, illustrating how successful the introduction of stochasticity has been for gradient descent.

There exists some research into stochastic MU, though only for linear NMF [43, 25, 24]. Nevertheless, these papers suggest that this approach holds potential. In addition to reducing the computational cost per iteration, stochastic MU makes parallelization possible, which can further improve performance on systems with multiple processors or cores.

We are motivated by this to investigate how stochasticity can be applied to MU for symmetric NMF, and whether it leads to improvement in performance. We choose to focus on symmetric NMF because it is an important variant of NMF where stochasticity has not been introduced previously. Furthermore, symmetric NMF poses some unique challenges and opportunities compared with linear NMF. The symmetric matrices are usually sparse, which means computation time can be significantly reduced by introducing algorithmic optimizations that handle zero-entries well. This is particularly important for very large datasets, where factorization may be computationally infeasible otherwise. However, these zero-entries also create challenges in the stochastic case. They produce multiplicative updates that are themselves zero, and this must be accounted for to avoid a factor matrix of all zeroes.

The remainder of this section is structured as follows: in Section 1.1, we define our goals and present specific research questions (RQ). Section 1.2 contains the research methods used to investigate these questions. After summarizing our contributions in Section 1.3, we present the structure of the thesis in Section 1.4.

## 1.1 Goals and Research Questions

Our overarching goal is to improve the performance of MU for symmetric NMF by introducing stochasticity. Our hypothesis is that it is possible to develop a converging algorithm for stochastic MU, which can be parallelized to achieve a faster rate of convergence. Our goal raises the question of what is meant by the term *performance*. First and foremost it is necessary to define the metrics that measure the quality of the factorization, since a perfect factorization where $\boldsymbol{W}\boldsymbol{W}^\top = \boldsymbol{X}$ is impossible to obtain for most datasets.

There are multiple candidates for quantifying factorization quality. Multiplicative update rules are derived based on some loss function that quantifies the difference between $\boldsymbol{W}\boldsymbol{W}^\top$ and $\boldsymbol{X}$. Thus, the loss function is a logical candidate when considering possible metrics. However, in applications the typical use case for symmetric NMF is cluster analysis. The quality of a particular clustering does not depend directly on the calculated loss, but on the extent to which the clusters reflect meaningful groups in the data. If the dataset is labeled with the true cluster identities, the quality of the clustering can then be measured using metrics like homogeneity and completeness. We choose to measure the quality of a clustering using the loss function rather than other metrics. This is because our focus is on MU as an optimization algorithm and not on any particular application. One advantage of this is that we do not require the datasets we use for experiments to be labeled.

Given that we use the loss function to determine factorization quality, there are still multiple ways to measure performance. For example, we may consider the minimum loss, or the time taken until a specific loss is reached. Since multiplicative algorithms depend on hyperparameters and randomized initialization, we may also consider the variance of an algorithm.

What measure of performance is the most important depends on the problem at hand. When the total running time is low, the minimum loss might be the only thing that really matters, since it is not problematic to run the factorization multiple times. However, for extremely large problems it may be that stability and a high rate of convergence is critical. Since we are not concerned with one specific use case, we thus need to consider multiple criteria.

Our goal and hypothesis, as well as our reasoning around how to best measure performance, lead us to the following research questions:

- **RQ1:** How can stochasticity be introduced to MU for symmetric NMF?
  - **RQ1.1:** Can stochastic MU be proven to converge?
  - **RQ1.2:** Can stochastic MU be parallelized?
- **RQ2:** How does stochastic MU for symmetric NMF perform?

- **RQ2.1:** How robust is stochastic MU to different datasets, hyperparameter configurations and initializations?
- **RQ2.2:** How does stochastic MU perform in terms of minimum loss?
- **RQ2.3:** How does stochastic MU perform in terms of convergence time?

## 1.2 Research Methods

In order to answer the research questions, our first step was to conduct a literature review on stochastic MU. We searched for relevant works based on predefined keywords and scope and filtered the results according to certain quality criteria. In addition, we included some foundational papers on MU. We then reviewed the resulting papers with a focus on features that were relevant for our work.

To address RQ1 we then developed a novel stochastic MU algorithm for symmetric NMF. We used full-batch MU as a basis and introduced stochasticity, loosely following the example of SGD. By analyzing potential issues and possible solutions, we developed a converging stochastic algorithm, Stochastic Bound-and-Scale Multiplicative Updates (SBSMU). We then optimized the algorithm and its implementation. To answer RQ1.2, we parallelized the factorization using the Hogwild framework [41].

We evaluated SBSMU through both theoretical analysis and empirical experiments. Theoretically, we investigated under what assumptions the algorithm can be proven to converge in order to answer RQ1.1. By applying existing frameworks, we derived a proof of convergence in expectation. Again, we used techniques from SGD to model the stochastic behavior. To test the performance of SBSMU and answer RQ2, we applied our implementation to 10 real-world datasets in three experiments. We focused on sparse datasets factorized using the I-divergence, as this is the most common use case for symmetric NMF. We empirically investigated how sensitive SBSMU is to its hyperparameters and how it performs on medium-sized and large datasets compared to three benchmark algorithms. To facilitate reproducibility, we present in detail how the experiments were run, including hardware environments, source code and datasets.

## 1.3 Contributions

Here we summarize the key contributions of this work. We divide our contributions into three categories: the SBSMU algorithm, the proof of convergence and the experimental results.

Our most important contribution is the novel SBSMU algorithm. To the best of our knowledge, it is the first stochastic algorithm to be devised for MU

for symmetric NMF. To achieve convergence, SBSMU utilizes a novel variance reduction technique which we call bound-and-scale. Beyond the algorithm itself, we also present several techniques for improving the efficiency of implementations of SBSMU. We provide three different algorithmic optimizations, which reduce execution time without affecting the logic of the pseudocode. Furthermore, we show how a modified version of stratified sampling can be applied to increase the sampling rate of nonzero entries in sparse matrices. Finally, we show how SBSMU can be parallelized without the use of object locks.

Another key contribution is a proof of convergence for SBSMU. The bound-and-scale step in SBSMU is controlled by the parameter $\alpha$. Our proof shows that SBSMU will converge as long as the bound-and-scale parameter $\alpha$ is sufficiently close to 1. The minimum value of $\alpha$ depends on an convergence condition, which we analyze as a part of our remarks on the proof.

Our final contributions are the results from a set of empirical experiments on real world datasets. We show data that suggests SBSMU is robust to the random factors in the algorithm, and relatively robust to hyperparameters and datasets for high values of $\alpha$. Furthermore, we demonstrate that SBSMU is able to factorize two large datasets, MNIST and Higgs. However, SBSMU is outperformed by full-batch MU, both in terms of minimum loss and convergence time.

## 1.4 Thesis Structure

Here we present how the remainder of the thesis is structured. Section 2 contains background knowledge and theory that is necessary to understand our work and put it into context. It includes foundational mathematical concepts, real-world applications of symmetric NMF and a summary of our literature review of the field. This section also introduces much of the mathematical notation we use throughout our thesis. For an overview of notation, see Appendix A. Section 3 introduces SBSMU, our theoretical analysis with the proof of convergence, and how we have optimized and parallelized our implementation. Section 4 presents the results of the empirical experiments, as well as the setup and datasets we used. Next, in Section 5 the research questions are discussed in light of the theoretical analysis and experimental results. Finally, Section 6 contains the evaluation. This includes our conclusions, a discussion of the limitations of our work and ideas on where future research may be directed.

# 2 Background

This chapter provides the background information and theory that our work is based on. Section 2.1 presents mathematical concepts that are central to our theoretical work. Section 2.2 contains an overview of the applications of symmetric NMF, including how it is closely related to kernel $k$-means. Finally, Section 2.3 presents our literature review protocol and the results of our literature search. The most relevant papers are presented in detail.

## 2.1 Mathematical Concepts

This section presents a selection of mathematical concepts that are key in understanding this work. The first sections introduce the concepts we build our algorithm on: NMF in Section 2.1.1, loss functions in Section 2.1.2 and MU in Section 2.1.3. The remaining subsections contain tools for proving the convergence of MU algorithms. The MM algorithm is introduced in Section 2.1.4, followed by Taylor approximations in Section 2.1.5 and Jensen's inequality in Section 2.1.6. The latter two are often used in conjunction with the MM algorithm.

### 2.1.1 Nonnegative Matrix Factorization

Matrix factorization is the process of finding a set of factor matrices whose matrix product is approximately equal to the original matrix. Nonnegative matrix factorization adds the constraint that each element in the original matrix and the factor matrices must be nonnegative. In this work we discuss two forms of NMF, linear NMF and symmetric NMF.[1]

**Definition 2.1.1** (Linear NMF)**.** Denote the original matrix $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{n \times m}$, and factors $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{n \times r}$ and $\boldsymbol{H} \in \mathbb{R}_{\geq 0}^{r \times m}$. Find $\boldsymbol{W}$ and $\boldsymbol{H}$ so that

$$\boldsymbol{W}\boldsymbol{H} \approx \boldsymbol{X}. \tag{2.1}$$

**Definition 2.1.2** (Symmetric NMF)**.** Denote the original matrix $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{n \times n}$ and the factor $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{n \times r}$. $\boldsymbol{X}$ is symmetric, i.e. $\boldsymbol{X} = \boldsymbol{X}^\top$. Find $\boldsymbol{W}$ so that

$$\boldsymbol{W}\boldsymbol{W}^\top \approx \boldsymbol{X}. \tag{2.2}$$

[1] For a comprehensive introduction to NMF, see the textbook by Cichocki et al. [8].

The rank $r$ of the factorization determines the representative capacity of the factors — the complexity of the relationships they are able to model. Let $i \in \{1 \dots n\}$, $j \in \{1 \dots m\}$ and $k \in \{1 \dots r\}$. Intuitively, each entry in the linear approximation $\boldsymbol{WH}$ is calculated as

$$(\boldsymbol{WH})_{ij} = \sum_k W_{ik} H_{kj} \tag{2.3}$$

and increasing $r$ thus increases the number of terms that are used to approximate each entry of $\boldsymbol{X}$. The same holds for symmetric NMF, where $i, j \in \{1 \dots n\}$ and

$$(\boldsymbol{WW}^\top)_{ij} = \sum_k W_{ik} W_{jk}. \tag{2.4}$$

In practical applications, it is usually the case that $r << \min\{n, m\}$.

For both linear and symmetric NMF, it is often not possible to find factors that exactly reproduce $\boldsymbol{X}$. It is useful to be able to quantify the quality of an approximation, which can be accomplished using a loss function.

### 2.1.2 Loss Functions

Loss functions map a variable to a scalar cost, which the goal is to minimize. In matrix factorization, the loss function compares an approximation with the ground truth and returns the error. More specifically, in symmetric NMF the loss function $L : \mathbb{R}_{\geq 0}^{n \times r} \to \mathbb{R}_{\geq 0}$ is of the form

$$L(\boldsymbol{W}) = D(\boldsymbol{X} \| \boldsymbol{WW}^\top), \tag{2.5}$$

and compares the ground truth $\boldsymbol{X}$ with its approximation $\widehat{\boldsymbol{X}} = \boldsymbol{WW}^\top$.

Different loss functions emphasize different properties of the approximation and are thus applied in different use cases. In this work, we are especially interested in the Euclidean distance

$$L_{EU}(\boldsymbol{W}) = \sum_{ij} \left[ X_{ij} - \widehat{X}_{ij} \right]^2,$$

which is commonly used for factorizing dense matrices, and the I-divergence

$$L_I(\boldsymbol{W}) = \sum_{ij} \left[ X_{ij} \ \ln \left( \frac{X_{ij}}{\widehat{X}_{ij}} \right) - X_{ij} + \widehat{X}_{ij} \right],$$

which is especially useful for factorizing sparse matrices [52].

### 2.1.3 Multiplicative Update Algorithms

Multiplicative update algorithms are widely used for NMF. As the name suggests, MU algorithms iteratively apply multiplicative updates to the factor matrices. The concrete updates depend on the loss function that is minimized. The updates are based on the gradient of the loss and are of the form

$$W_{ik} \leftarrow W_{ik} \cdot \left[\frac{\nabla^-_{ik}}{\nabla^+_{ik}}\right]^\eta . \tag{2.6}$$

Here, $0 < \eta \leq 1$ is an exponential learning rate. $\nabla^+_{ik}$ and $\nabla^-_{ik}$ are the unsigned sums of the positive and negative terms of the partial derivative of the loss function, such that

$$\frac{\partial L(\boldsymbol{W})}{\partial W_{ik}} = \nabla^+_{ik} - \nabla^-_{ik}. \tag{2.7}$$

As opposed to SGD, where updates are applied additively, multiplicative updates naturally maintain nonnegativity. If the factor $\boldsymbol{W}$ is initialized to be nonnegative, the updates and in turn the updated $\boldsymbol{W}$ will remain nonnegative.

### 2.1.4 MM Algorithm

The MM algorithm is not strictly an algorithm, but a method for creating iterative optimization algorithms [39, 23] and proving their convergence. The central idea is to define an auxiliary function to derive the updates. If the goal is the minimization of a loss function $f$, MM stands for *majorize-minimize*[2].

The auxiliary function $g$ majorizes $f$ and is itself minimized by the update of the model parameters at each iteration. Majorization requires that for any $a$ and $b$ in the domain of $f$,

$$g(a, a) = f(a) \tag{P1}$$
$$g(b, a) \geq f(b). \tag{P2}$$

Let $x$ represent the current value of the model parameters and $\widetilde{x}$ the model parameters as a variable. In each iteration, the next value $x^{\text{new}}$ is found by minimizing $g$ so that

$$x^{\text{new}} = \arg\min_{\widetilde{x}} g(\widetilde{x}, x), \tag{2.8}$$

which ensures

$$g(x, x) \geq g(x^{\text{new}}, x). \tag{P3}$$

[2]When the goal is maximization, MM analogously stands for *minorize-maximize*.

This in turn ensures that the optimization algorithm is monotonically decreasing, because

$$f(x) \overset{(P1)}{=} g(x,x) \overset{(P3)}{\geq} g(x^{\text{new}},x) \overset{(P2)}{\geq} f(x^{\text{new}}) \tag{2.9}$$

To prove convergence, all three properties (P1), (P2) and (P3) have to be shown.

The challenge in constructing an MM algorithm is to find an auxiliary function that both majorizes $f$ and itself is easy to minimize. Two important tools in this regard are Taylor's theorem and Jensen's inequality, which are presented in the next two subsections. Furthermore, Section 2.3.3 contains an example of how they can be used to majorize concave and convex functions.

### 2.1.5 Taylor's Theorem

Taylor's theorem allows us to create polynomials that approximate differentiable functions.

**Definition 2.1.3** (Taylor Polynomial)**.** Let $f : \mathbb{R} \to \mathbb{R}$ be $k$ times differentiable at $a \in \mathbb{R}$. The $k$th Taylor polynomial of $f$ near $a$ is defined as

$$P_k(x) = \sum_{i=0}^{k} \frac{(x-a)^i}{i!} \frac{d^i f(a)}{dx^i}. \tag{2.10}$$

The Taylor polynomial is an approximation of $f$, where $P_k(x) \approx f(x)$ for values near $a$, and $P_k(a) = f(a)$ [46].

If $f$ is concave, the first order Taylor polynomial upper bounds it [51],

$$f(x) \leq f(a) + \frac{df(a)}{dx}(x-a) = P_1(x). \tag{2.11}$$

This bound can be visualized geometrically by considering that $P_1(x)$ is a tangent to $f(a)$, as shown in Figure 2.1.

Taylor polynomials can be generalized to multiple dimensions. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a concave function, then

$$f(\boldsymbol{x}) \leq f(\boldsymbol{a}) + (\boldsymbol{x}-\boldsymbol{a})\nabla f(\boldsymbol{a}) \tag{2.12}$$

$$= f(\boldsymbol{a}) + \sum_i (x_i - a_i)\frac{\partial f(\boldsymbol{a})}{\partial x_i}. \tag{2.13}$$

### 2.1.6 Jensen's Inequality

Jensen's inequality is a useful tool for upper bounding convex functions. Geometrically, convexity means that the graph of $f$ lies below the weighted arithmetic

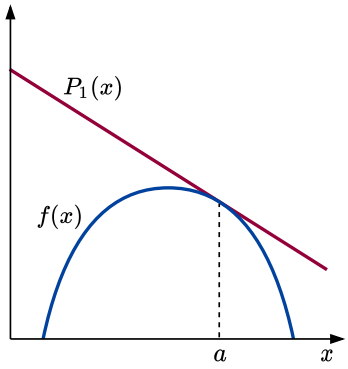



Figure 2.1: Illustration of first order Taylor polynomial upper bounding a concave function

mean of any two of its points. The idea of Jensen's inequality is that this definition can be extended to any number of points $n$ and their weighted arithmetic mean [37].

**Theorem 2.1.1** (Jensen's Inequality)**.** Let $f : \mathbb{R} - > \mathbb{R}$ be a convex function. For $x_1, \ldots, x_n \in \mathbb{R}$ and $\lambda_1, \ldots \lambda_n \in \mathbb{R}_{\geq 0}$ where $\sum_i \lambda_i = 1$, Jensen's inequality states that

$$f\left(\sum_i \lambda_i x_i\right) \leq \sum_i \lambda_i f(x_i). \tag{2.14}$$

Jensen's inequality is visualized geometrically in Figure 2.2. The triangle represents all possible weighted means between the three points, $(x_1, f(x_1))$, $(x_2, f(x_2))$ and $(x_3, f(x_3))$.

## 2.2 Application of Symmetric NMF: Cluster Analysis

The main application of symmetric NMF is clustering. Clustering is a form of unsupervised learning where a dataset is partitioned into disjoint groups. Samples with similar features are grouped together. In this way, clustering adds structure to the data and may provide the researcher with new insights. Section 2.2.1 shows why NMF can be used for clustering by demonstrating how symmetric NMF is closely related to kernel $k$-means. Section 2.2.2 presents a selection of typical applications of cluster analysis.

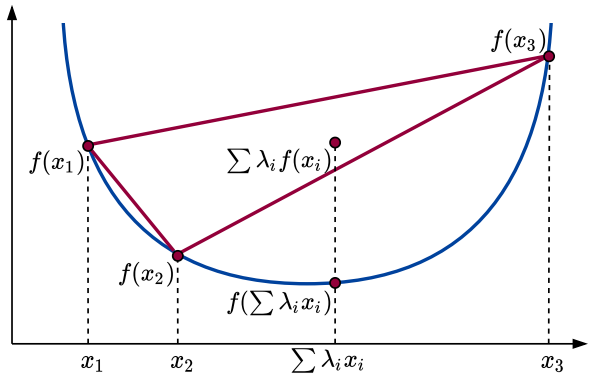



Figure 2.2: Illustration of Jensen's inequality

### 2.2.1 K-Means and Symmetric NMF

One of the most widely used techniques for cluster analysis is the $k$-means algorithm [32, 34]. The $k$-means algorithm partitions the dataset into $k$ clusters, each of which is represented by a *centroid* — the mean of all samples in that cluster. In this section, we first introduce the fundamentals of the $k$-means algorithm. Based on this foundation, we then present a sketch of the proof for how $k$-means is nearly equivalent to symmetric NMF with rank $r = k$ using the Euclidean distance.[3] The only difference is that $k$-means is a form of *hard clustering*, while symmetric NMF is a form of *soft clustering*. In hard clustering each sample is assigned to exactly one cluster. In soft cluster the samples may belong partially to multiple clusters.

#### K-Means

Let $\boldsymbol{Y} \in \mathbb{R}^{n \times m}$ be the original dataset to be clustered, where $n$ represents the number of samples and $m$ the dimensionality of the samples. Furthermore, let $\boldsymbol{C} = \{\boldsymbol{C}_l : l \in 1 \dots k\}$ be the set of all clusters, where each cluster $\boldsymbol{C}_l \subseteq \boldsymbol{Y}$ is a subset of the samples in $\boldsymbol{Y}$. The centroid $\boldsymbol{z}_l \in \mathbb{R}^m$ is the mean of the samples $\boldsymbol{y} \in \mathbb{R}^m$ in $\boldsymbol{C}_l$,

$$\boldsymbol{z}_l = \frac{1}{|\boldsymbol{C}_l|} \sum_{\boldsymbol{y} \in \boldsymbol{C}_l} \boldsymbol{y}. \tag{2.15}$$

Each sample $\boldsymbol{y}$ is assigned to the nearest centroid and the $k$-means algorithm chooses the clusters by minimizing the variance within each cluster. Hence, the

[3] In the remainder of the thesis we use both $k$ and $l$ as indices for the rank $r$. Due to convention we let $r = k$ only in this section.

algorithm optimizes for the loss function

$$L(\boldsymbol{C}) = \sum_{l} \sum_{\boldsymbol{y} \in \boldsymbol{C}_l} ||\boldsymbol{y} - \boldsymbol{z}_l||^2. \tag{2.16}$$

This loss function can be *kernelized*, meaning that it is rewritten in such a way that the samples only appear in the form of inner products $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_s a_s b_s$ [42],

$$L(\boldsymbol{C}) = \sum_{\boldsymbol{y} \in \boldsymbol{Y}} \langle \boldsymbol{y}, \boldsymbol{y} \rangle - \sum_{l} \frac{1}{|\boldsymbol{C}_l|} \sum_{\boldsymbol{y}_i \in \boldsymbol{C}_l} \sum_{\boldsymbol{y}_j \in \boldsymbol{C}_l} \langle \boldsymbol{y}_i, \boldsymbol{y}_j \rangle. \tag{2.17}$$

Here we use the subscripts $i, j$ to clarify that we sum over the cluster samples $\boldsymbol{y} \in \boldsymbol{C}_l$ twice.

One severe limitation of the standard $k$-means algorithm is that it generates spherical clusters with linear decision boundaries between them. Achieving good results therefore requires that the clusters are linearly separable. This can be remedied by applying a kernel function $K : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, instead of taking the inner product in (2.17). This gives a modified loss function

$$L_K(\boldsymbol{C}) = \sum_{\boldsymbol{y} \in \boldsymbol{Y}} K(\boldsymbol{y}, \boldsymbol{y}) - \sum_{l} \frac{1}{|\boldsymbol{C}_l|} \sum_{\boldsymbol{y}_i \in \boldsymbol{C}_l} \sum_{\boldsymbol{y}_j \in \boldsymbol{C}_l} K(\boldsymbol{y}_i, \boldsymbol{y}_j). \tag{2.18}$$

Like the inner product, kernel functions can be viewed as mapping a pair of samples to their similarity. However, the kernel functions implicitly transform the feature space of the dataset into a higher dimension before calculating the similarity. The clusters may be linearly separable in this higher dimension feature space. An illustration of this is shown in Figure 2.3.

One important advantage of this approach is that it is not necessary to explicitly map the dataset into a higher-dimensional feature space. Instead, the inner products between each pair of samples is calculated directly in the transformed feature space. The kernel functions that are used in practice avoid an explicit mapping, resulting in an operation that is far less computationally expensive. For this reason, this approach is commonly known as the *kernel trick* [47]. One drawback of kernel $k$-means is that the choice of kernel function is highly problem dependent.

### Equivalence to Symmetric NMF

It has been shown that symmetric NMF with rank $r = k$ using the squared Euclidean distance creates a clustering by minimizing the same loss function as kernel $k$-means clustering [12]. We summarize the main points of the proof here.

Let $i, j \in \{1 \dots n\}$ index the samples in $\boldsymbol{Y}$. The kernel matrix $\boldsymbol{X} \in \mathbb{R}^{n \times n}_{\geq 0}$ is defined by

$$X_{ij} = K(\boldsymbol{y}_i, \boldsymbol{y}_j). \tag{2.19}$$

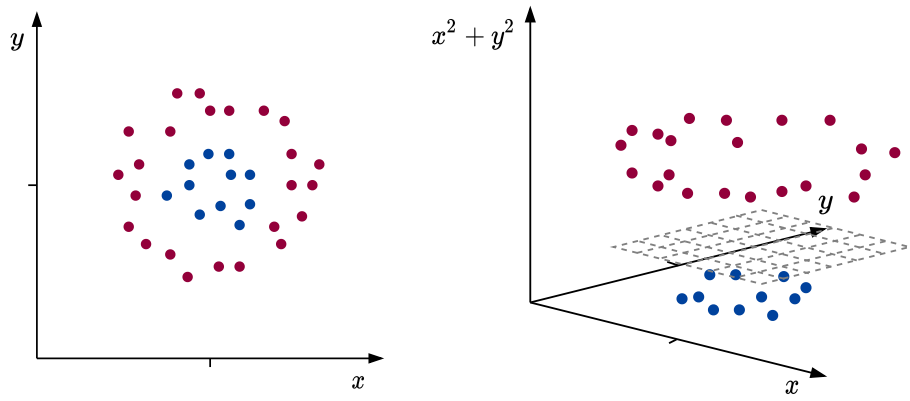



Figure 2.3: Illustration of how mapping a dataset into a higher dimension can make its clusters linearly separable

We require that $\boldsymbol{X}$ is nonnegative. Note that since $K(\boldsymbol{y}_i, \boldsymbol{y}_j)$ is equivalent to the inner product between the transformed images of $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$, $\boldsymbol{X}$ is always symmetric. Next, the clustering found by the $k$-means algorithm can be represented by $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{n \times k}$, where

$$W_{il} = \begin{cases} |\boldsymbol{C}_l|^{-1/2} & \text{if } \boldsymbol{y}_i \in \boldsymbol{C}_l, \\ 0 & \text{otherwise.} \end{cases} \tag{2.20}$$

Note that this imposes the orthogonality constraint

$$\boldsymbol{W}^T \boldsymbol{W} = \boldsymbol{I}, \tag{2.21}$$

which reflects the fact that $k$-means is a hard clustering algorithm. The trace of a matrix is the sum of the elements along its diagonal,

$$\mathrm{Tr}(\boldsymbol{X}) = \sum_i X_{ii}. \tag{2.22}$$

Using this definition, the kernelized loss function can be expressed as

$$L_K(\boldsymbol{C}) = \mathrm{Tr}(\boldsymbol{X}) - \mathrm{Tr}(\boldsymbol{W}^\top \boldsymbol{Y} \boldsymbol{W}). \tag{2.23}$$

Finally, minimizing (2.23) with respect to $\boldsymbol{W}$ is equivalent with minimizing the

squared Euclidean distance for symmetric NMF,

$$\arg\min_{\boldsymbol{W}} \left[\operatorname{Tr}(\boldsymbol{X}) - \operatorname{Tr}(\boldsymbol{W}^\top \boldsymbol{Y}\boldsymbol{W})\right] \tag{2.24}$$

$$= \arg\min_{\boldsymbol{W}} \left[-2\operatorname{Tr}(\boldsymbol{W}^\top \boldsymbol{Y}\boldsymbol{W})\right] \tag{2.25}$$

$$= \arg\min_{\boldsymbol{W}} \left[||\boldsymbol{X}||^2 - 2\operatorname{Tr}(\boldsymbol{W}^\top \boldsymbol{Y}\boldsymbol{W}) + ||\boldsymbol{W}^\top \boldsymbol{W}||^2\right] \tag{2.26}$$

$$= \arg\min_{\boldsymbol{W}} ||\boldsymbol{X} - \boldsymbol{W}\boldsymbol{W}^\top||^2. \tag{2.27}$$

The last step uses the orthogonality constraint from (2.21). It shows that kernel $k$-means clustering finds $\boldsymbol{W}$ by optimizing for the same objective as symmetric NMF. However, while the orthogonality constraint on $\boldsymbol{W}$ is approximated by symmetric NMF, it is not strictly imposed [12]. This is a result of $k$-means being a form of hard clustering, while symmetric NMF is a form of soft clustering. Nevertheless, the fact that symmetric NMF approximates the orthogonality constraint means that each sample will tend mostly to one cluster. This makes symmetric NMF suitable for clustering applications.

### 2.2.2 Use Cases for Cluster Analysis

Having discussed how symmetric NMF can be used to perform cluster analysis on data, we now turn to the question of why this is useful. The purpose of clustering is to generate meaningful groups in a dataset. These groups then typically serve as the starting point for further analysis. For example, the researcher may wish to assign a label to each group and summarize their key characteristics. The specifics of how cluster analysis is applied depends on the problem at hand, and clustering has been used successfully in a variety of domains [45]. This section presents three examples of problems that can be solved using clustering: text mining, image segmentation and gene expression analysis.

#### Text Mining

Text mining is one domain where cluster analysis can be applied [8]. The dataset is typically a large dataset of unlabeled documents, for example from an email database or search engine. Cluster analysis can be applied to segment the documents into groups based on their content. Arranging the documents into topics adds some structure to the dataset and facilitates further analysis.

Preprocessing involves converting each document into a fixed-size vector that represents it content. There are several ways to accomplish this. One approach is to remove stop words, generate a dictionary of terms and convert each document to a vector with the term frequencies. For each document and each term in the

dictionary, the term frequency is the number of times that term occurs in the document. This value is also weighted by how common the term is in the dataset. Rarer terms will typically be more useful in determining the topic of a document.

Regardless of the method used to generate the fixed-size vectors, the resulting matrix can be multiplied by itself to generate a symmetric matrix of pairwise similarities. The matrix of pairwise similarities can in turn be factorized using symmetric NMF to group the documents into the desirable number of clusters.

### Image Segmentation

Image segmentation is the partitioning of an image into meaningful clusters. It is also known as pixel classification, because each pixel is assigned to a group. There are numerous applications for image segmentation, including the detection of tumors in MRI images, classifying land cover in satellite imagery and finding road signs for self-driving cars [31].

One way to apply cluster analysis to an image is by flattening it and treating each pixel as a sample $\boldsymbol{x}_i$. The dimensionality of the samples then depends on the type of image. With the RGB color model each pixel is encoded by three integer values, whereas each pixel is represented by a single scalar in a black and white image. A pairwise similarity matrix is created by taking the inner product of every pair of samples $X_{ij} = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$, and this similarity matrix can be factorized using symmetric NMF [6].

### Gene Expression Analysis

Genes contain instructions that are used to create proteins — large biological molecules that serve a variety of functions. The expression level of a gene is a measure of how frequently it is used. Gene expression datasets contain the expression levels of a set of genes, usually measured across some other variable like time or patient. In the former case each sample will be a gene, and the goal may be to find genes with similar functionality. In the latter case, where the dataset contains gene expression levels versus patient, each patient represents a sample. The goal could then be to estimate the efficacy of treatments or chance of survival by grouping patients with similar profiles.

A pairwise similarity matrix can be constructed from gene expression datasets in the same manner as above. However, the clusters may be arbitrarily shaped and are unlikely to be linearly separable. Hence, it is often necessary to transform the dataset using a kernel function [2].

## 2.3 Literature Review

This section presents an overview of the literature that is most closely related to our work. Section 2.3.1 contains the literature review protocol we used for the literature search. The subsequent sections summarize the key contributions of papers we found. Section 2.3.2 and 2.3.3 present two important papers for understanding full-batch MU for NMF. Section 2.3.4 and 2.3.5 focus on the literature that exists on stochastic MU for linear NMF.

### 2.3.1 Literature Review Protocol

This subsection presents how we conducted the literature review. It explains quality criteria and limitations of the scope, as well as content and keywords for the search. Finally, it introduces two papers we include in addition to the results of the formal literature protocol.

We applied certain quality screening criteria in selecting the papers we include here. We only considered papers that provide either source code, pseudocode or formulas describing the algorithms they discuss. We also required that the claimed properties of the algorithms are either proven analytically or at least demonstrated empirically. Finally, all the works presented in this section have been published in peer-reviewed journals or conferences. Since MU algorithms were only introduced in 2001 [30], we limited our scope to papers published in this year or later.

In terms of content, the search was focused on stochastic MU. We included both linear and symmetric NMF, however, we only found stochastic approaches for linear factorization. There exists some literature on stochastic NMF using other algorithms [5, 54, 20, 40] than MU, but none of these papers considered symmetric NMF. We therefore concluded that they were not relevant for our work and do not discuss them here.

We extended the scope of stochastic MU approaches to include online and mini-batch MU as well, since the terms *online* and *mini-batch* describe similar approaches and are often used interchangeably. All three terms describe models that learn by processing a single or a few data points at a time, as opposed to full-batch models that use the whole dataset. However, online models are often used to describe models that handle streaming data. Unlike stochastic models, they do not necessarily require that the order of processing is randomized. What distinguishes mini-batch models from stochastic models is that the former operates on batches that contain multiple samples, whereas stochastic models generally process a single sample at a time. Despite these differences, there is considerable overlap between the online, mini-batch and stochastic models.

As previously mentioned, the field of stochastic MU is relatively small. The

result of our literature review was the work of two research groups, [43] and [25, 24]. The advantage of a limited literature base is that it allows us review these works in-depth. This is the subject of Section 2.3.4 and 2.3.5.

In addition to the papers named above, we include two foundational papers in the review. The first additional paper is the original work by Lee and Seung from 2001 [30]. The authors introduced the concept of MU algorithms and proved their convergence for Euclidean distance and I-divergence. The second paper introduced a unified development procedure for MU algorithms with different loss functions and was published by Yang and Oja in 2011 [51]. It presented a framework for deriving auxiliary functions and proving the convergence of multiplicative update rules for a wide range of loss functions.

These two papers were not the product of a formal literature search. However, we include them as a part of this review for two reasons. Firstly, they serve as an introduction to MU algorithms for NMF for the reader. Secondly, our theoretical analysis uses several ideas from these papers. Especially [51] is used heavily in our proof of convergence.

### 2.3.2 Lee-Seung Algorithms

Algorithms using multiplicative updates for NMF were first formulated by Lee and Seung in 2001 [30]. They introduced two iterative algorithms, sometimes referred to as the Lee-Seung algorithms, for performing linear NMF. One algorithm minimizes the Euclidean distance and the other one the I-divergence.

The multiplicative update rules for the Euclidean distance are

$$W_{ik} = W_{ik} \frac{(\boldsymbol{X}\boldsymbol{H}^\top)_{ik}}{(\boldsymbol{W}\boldsymbol{H}\boldsymbol{H}^\top)_{ik}} \tag{2.28}$$

$$H_{kj} = H_{kj} \frac{(\boldsymbol{W}^\top\boldsymbol{X})_{kj}}{(\boldsymbol{W}^\top\boldsymbol{W}\boldsymbol{H})_{kj}}. \tag{2.29}$$

The multiplicative update rules for the I-divergence are

$$W_{ik} = W_{ik} \frac{\sum_j H_{kj} X_{ij}/(\boldsymbol{W}\boldsymbol{H})_{ij}}{\sum_j H_{kj}} \tag{2.30}$$

$$H_{jk} = H_{jk} \frac{\sum_i W_{ik} X_{ij}/(\boldsymbol{W}\boldsymbol{H})_{ij}}{\sum_i H_{ik}}. \tag{2.31}$$

As shown in Algorithm 1, the Lee-Seung algorithms perform NMF by alternately updating $\boldsymbol{W}$ and $\boldsymbol{H}$ until some convergence criterion is achieved. Lee and Seung proved the convergence of this algorithm by constructing appropriate auxiliary functions and using the MM algorithm.

**Algorithm 1:** Lee-Seung algorithms

**Input :** $\boldsymbol{X}$
**Output:** $\boldsymbol{W}$, $\boldsymbol{H}$
$\boldsymbol{W} \leftarrow$ `RandomNonnegativeMatrix();`
$\boldsymbol{H} \leftarrow$ `RandomNonnegativeMatrix();`
**while** $\neg$*(convergence condition)* **do**
  $\boldsymbol{W} \leftarrow$ `MultiplicativeUpdateW(`$\boldsymbol{X}$, $\boldsymbol{W}$, $\boldsymbol{H}$`)` ; `//` (2.28) `or` (2.30)
  $\boldsymbol{H} \leftarrow$ `MultiplicativeUpdateH(`$\boldsymbol{X}$, $\boldsymbol{W}$, $\boldsymbol{H}$`)` ; `//` (2.29) `or` (2.31)

The Lee-Seung algorithms have been extended to a number of different loss functions and other NMF problems, including symmetric NMF. Furthermore, various algorithmic enhancements have been proposed to improve the rate of convergence [8].

### 2.3.3 Unified Development of Multiplicative Algorithms

Yang and Oja introduced a unified procedure for deriving multiplicative update rules with guaranteed convergence for a broad range of loss functions [51]. The procedure extends and generalizes the work of other researchers, who have previously derived methods covering the Bregman divergences [11] and most of the $\alpha$-divergences [7].

Specifically, the unified procedure is applicable to all loss functions that can be expressed as a finite sum of monomials. The monomials take the form $a\widehat{X}_{ij}^{b}$, where $a, b \in \mathbb{R}$ and $\widehat{X}_{ij}$ is the current approximation of $X_{ij}$. It must be possible to express the loss function as

$$L(\widehat{\boldsymbol{X}}) = \sum_{dij} a_{dij} \widehat{X}_{ij}^{b_d} + \text{constant}, \tag{2.32}$$

where $d$ identifies the monomial and $i, j$ are the indices of $\widehat{\boldsymbol{X}}$. Formulas for transforming logarithmic and nonseparable loss functions into the required format are provided in the paper.

The procedure constructs an auxiliary function based on the loss function in question. This is then used to obtain the update rules and prove convergence using the MM algorithm. The steps can be summarized as follows:

1. Transform the loss function into a finite sum of monomials.

2. Derive an auxiliary function by upper bounding concave monomials using Taylor's theorem and convex monomials using Jensen's inequality.

3. If there are more than two monomials, merge their bounds[4].

4. Differentiate the auxiliary function with respect to the factor matrix and set the derivative equal to zero to derive the update rules.

As explained in Section 2.1.4, the MM algorithm is based on three properties which together ensure that the loss is nonincreasing. Here, step 2 guarantees Property (P2) by upper bounding the loss function. Step 4 is equivalent to Property (P3).

The auxiliary function derived from the unified development procedure can always be expressed as

$$G(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) = \sum_{ik} W_{ik} \left[ \frac{\nabla_{ik}^{+}}{\psi_{\max}} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^{\psi_{\max}} - \frac{\nabla_{ik}^{-}}{\psi_{\min}} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^{\psi_{\min}} \right] + \text{constant}, \tag{2.33}$$

where $\psi_{\max}, \psi_{\min} \in \mathbb{R}$ and $\psi_{\max} > \psi_{\min}$. Since $\left(\frac{\widetilde{W}_{ik}}{W_{ik}}\right) = 1$ if $\widetilde{\boldsymbol{W}} = \boldsymbol{W}$, Property (P1) is guaranteed by an auxiliary function of this form.

In most cases, the update derived by setting the partial derivative of (2.33) equal to 0 will be of the form

$$W_{ik} = W_{ik} \left( \frac{\nabla_{ik}^{-}}{\nabla_{ik}^{+}} \right)^{1/(\psi_{\max} - \psi_{\min})}. \tag{2.34}$$

Exceptions include some loss functions that contain logarithms. However, updates for the I-divergence follow the usual format.

### Logarithmic Loss Functions

For loss functions that contain logarithms, such as the I-divergence, step 1 of the unified development procedure requires the use of the limit

$$\ln x = \lim_{\epsilon \to 0^{+}} \left( \frac{x^{\epsilon}}{\epsilon} - \frac{1}{\epsilon} \right). \tag{2.35}$$

The expression on the right-hand side takes the limit of two monomials. These can be upper bounded as any other monomials. We can differentiate and then subsequently apply the limit, as long as the loss function is smooth with respect to both $\epsilon$ and the factor matrix.

[4]Both the Euclidean distance and the I-divergence can be represented using two monomials. For this reason we do not discuss this step in detail.

### Quadratic Monomials

The unified development procedure can be applied to loss functions that yield quadratic monomials. This is the case for symmetric NMF, since $\widehat{\boldsymbol{X}} = \boldsymbol{W}\boldsymbol{W}^\top$. The construction of an auxiliary function that includes quadratic monomials follows the same steps as the linear case, using Taylor's theorem or Jensen's inequality.

### Example Derivation for Euclidean Distance

We conclude this section with an example of how the unified development procedure is used. We apply the steps defined above to symmetric NMF with Euclidean distance:

1. Writing the Euclidean distance as a finite sum of monomials gives

$$L_{EU}(\widetilde{\boldsymbol{X}}) = \sum_{ij}(X_{ij} - \widetilde{X}_{ij})^2 = \sum_{ij} -2X_{ij}\widetilde{X}_{ij} + \widetilde{X}_{ij}^2 + \text{constant}. \tag{2.36}$$

2. In order to upper bound the convex monomial

$$\sum_{ij}\widetilde{X}_{ij}^2 = \sum_{ij}\left(\sum_k \widetilde{W}_{ik}\widetilde{W}_{jk}\right)^2, \tag{2.37}$$

we introduce $\lambda_{ijk} = \frac{W_{ik}W_{jk}}{(\boldsymbol{W}\boldsymbol{W}^\top)_{ij}}$ and use Jensen's inequality from Section 2.1.6 to derive an upper bound,

$$\begin{aligned}
\sum_{ij}\widetilde{X}_{ij}^2 &= \sum_{ij}\left(\sum_k \lambda_{ijk}\frac{\widetilde{W}_{ik}\widetilde{W}_{jk}}{\lambda_{ijk}}\right)^2 && (2.38)\\
&\leq \sum_{ijk}\lambda_{ijk}\left(\frac{\widetilde{W}_{ik}\widetilde{W}_{jk}}{\lambda_{ijk}}\right)^2 && (2.39)\\
&= \sum_{ijk}\frac{\widetilde{W}_{ik}^2\widetilde{W}_{ij}^2}{W_{ik}W_{ij}}(\boldsymbol{W}\boldsymbol{W}^\top)_{ij} && (2.40)\\
&\leq \sum_{ik}\frac{\widetilde{W}_{ik}^4}{2\widetilde{W}_{ik}^3}\sum_j\left((\boldsymbol{W}\boldsymbol{W}^\top)_{ij} + (\boldsymbol{W}\boldsymbol{W}^\top)_{ji}\right)W_{jk} && (2.41)\\
&= \sum_{ik}\frac{\widetilde{W}_{ik}^4}{\widetilde{W}_{ik}^3}\sum_j(\boldsymbol{W}\boldsymbol{W}^\top)_{ij}W_{jk} && (2.42)
\end{aligned}$$

$$= \sum_{ik} \frac{W_{ik}}{4} \nabla_{ik}^{+} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^4 . \tag{2.43}$$

In (2.41), we follow [51] by using

$$\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \leq \sum_i \frac{x_i^2}{2y_i} (\boldsymbol{A}\boldsymbol{y} + \boldsymbol{A}^\top \boldsymbol{y}), \tag{2.44}$$

with $x_i = \widetilde{W}_{ik}^2 / W_{ik}$, $y_i = W_{ik}$ and $\boldsymbol{A} = \boldsymbol{W}\boldsymbol{W}^\top$. This inequality has been proven in [30] and [13].

The monomial $\sum_{ij} -2X_{ij}\widetilde{X}_{ij}$ is linear and therefore both convex and concave. For a complete example, we treat it as concave and apply the upper bound derived from the Taylor polynomial in Section 2.1.5. Let $f(\widetilde{\boldsymbol{X}} = \sum_{ij} -2X_{ij}\widetilde{X}_{ij}$, then

$$\sum_{ij} -2X_{ij}\widetilde{X}_{ij} \leq f(-\widehat{\boldsymbol{X}}) + \sum_{ij} (\widetilde{X}_{ij} - \widehat{X}_{ij}) \frac{\partial f}{\partial \widetilde{X}_{ij}} \bigg|_{\widetilde{\boldsymbol{X}} = (-\widehat{\boldsymbol{X}})} \tag{2.45}$$

$$= \sum_{ij} 2X_{ij}\widehat{X}_{ij} + \sum_{ij} \widetilde{X}_{ij} \frac{\partial}{\partial \widetilde{X}_{ij}} \sum_{ab} \left( -2X_{ab}\widetilde{X}_{ab} \right) \bigg|_{\widetilde{\boldsymbol{X}} = (-\widehat{\boldsymbol{X}})} \tag{2.46}$$

$$= \sum_{ij} 2X_{ij}\widehat{X}_{ij} + \sum_{ijk} -4X_{ij}\widetilde{W}_{ik}\widetilde{W}_{jk} \tag{2.47}$$

$$= \sum_{ij} 2X_{ij}\widehat{X}_{ij} - \sum_{ik} \widetilde{W}_{ik}\nabla_{ik}^{-} \tag{2.48}$$

$$= -\sum_{ik} \widetilde{W}_{ik}\nabla_{ik}^{-} + \text{constant}. \tag{2.49}$$

3. Since the Euclidean distance consists of two monomials only, no merging is needed and the auxiliary function is given by

$$G(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) = \sum_{ik} \left[ \frac{W_{ik}}{4} \nabla_{ik}^{+} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^4 - \widetilde{W}_{ik}\nabla_{ik}^{-} \right] + \text{constant}, \tag{2.50}$$

which fulfills Property (P1):

$$G(\boldsymbol{W}, \boldsymbol{W}) = \sum_{ik} \left[ \frac{W_{ik}}{4} \nabla_{ik}^{+} \left( \frac{W_{ik}}{W_{ik}} \right)^4 - W_{ik}\nabla_{ik}^{-} \right] + \text{constant} \tag{2.51}$$

$$= \sum_{ik} \left[ \frac{W_{ik}}{4} \nabla^+_{ik} - W_{ik} \nabla^-_{ik} \right] + \text{constant} \tag{2.52}$$

$$= \sum_{ikj} \left[ \frac{W_{ik}}{4} (4\widehat{X}_{ij} W_{jk}) - W_{ik} (4X_{ij} W_{jk}) \right] + \text{constant} \tag{2.53}$$

$$= \sum_{ij} \left[ \widehat{X}^2_{ij} - 4X_{ij} \widehat{X}_{ij} \right] + \text{constant} \tag{2.54}$$

$$= \sum_{ij} \left[ \widehat{X}^2_{ij} - 4X_{ij} \widehat{X}_{ij} \right] + \sum_{ij} \left[ X^2_{ij} + 2X_{ij} \widehat{X}_{ij} \right] \tag{2.55}$$

$$= \sum_{ij} \left[ \widehat{X}^2_{ij} - 2X_{ij} \widehat{X}_{ij} + X^2_{ij} \right] \tag{2.56}$$

$$= L_{EU}(\boldsymbol{W}). \tag{2.57}$$

4. Setting the derivative of (2.50) equal to 0 yields the update rule

$$\begin{aligned}
& \frac{\partial G(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} && = 0 \\
\Leftrightarrow \quad & \frac{\partial}{\partial \widetilde{W}_{al}} \sum_{ik} \left[ \frac{W_{ik}}{4} \nabla^+_{ik} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^4 - \widetilde{W}_{ik} \nabla^-_{ik} \right] + \text{constant} && = 0 \\
\Leftrightarrow \quad & \nabla^+_{ik} \left( \frac{\widetilde{W}_{ik}}{W_{ik}} \right)^3 - \nabla^-_{ik} && = 0 \\
\Leftrightarrow \quad & \widetilde{W}_{ik} = W_{ik} \left[ \frac{\nabla^-_{ik}}{\nabla^+_{ik}} \right]^{\frac{1}{3}} && (2.58)
\end{aligned}$$

### 2.3.4 Mini-Batch MU for Linear NMF

Serizel, Essid and Richard [43] presented several versions of mini-batch MU algorithms that are only applicable to linear NMF. This is distinct from our approach, which focuses on symmetric NMF. In [43], the authors introduced two basic algorithms which differ in how often the factors are updated, as well as two augmented versions using a gradient averaging technique.

The proposed approaches use sets of rows from the ground truth as mini-batches. These are denoted $\boldsymbol{W}_b$ and $\boldsymbol{X}_b$, where $b$ contains the indices of multiple rows. Since $\boldsymbol{H}$ contributes to the approximation in a column-wise fashion, an update based on $\boldsymbol{X}_b$ changes the full factor matrix $\boldsymbol{H}$.[5]

[5] In [43], the mini-batches consist of columns of $\boldsymbol{X}$. Here we use mini-batches with rows of $\boldsymbol{X}$ instead to facilitate the comparison with our approach.

**Algorithm 2:** Asymmetric Stochastic Gradient MU (ASG-MU)

```
Input  : X, mini-batches
Output: W, H
W ← RandomNonnegativeMatrix();
H ← RandomNonnegativeMatrix();
Shuffle(X);
while ¬(convergence condition) do
    Shuffle(mini-batches);
    for b in mini-batches do
        W_b ← MultiplicativeUpdateW(X_b, W_b, H);
        H ← MultiplicativeUpdateH(X_b, W_b, H);
```

Algorithm 2 shows the first variant introduced by [43], ASG-MU. The name comes from the asymmetry in how $\boldsymbol{W}$ and $\boldsymbol{H}$ are updated. While the full matrix $\boldsymbol{H}$ is updated with each mini-batch and thus multiple times per epoch, the full matrix $\boldsymbol{W}$ is only updated once per epoch. Note that the variable *mini-batches* is a list of lists of indices. It is also worth noting that `Shuffle()` only shuffles the first dimension. In other words, it shuffles the order of lists in *mini-batches*.

**Algorithm 3:** Greedy Stochastic Gradient MU (GSG-MU)

```
Input  : X, mini-batches
Output: W, H
W ← RandomNonnegativeMatrix();
H ← RandomNonnegativeMatrix();
Shuffle(X);
while ¬(convergence condition) do
    Shuffle(mini-batches);
    for b in mini-batches do
        W_b ← MultiplicativeUpdateW(X_b, W_b, H);
    H ← MultiplicativeUpdateH(X_b, W_b, H);
```

The second variant is GSG-MU, shown in Algorithm 3. It differs from ASG-MU only in the way $\boldsymbol{H}$ is updated. While ASG-MU updates $\boldsymbol{H}$ multiple times per epoch, GSG-MU only updates $\boldsymbol{H}$ once per epoch using the last mini-batch. Since the mini-batches are shuffled at the beginning of each epoch, this is equivalent with choosing a random mini-batch.

ASG-MU and GSG-MU can be extended by adding gradient averaging. The

goal of gradient averaging is to increase convergence speed by reducing the variability of the gradient. The extensions are named Asymmetric Stochastic Average Gradient (ASAG) MU and Greedy Stochastic Average Gradient (GSAG) MU, respectively. Gradient averaging is performed by incorporating the stochastic gradient into the current gradient estimate. Let $\nabla^-$ and $\nabla^+$ denote the negative and positive components of the gradient estimate, while $\lambda \in (0, 1]$ denotes the forgetting factor. In each iteration, the gradient estimates are updated by

$$\nabla^- \leftarrow (1 - \lambda)\nabla^- + \lambda\nabla_b^- \tag{2.59}$$

$$\nabla^+ \leftarrow (1 - \lambda)\nabla^+ + \lambda\nabla_b^+. \tag{2.60}$$

In [43], ASG-MU, GSG-MU, ASAG-MU and GSAG-MU were all tested on a speech recognition dataset with regular MU as a baseline. The authors found that GSG-MU and GSAG-MU converges at a rate similar to the baseline, while ASG-MU and ASAG-MU converge significantly faster.

### 2.3.5 Variance Reduced Stochastic MU for Linear NMF

Kasai presented several algorithms that use stochastic multiplicative updates in his two papers from 2018 [25, 24]. The algorithms are specific to Euclidean distance and only applicable to linear NMF, as opposed to symmetric NMF in our approach. The first paper introduced SVRMU and its two extensions, SVRMU-ACC and R-SVRMU. In the second paper Kasai presented SAGMU, an improved version of SVRMU. In this section, we first present SVRMU and then explain SAGMU based on SVRMU.

#### SVRMU

SVRMU uses variance reduction to improve the performance of stochastic MU. Factorization is controlled by two loops. An execution of the outer loop is referred to as an epoch, and an execution of the inner loop as an iteration. In each iteration, the stochastic contributions are combined with contributions that are calculated once per epoch and stored. To clearly distinguish between calculations that are performed every iteration and the stored components, let the latter be denoted $\left[\boldsymbol{W}^\top \boldsymbol{X}/|n|\right]^{\text{stored}}$ and $\left[\boldsymbol{W}^\top \boldsymbol{W}\boldsymbol{H}/|n|\right]^{\text{stored}}$.

Following the notation from the previous section, let $\boldsymbol{X}_b$ be the mini-batch of $\boldsymbol{X}$ entries and let $\boldsymbol{W}_b$ denote the corresponding rows in $\boldsymbol{W}$. The values used to

update $\boldsymbol{H}$ are

$$\upsilon^- = \frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}} \tag{2.61}$$

$$\upsilon^+ = \frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{H}}{|n|}\right]^{\text{stored}} . \tag{2.62}$$

The most computationally expensive terms only need to be computed once per epoch. Hence this technique reduces the variance at a limited computational cost. SVRMU is shown in Algorithm 4.

**Algorithm 4:** Stochastic Variance Reduced MU (SVRMU)

**Input :** $\boldsymbol{X}$, *mini-batch-size*, *n-iter*
**Output:** $\boldsymbol{W}$, $\boldsymbol{H}$
$\boldsymbol{W} \leftarrow$ `RandomNonnegativeMatrix()`;
$\boldsymbol{H} \leftarrow$ `RandomNonnegativeMatrix()`;
**while** $\neg$*(convergence condition)* **do**
  Compute the components of $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$ and $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{W}^e \boldsymbol{H}^e}{|n|}\right]^{\text{stored}}$;
  **for** $t = 1, 2, \ldots,$ *n-iter* **do**
    $b \leftarrow$ `RandomMiniBatch(`*mini-batch-size*`)`;
    $\boldsymbol{W}_b \leftarrow$ `MultiplicativeUpdateW(`$\boldsymbol{X}_b$`,` $\boldsymbol{W}_b$`,` $\boldsymbol{H}$`)`;
    $\upsilon^- \leftarrow \frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$;
    $\upsilon^+ \leftarrow \frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{H}}{|n|}\right]^{\text{stored}}$;
    $\boldsymbol{H} \leftarrow \boldsymbol{H} \odot \frac{\upsilon^-}{\upsilon^+}$;

An accelerated version of SVRMU, called SVRMU-ACC, takes advantage of the fact that updating $\boldsymbol{W}_b$ is much faster than updating $\boldsymbol{H}$ for small batch sizes. Instead of only updating $\boldsymbol{W}_b$ once for that mini-batch, it repeatedly updates $\boldsymbol{W}_b$ until some convergence criterion is reached. The convergence criterion in the outermost loop may be different from the convergence criterion in the innermost loop. This is somewhat similar to GSG-MU and GSAG-MU, which balance out the frequency of updates for $\boldsymbol{W}$ and $\boldsymbol{H}$. However, while those algorithms achieve this by only updating $\boldsymbol{H}$ once each epoch, SVRMU-ACC increases the time spent on updating each mini-batch of rows from $\boldsymbol{W}$.

The final variant of SVRMU is Robust SVRMU (R-SVRMU). It is based on the success of robust NMF [22] and robust online NMF [53]. The approach is designed to handle outliers in the dataset $\boldsymbol{X}$. It does this using an outlier matrix

$\boldsymbol{R} \in \mathbb{R}^{n \times m}$, where $\boldsymbol{W}\boldsymbol{H} + \boldsymbol{R} \approx \boldsymbol{X}$. Using a regularization parameter $\gamma > 0$ this gives the following set of updates for R-SVRM,

$$\boldsymbol{W}_b^{\text{new}} = \boldsymbol{W}_b \odot \frac{\boldsymbol{X}_b \boldsymbol{H}^\top}{(\boldsymbol{W}_b \boldsymbol{H} + \boldsymbol{R}_b)\boldsymbol{H}^\top}$$
$$\boldsymbol{R}_b^{\text{new}} = \boldsymbol{R}_b \odot \frac{\boldsymbol{X}_b}{\boldsymbol{W}_b \boldsymbol{H} + \boldsymbol{R}_b + \gamma}$$

$$\upsilon^- = \frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|} + \left[\frac{\boldsymbol{W}_b^\top (\boldsymbol{W}_b \boldsymbol{H} + \boldsymbol{R}_b)}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$$
$$\upsilon^+ = \frac{\boldsymbol{W}_b^\top (\boldsymbol{W}_b \boldsymbol{H} + \boldsymbol{R}_b)}{|b|} + \left[\frac{(\boldsymbol{W}_b)^\top \boldsymbol{X}_b}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top (\boldsymbol{W}\boldsymbol{H} + \boldsymbol{R})}{|n|}\right]^{\text{stored}}.$$

The steps of R-SVRMU are essentially the same as in basic SVRMU, but the updates are modified to incorporate the outlier matrix $\boldsymbol{R}$ and the regularization parameter $\gamma$.

Kasai [25] compared SVRMU and its extensions to ASAG-MU and GSG-MU from the previous section. The author showed that SVRMU and SVRMU-ACC outperform ASAG-MU on a synthetic dataset, with SVRMU-ACC achieving the fastest convergence speed by a significant margin.

### SAGMU

SAGMU and its two extensions, SAGMU-ACC and R-SAGMU were presented in [24]. These are very similar to SVRMU and its extensions. In fact, the only difference lies in the treatment of stored values. While SVRMU updates the stored values once per epoch, SAGMU initializes its stored values to zero and then updates them every iteration.

SAGMU is shown in Algorithm 5. With each iteration a mini-batch is chosen uniformly at random and the stored values are updated. This eliminates the need for an inner loop. It also means that each iteration becomes somewhat more expensive, but it removes the need for an expensive full-batch computation every epoch. Furthermore, constantly updating the stored values may improve the quality of those estimates and in turn increase convergence speed. The extensions of SAGMU use the same techniques as SVRMU-ACC and R-SVRMU.

Kasai found that SAGMU-ACC converges faster than both ASAG-MU and SVRMU-ACC on several synthetic datasets. Furthermore, R-SAGMU outperforms R-SVRMU on an image dataset with randomly added outliers [24].

**Algorithm 5:** Stochastic Average Gradient MU (SAGMU)

**Input :** $\boldsymbol{X}$, *mini-batch-size*
**Output:** $\boldsymbol{W}$, $\boldsymbol{H}$
$\boldsymbol{W} \leftarrow$ `RandomNonnegativeMatrix()`;
$\boldsymbol{H} \leftarrow$ `RandomNonnegativeMatrix()`;
Initialize the values in $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$ and $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{W}^e \boldsymbol{H}^e}{|n|}\right]^{\text{stored}}$ to zeros;
**while** $\neg$*(convergence condition)* **do**
  $b \leftarrow$ `RandomMiniBatch(`*mini-batch-size*`)`;
  $\boldsymbol{W}_b \leftarrow$ `MultiplicativeUpdateW(`$\boldsymbol{X}_b$, $\boldsymbol{W}_b$, $\boldsymbol{H}$`)`;
  $\upsilon^- \leftarrow \frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$;
  $\upsilon^+ \leftarrow \frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|} + \left[\frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|}\right]^{\text{stored}} + \left[\frac{\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{H}}{|n|}\right]^{\text{stored}}$;
  $\boldsymbol{H} \leftarrow \boldsymbol{H} \odot \frac{\upsilon^-}{\upsilon^+}$;
  Update $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{X}}{|n|}\right]^{\text{stored}}$ using $\frac{\boldsymbol{W}_b^\top \boldsymbol{X}_b}{|b|}$;
  Update $\left[\frac{(\boldsymbol{W}^e)^\top \boldsymbol{W}^e \boldsymbol{H}^e}{|n|}\right]^{\text{stored}}$ using $\frac{\boldsymbol{W}_b^\top \boldsymbol{W}_b \boldsymbol{H}}{|b|}$;

# 3 Stochastic Multiplicative Updates for Symmetric NMF

In this chapter we present a novel algorithm for stochastic multiplicative updates for symmetric NMF. We call the algorithm Stochastic Bound-and-Scale Multiplicative Updates (SBSMU). Based on [51], we use the following notation throughout this section: $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{n \times n}$ is the symmetric ground truth matrix, while $\widehat{\boldsymbol{X}} = \boldsymbol{W}\boldsymbol{W}^\top$ is its approximation. The factor matrix is denoted $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{n \times r}$, where $r$ is the rank. We let $\boldsymbol{W}$, $\boldsymbol{W}^{\text{new}}$ and $\widetilde{\boldsymbol{W}}$ denote the current estimate of the factor, the new estimate and the factor as a variable, respectively. Likewise, we use $\widetilde{\boldsymbol{X}}$ in place of $\widehat{\boldsymbol{X}}$ to indicate that the approximation contains $\widetilde{\boldsymbol{W}}$.

Section 3.1 contains the algorithm itself, which uses a technique for variance reduction we called *bound-and-scale*. Section 3.2 introduces a proof showing that our algorithm converges in expectation given a certain condition on the bound-and-scale parameter. Finally, Section 3.3 presents how we optimized and implemented the algorithm. While much of what is introduced in this chapter is applicable to various loss functions, we focus exclusively on the Euclidean distance and the I-divergence.

## 3.1 Algorithm

Comparing stochastic gradient descent (SGD) to gradient descent is a useful analog to how we introduce stochasticity to MU. Instead of using the full dataset to calculate the gradient each iteration, SGD uses a random subset of the entries — a mini-batch. The idea is to replace loss and gradient calculations that involve the full dataset with stochastic approximations based on a fraction of the data. This has two potential advantages. First, it significantly reduces the computation time per iteration. The complexity of an update is reduced from $O(m)$, where $m$ is the size of the dataset, to $O(b)$, where $b$ is the size of each mini-batch. If the stochastic approximations of the gradient are sufficiently accurate, this will increase the rate of convergence. The second advantage is that SGD sometimes is able to find a better local optimum than full-batch gradient descent. The noise introduced by stochasticity can lead SGD to escape local minima, and in turn find a better solution. The success of SGD is an important motivation for introducing stochasticity to MU.

In this section we analyze how stochasticity can be introduced to MU for symmetric NMF. The stochastic derivative calculation is based on stochastic approximations of the loss, which are introduced in Section 3.1.1. This is followed by the derivation of stochastic partial derivatives in Section 3.1.2. However, simply replacing the full-batch derivatives by the stochastic derivatives does not yield a converging algorithm. In Section 3.1.3 we discuss why this naive approach fails. Section 3.1.4 presents our algorithm, SBSMU, which remedies the problems with the naive approach.

### 3.1.1 Stochastic Approximation of Loss

The basis for introducing stochasticity to MU are stochastic approximations of the loss functions we wish to optimize. Each stochastic approximation is calculated based on a mini-batch. The mini-batch size determines how many entries of $X_{ij}$ are used in the stochastic loss calculation every iteration. In general, a larger mini-batch size means more accurate estimates but slower iterations. In this work we use a single entry of the ground truth $\boldsymbol{X}$ in each mini-batch. However, we utilize the symmetry of $\boldsymbol{X}$ by including both $X_{ij}$ and $X_{ji}$.

There are two reasons why we use mini-batches with only two symmetric entries. Firstly, this means that we only update two rows of the factor matrix in each iteration. As we will see in Section 3.3.5, this simplifies parallelization. Secondly, preliminary experiments indicated that increasing the mini-batch size does not improve performance.

In order to approximate the loss based on only two entries, the loss function needs to be additively separable over the entries of $\boldsymbol{X}$. Hence, we require that the loss function is of the form

$$L(\boldsymbol{W}) = \sum_{ij} l(X_{ij}, \widehat{X}_{ij}). \tag{3.1}$$

Each summand in (3.1) only depends on two rows of $\boldsymbol{W}$, since $\widehat{X}_{ij} = \sum_k W_{ik}W_{jk}$. This means that the gradient of $l(X_{ij}, \widehat{X}_{ij})$ is zero for all elements of $\boldsymbol{W}$ except these two rows. Note that both the Euclidean distance and the I-divergence are sums and satisfy the separability requirement.

The stochastic approximation of the loss is then defined by including only the corresponding summands of the loss function. Due to symmetry, $X_{ij} = X_{ji}$ and $\widehat{X}_{ij} = \sum_k W_{ik}W_{jk} = \widehat{X}_{ji}$. Hence we define a stochastic loss function $L^{(ij)}$ as

$$L^{(ij)}(\boldsymbol{W}) = l(X_{ij}, \widehat{X}_{ij}) + l(X_{ji}, \widehat{X}_{ji}) = 2l(X_{ij}, \widehat{X}_{ij}). \tag{3.2}$$

### 3.1.2 Derivation of Stochastic Partial Derivatives

Multiplicative updates can be formulated based on the partial derivative of the loss function. Similarly to how SGD replaces gradients with stochastic gradients, we use the partial derivative of the stochastic loss function instead. We denote the stochastic partial derivative

$$\nabla_{ij,al} = \frac{\partial L^{(ij)}(\widetilde{\boldsymbol{W}})}{\partial \widetilde{W}_{al}}. \tag{3.3}$$

The update rule is constructed using the positive and negative components of the gradient separately. In alignment with the full-batch updates in Section 2.1.3, we denote them

$$\nabla_{ij,al} = \nabla^{+}_{ij,al} - \nabla^{-}_{ij,al}. \tag{3.4}$$

Here, we derive the partial derivative and its components both for the Euclidean distance and the I-divergence.

#### Euclidean Distance

First we calculate the partial derivative of the stochastic loss function,

$$\begin{aligned}
\nabla_{ij,al} &= \frac{\partial}{\partial \widetilde{W}_{al}} 2\left(X_{ij} - \widetilde{X}_{ij}\right)^2 && (3.5)\\
&= 4\left(X_{ij} - \widehat{X}_{ij}\right)\frac{\partial}{\partial \widetilde{W}_{al}}\left(X_{ij} - \sum_k \widetilde{W}_{ik}\widetilde{W}_{jk}\right) && (3.6)\\
&= 4\left(\widehat{X}_{ij} - X_{ij}\right)\frac{\partial}{\partial \widetilde{W}_{al}}\left(\widetilde{W}_{il}\widetilde{W}_{jl}\right). && (3.7)
\end{aligned}$$

This gives four cases depending on the value of $a$, as shown in Table 3.1.

Table 3.1: Stochastic partial derivatives for the Euclidean distance

| **Case** | $\nabla_{ij,al}$ | $\nabla^{-}_{ij,al}$ | $\nabla^{+}_{ij,al}$ |
|---|---|---|---|
| $a = i \cap a \neq j$ | $4(\widehat{X}_{ij} - X_{ij})W_{jl}$ | $4X_{ij}W_{jl}$ | $4\widehat{X}_{ij}W_{jl}$ |
| $a \neq i \cap a = j$ | $4(\widehat{X}_{ij} - X_{ij})W_{il}$ | $4X_{ij}W_{il}$ | $4\widehat{X}_{ij}W_{il}$ |
| $a = i = j$ | $8(\widehat{X}_{ij} - X_{ij})W_{al}$ | $8X_{ij}W_{al}$ | $8\widehat{X}_{ij}W_{al}$ |
| $a \neq i \cap a \neq j$ | 0 | 0 | 0 |

**I-Divergence**

Likewise, we derive the partial derivative with its positive and negative parts for the I-divergence,

$$\nabla_{ij,al} = \frac{\partial}{\partial \widetilde{W}_{al}} 2 \left( X_{ij} \ln \frac{X_{ij}}{\widetilde{X}_{ij}} - X_{ij} + \widetilde{X}_{ij} \right) \tag{3.8}$$

$$= 2 \frac{\partial}{\partial \widetilde{W}_{al}} \left( -X_{ij} \ln \widetilde{X}_{ij} + \widetilde{X}_{ij} \right) \tag{3.9}$$

$$= 2 \left( 1 - \frac{X_{ij}}{\widetilde{X}_{ij}} \right) \frac{\partial}{\partial \widetilde{W}_{al}} \left( \widetilde{W}_{il} \widetilde{W}_{jl} \right). \tag{3.10}$$

This also gives four cases depending on $a$, shown in Table 3.2.

Table 3.2: Stochastic partial derivatives for the I-divergence

| **Case** | $\nabla_{ij,al}$ | $\nabla^{-}_{ij,al}$ | $\nabla^{+}_{ij,al}$ |
|---|---|---|---|
| $a = i \cap a \neq j$ | $2(1 - X_{ij}/\widehat{X}_{ij})W_{jl}$ | $2W_{jl}X_{ij}/\widehat{X}_{ij}$ | $2W_{jl}$ |
| $a \neq i \cap a = j$ | $2(1 - X_{ij}/\widehat{X}_{ij})W_{il}$ | $2W_{il}X_{ij}/\widehat{X}_{ij}$ | $2W_{il}$ |
| $a = i = j$ | $4(1 - X_{ij}/\widehat{X}_{ij})W_{al}$ | $4W_{al}X_{ij}/\widehat{X}_{ij}$ | $4W_{al}$ |
| $a \neq i \cap a \neq j$ | 0 | 0 | 0 |

### 3.1.3 Failure of the Naive Approach

In general, the naive approach to stochastic MU for symmetric NMF does not converge. Replacing the full-batch partial derivatives with the stochastic partial derivatives presented in Section 3.1.2 without further modification yields a stochastic update

$$W^{\text{new}}_{al} \leftarrow W_{al} \left[ \frac{\nabla^{-}_{ij,al}}{\nabla^{+}_{ij,al}} \right]^{\eta}. \tag{3.11}$$

Preliminary experiments showed that factorization by (3.11) fails to converge for symmetric NMF.

The reason for the failure of this naive approach can be explained by considering updates calculated from zero-valued entries in $\boldsymbol{X}$. For both Euclidean distance and I-divergence,

$$\nabla^{-}_{ij,al} \Big|_{X_{ij}=0} = 0, \tag{3.12}$$

for any iteration with $X_{ij} = 0$. This gives the update

$$W_{ik} \leftarrow W_{ik} \odot \left( \frac{\nabla^-_{ij,ik}}{\nabla^+_{ij,ik}} \right)^{\eta} \Bigg|_{X_{ij}=0, W_{ik}\neq 0} = 0. \tag{3.13}$$

Regardless of the previous value, all entries in the rows $i$ and $j$ in $\boldsymbol{W}$ are set to zero. As a result, subsequent updates targeting those rows do not have any effect. This means that a single stochastic update cancels out all previous or future updates for some rows in $\boldsymbol{W}$. These rows therefore only model zero entries and do not have the capacity of approximating the non-zero entries of $\boldsymbol{X}$. The problem is particularly acute for the sparse matrices that are most relevant to us. A sparse $\boldsymbol{X}$ will often have at least one zero entry per row or column, and this will quickly lead to a factor matrix $\boldsymbol{W}$ of all zeros.

Adding a small constant to each update in order to avoid zero entries is not sufficient to remedy this problem. Once the algorithm encounters a zero-valued entry in $\boldsymbol{X}$, the two corresponding rows in $\boldsymbol{W}$ are set to very small values. When a later update for a nonzero entry $X_{ij}$ uses either of these rows, the approximation $\widehat{X}_{ij}$ also becomes very small. Since the naive updates for both Euclidean distance and I-divergence are calculated with $\widehat{X}_{ij}$ in the denominator, they become very large in turn. An update of this magnitude is required to substantially change the very small entries of $\boldsymbol{W}$. However, it causes the other entries to become very large, leading to an imbalance that causes the algorithm to diverge. Eventually, some entries become so large that overflow errors arise.

Zero-valued entries in $\boldsymbol{X}$ are not a problem in full-batch MU, since the corresponding entries in $\boldsymbol{W}$ are only set to 0 if this is in fact the optimal value. This is a result of the full-batch partial derivative calculation. The full-batch partial derivative,

$$\nabla_{ik} = \frac{\partial L}{\partial \widetilde{W}_{ik}}(\widetilde{\boldsymbol{W}}), \tag{3.14}$$

takes into account all the entries in $\widetilde{\boldsymbol{X}}$ that contain $\widetilde{W}_{ik}$. As a consequence, the negative component $\nabla^-_{ik}$ will only be 0 if all the corresponding entries in $\boldsymbol{X}$ is also 0. It is easy to see that in this case, $W_{ik} = 0$ is the optimal value.

Of the stochastic algorithms for linear NMF introduced in Section 2.3, GSG-MU and ASG-MU are susceptible to this problem with zero-valued entries. GSG-MU and ASG-MU both use mini-batches containing complete rows of $\boldsymbol{X}$. Since $\widehat{\boldsymbol{X}} = \boldsymbol{W}\boldsymbol{H}$ in linear NMF, the updates effect a subset of the rows in $\boldsymbol{W}$ and all entries in $\boldsymbol{H}$. Hence, a column with all zeros in the mini-batch would produce a zero-update for the corresponding column in $\boldsymbol{H}$. The optimal value for that column is not necessarily zero and the problem described above applies. In contrast, ASAG-MU and GSAG-MU, as well as SVRMU and its variants, are

unlikely to have this problem due to their use of variance reduction. Although the algorithms differ in how they use variance reduction, they all ensure that each update contains information from previous updates.

### 3.1.4 Bound-and-Scale Algorithm

In this section we introduce Stochastic Bound-and-Scale Multiplicative Updates (SBSMU), our novel stochastic MU algorithm for symmetric NMF. It is summarized in Algorithm 6. To overcome the challenges presented in the previous section, we augment the naive approach with a variance reduction technique we call *bound-and-scale*. It is parameterized with $\alpha \in [0, 1)$. Each iteration, we randomly choose two indices $i$ and $j$ and perform the update

$$W_{al}^{\text{new}} = W_{al} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,al}}{\alpha + (1-\alpha)\nabla^{+}_{ij,al}} \right)^{\eta}. \tag{3.15}$$

The bound-and-scale technique is applied to both components of the partial derivative. Each component is lower bounded by $\alpha$ and scaled down by $(1-\alpha)$.

**Algorithm 6:** Stochastic Bound-and-Scale Multiplicative Updates

**Input :** $\boldsymbol{X}$, $\eta$, $\alpha$
**Output:** $\boldsymbol{W}$
$\boldsymbol{W} \leftarrow$ `RandomNonnegativeMatrix()`;
**while** $\neg$*(convergence condition)* **do**
  $i, j \leftarrow$ `SelectRandomIndices()`;
  **for** $k$ **in** $1 \dots r$ **do**
    $$W_{ik} \leftarrow W_{ik} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,ik}}{\alpha + (1-\alpha)\nabla^{+}_{ij,ik}} \right)^{\eta};$$
    $$W_{jk} \leftarrow W_{jk} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,jk}}{\alpha + (1-\alpha)\nabla^{+}_{ij,jk}} \right)^{\eta};$$

The bound-and-scale parameter $\alpha$ is independent of the magnitude of the partial derivative components. Although each gradient component is scaled by $(1-\alpha)$, there is no mechanism to prevent gradient components that are so large as to make $\alpha$ virtually negligible. Likewise, there is nothing that prevents gradient components that are so small that $\alpha$ dominates the fraction. These issues could be compounded by cases where the magnitude of $\nabla^{-}_{ij,al}$ and $\nabla^{+}_{ij,al}$ varies significantly between entries. Consider the case where $\alpha$ is nearly negligible for one entry, while simultaneously dominating another entry. This may lead to an imbalance in how

different parts of $\boldsymbol{W}$ are updated, with unpredictable results. However, as we show in the experiments in Section 4, there appears to be values for $\alpha$ that work well across datasets in practice. Normalizing $\boldsymbol{X}$ seems to remedy the problem by reducing the variation in gradient component magnitude between datasets.

Bound-and-scale is similar to the variance reduction methods used in GSAG-MU and SVRMU, presented in Sections 2.3.4 and 2.3.5. In statistics, two important concepts for a series of estimates are accuracy and precision. The accuracy of one estimate describes how close this estimate is to the true value. The precision, in contrast, describes how close the estimate is to the other estimates in the series. Recall that the stochastic partial derivative is an estimate of the full-batch partial derivative. While SBSMU cannot improve the accuracy of these estimates, it clearly increases precision by reducing their variance. Hence, the bound-and-scale step can be considered a variance reduction technique. Unlike the form of variance reduction in GSAG-MU and SVRMU, however, the bound-and-scale step is not a variant of gradient averaging and does not rely on information about previous gradients. One advantage of this is that SBSMU does not require any extra memory.

To the best of our knowledge, SBSMU is the first stochastic MU algorithm for symmetric NMF. This distinguishes it from the stochastic MU algorithms presented in the literature review in Section 2.3. Those algorithms are only applicable to linear NMF, as opposed to symmetric NMF in our approach. Furthermore, the algorithms from the literature review use mini-batches containing complete rows or columns from $\boldsymbol{X}$. In contrast, SBSMU only uses a single sample at a time. One advantage of the purely stochastic approach used by SBSMU is that this enables parallelization without locking, as we demonstrate in Section 3.3.5.

## 3.2 Proof of Convergence

This section presents a proof of convergence of SBSMU. The proof relies on the MM algorithm from Section 2.1.4, as well as the unified development procedure from Section 2.3.3. We denote the loss function, that is either the I-divergence or the Euclidean distance, by $L$. Furthermore, $L^{(ij)}$ is the corresponding stochastic loss function as defined in Section 3.1.1. The proof is not specific to Euclidean distance and I-divergence and could therefore hold for other separable loss functions as well. However, since it relies on assumptions that only have been evaluated for Euclidean distance and I-divergence, no claims can be made beyond these two loss functions.

Section 3.2.1 shows how the unified procedure can be applied to a stochastic loss function. While this is sufficient to majorize the stochastic loss, it is not sufficient to show convergence for the total loss. This is the main challenge in

proving convergence, and Section 3.2.2 outlines how the bound-and-scale step solves it. Building on this analysis, we introduce the main theorem with the proof of convergence in Section 3.2.3. Finally, we present a set of remarks on the proof and the convergence condition in Section 3.2.4.

### 3.2.1 Majorizing the Stochastic Loss

By applying the unified development procedure from Section 2.3.3, we can majorize the stochastic loss function. Recall that the unified procedure transforms a loss function $L$ into an equivalent polynomial form that is additively separable over the entries of the ground truth matrix. Then, Jensen's inequality and Taylor's theorem are applied separately to upper bound the monomials of each summand. For SBSMU, we require $L$ to be separable in order to define a stochastic loss $L^{(ij)}$ as in Section 3.1.1. $L^{(ij)}$ can be thought of as a loss function over a matrix with a single element. Furthermore, for both the Euclidean distance and the I-divergence, $L^{(ij)}$ can be expressed as a polynomial with respect to $\widehat{X}_{ij}$. To majorize the monomials of $L^{(ij)}$, we can thus apply the unified development procedure directly.

***Lemma* 3.2.1** (Stochastic Majorization Function)**.** There exist $\psi_{\max}, \psi_{\min} \in \mathbb{R}$ with $\psi_{\max} > \psi_{\min}$, such that

$$G^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) = \sum_{al} \left[ \frac{W_{al}}{\psi_{\max}} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}} \nabla^{+}_{ij,al} - \frac{W_{al}}{\psi_{\min}} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}} \nabla^{-}_{ij,al} \right] + \text{constant} \tag{3.16}$$

majorizes $L^{(ij)}$, that is

$$G^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) = L^{(ij)}(\boldsymbol{W}) \tag{3.17}$$

$$G^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) \geq L^{(ij)}(\widetilde{\boldsymbol{W}}) \tag{3.18}$$

for all $i$, $j$.

*Proof.* See the proof of Theorem 1 in [51], which is the basis of the unified development procedure. For both the Euclidean distance and the I-divergence, the procedure is applicable as the loss functions can be expressed in the required form. $L^{(ij)}$ can be viewed as the loss $2L$ of a matrix with a single entry. The proof can thus be applied directly to $L^{(ij)}$. □

### 3.2.2 Bound-and-Scale-Divergence

$G^{(ij)}$ majorizes $L^{(ij)}$. However, $L^{(ij)}$ depends on the indices $i, j$ chosen for the update. Herein lies the challenge. We need to extend this to the deterministic loss function $L$, which also depends on all the other indices $a, b \neq i, j$. For clarity, we define the concepts of *self-change* and *cross-change*. Let $\boldsymbol{W}$ denote the current value of the factor matrix, and $\boldsymbol{W}^{ij,\text{new}}$ the new value after applying an iteration of SBSMU with indices $i$ and $j$.

**Definition 3.2.1** (Self-Change)**.**

$$L^{(ij)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ij)}(\boldsymbol{W}) \tag{3.19}$$

is called self-change and

$$\mathbb{E}_{ij}\left[L^{(ij)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ij)}(\boldsymbol{W})\right] \tag{3.20}$$

is called expected self-change.

**Definition 3.2.2** (Cross-Change)**.**

$$\sum_{ab\neq ij} L^{(ab)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ab)}(\boldsymbol{W}) \tag{3.21}$$

is called cross-change and

$$\mathbb{E}_{ij} \sum_{ab\neq ij} L^{(ab)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ab)}(\boldsymbol{W}) \tag{3.22}$$

is called expected cross-change.

The main challenge is the cross-change, which represents the change in loss for every entry except $\widehat{X}_{ij}$. Indeed, the stochastic update is completely unaware of the impact on the loss of other entries than $\widehat{X}_{ij}$. Thus there is no guarantee that the cross-change nonpositive.

Our solution to this problem is the bound-and-scale step. In each iteration of SBSMU, the bound-and-scale step counteracts the impact of the cross-change. In order to show this theoretically and formulate an auxiliary function from which SBSMU can be derived, we define the bound-and-scale-divergence $H$.

**Definition 3.2.3** (Bound-and-Scale-Divergence)**.** Let $\psi_{\max}$ and $\psi_{\min}$ be the constants used in the stochastic majorization function $G^{(ij)}$. $H : \mathbb{R}^{n\times r}_{\geq 0} \times \mathbb{R}^{n\times r}_{\geq 0} \to \mathbb{R}$,

$$H(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) = \sum_{al} \left[ \frac{\left(\frac{\widetilde{W}_{al}}{W_{al}}\right)^{\psi_{\max}} - 1}{\psi_{\max}} - \frac{\left(\frac{\widetilde{W}_{al}}{W_{al}}\right)^{\psi_{\min}} - 1}{\psi_{\min}} \right] \tag{3.23}$$

is called the bound-and-scale-divergence.

***Lemma* 3.2.2.** The bound-and-scale-divergence defines a valid divergence between the factor matrix $\boldsymbol{W}$ and an updated factor matrix $\widetilde{\boldsymbol{W}}$, that is

$$H(\boldsymbol{W}, \boldsymbol{W}) = 0, \tag{3.24}$$

$$\forall\ \widetilde{\boldsymbol{W}} \neq \boldsymbol{W} : H(\boldsymbol{W}, \widetilde{\boldsymbol{W}}) > 0. \tag{3.25}$$

*Proof.* Consider the function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$,

$$f(x, \psi) = \frac{x^{\psi} - 1}{\psi}. \tag{3.26}$$

$f(x, \psi)$ is monotonically increasing over $\psi$ if $x > 0$ [51]. For $x = 1$, $f$ is constant with $f(1, \psi) = 0$ for all $\psi$. Since $\psi_{\max} > \psi_{\min}$, we get

$$f(x, \psi_{\max}) - f(x, \psi_{\min}) \geq 0. \tag{3.27}$$

The equality holds iff $x = 1$. Setting $x = \frac{\widetilde{W}_{al}}{W_{al}}$ and summing over the entries of the factor matrix gives $H$. □

### 3.2.3 Theorem and Proof

**Theorem 3.2.1** (Convergence of SBSMU)**.** If

$$\frac{\mathbb{E}_{ij} \sum_{ab \neq ij} \left[ L^{(ab)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ab)}(\boldsymbol{W}) \right]}{\mathbb{E}_{ij} H(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W})} \leq \frac{\alpha}{1 - \alpha}, \tag{3.28}$$

then

$$L(\boldsymbol{W}) \geq \mathbb{E}_{ij} L(\boldsymbol{W}^{ij,\text{new}}). \tag{3.29}$$

Because SBSMU is a stochastic algorithm, it is not guaranteed to monotonically decrease the loss in every iteration. However, like in the convergence proof for SGD [4], we can still prove converge in expectation. Specifically, we consider the expected value of the loss with respect to the entry we choose at a given iteration, $\mathbb{E}_{ij} L(\boldsymbol{W}^{\text{new}})$. Using the MM algorithm from Section 2.1.4, we show that $\mathbb{E}_{ij} L(\boldsymbol{W}^{\text{new}})$ is nonincreasing if the condition (3.28) holds. To do so, we define a function $F^{(ij)}$ and prove that it satisfies the properties required for an auxiliary function in expectation. This includes the majorization properties (P1) and (P2), as well as the minimization property (P3).

Let $F^{(ij)} : \mathbb{R}_{\geq 0}^{n \times r} \times \mathbb{R}_{\geq 0}^{n \times r} \to \mathbb{R}_{\geq 0}$ be defined as

$$F^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) = \frac{1}{2} \left[ G^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) + \frac{\alpha}{1 - \alpha} H(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) + \sum_{ab \neq ij} L^{(ab)}(\boldsymbol{W}) \right]. \tag{3.30}$$

***Lemma* 3.2.3** (P3)**.** $\mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) \geq \mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W})$.

*Proof.* Setting

$$\frac{\partial F^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} = 0 \tag{3.31}$$

gives the stochastic update rule[1]

$$W_{al}^{ij,\text{new}} = W_{al} \left( \frac{\alpha + (1-\alpha)\nabla_{ij,al}^{-}}{\alpha + (1-\alpha)\nabla_{ij,al}^{+}} \right)^{\frac{1}{\psi_{\max} - \psi_{\min}}}. \tag{3.32}$$

This is the update we apply in SBSMU. Because $\boldsymbol{W}^{ij,\text{new}}$ is the minimum of $F^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W})$ over $\widetilde{\boldsymbol{W}}$, we have

$$F^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) \geq F^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) \tag{3.33}$$

for any indices $i$ and $j$. Taking the expectation completes the lemma. □

***Lemma* 3.2.4** (P1)**.** $L(\boldsymbol{W}) = \mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}, \boldsymbol{W})$.

*Proof.* Lemma 3.2.1 shows that $G^{(ij)}$ majorizes $L^{(ij)}$, hence $G^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) = L^{(ij)}(\boldsymbol{W})$. Lemma 3.2.2 shows that $H$ is a valid divergence with $H(\boldsymbol{W}, \boldsymbol{W}) = 0$. Hence,

$$\mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) = \mathbb{E}_{ij} \frac{1}{2} \left[ L^{(ij)}(\boldsymbol{W}) + \sum_{ab \neq ij} L^{(ab)}(\boldsymbol{W}) \right] \tag{3.34}$$

$$= \frac{1}{2} \sum_{ab} L^{(ab)}(\boldsymbol{W}) \tag{3.35}$$

$$= L(\boldsymbol{W}), \tag{3.36}$$

where the last equality follows from the definition of $L^{(ij)}$ in Section 3.1.1. □

***Lemma* 3.2.5** (P2)**.** $\mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) \geq \mathbb{E}_{ij} L(\boldsymbol{W}^{ij,\text{new}})$.

*Proof.* From Lemma 3.2.2 and the assumption in (3.28), we have

$$\frac{\alpha}{1-\alpha} \mathbb{E}_{ij} H(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) \geq \mathbb{E}_{ij} \sum_{ab \neq ij} \left[ L^{(ab)}(\boldsymbol{W}^{ij,\text{new}}) - L^{(ab)}(\boldsymbol{W}) \right]. \tag{3.37}$$

[1] See Appendix B for the full derivation.

This gives

$$\begin{aligned}
&\mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) \\
&= \mathbb{E}_{ij} \frac{1}{2} \left[ G^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) + \frac{\alpha}{1-\alpha} H(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) + \sum_{ab \neq ij} L^{(ab)}(\boldsymbol{W}) \right] && (3.38) \\
&\geq \mathbb{E}_{ij} \frac{1}{2} \left[ L^{(ij)}(\boldsymbol{W}^{ij,\text{new}}) + \frac{\alpha}{1-\alpha} H(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) + \sum_{ab \neq ij} L^{(ab)}(\boldsymbol{W}) \right] && (3.39) \\
&\geq \mathbb{E}_{ij} \frac{1}{2} \left[ L^{(ij)}(\boldsymbol{W}^{ij,\text{new}}) + \sum_{ab \neq ij} L^{(ab)}(\boldsymbol{W}^{ij,\text{new}}) \right] && (3.40) \\
&= \mathbb{E}_{ij} L(\boldsymbol{W}^{ij,\text{new}}), && (3.41)
\end{aligned}$$

where (3.39) uses that $G^{(ij)}$ majorizes $L^{(ij)}$, proven in Lemma 3.2.1. (3.40) uses the inequality in (3.37), which comes from the convergence condition. □

*Proof of Theorem 3.2.1.* Lemma 3.2.4, Lemma 3.2.3 and Lemma 3.2.5 prove each of the properties of an auxiliary function under the condition (3.28). Thus, in expectation, $F^{(ij)}$ is a valid auxiliary function for the loss $L$. We get

$$L(\boldsymbol{W}) \overset{\text{(P1)}}{=} \mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}, \boldsymbol{W}) \overset{\text{(P3)}}{\geq} \mathbb{E}_{ij} F^{(ij)}(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W}) \overset{\text{(P2)}}{\geq} \mathbb{E}_{ij} L(\boldsymbol{W}^{ij,\text{new}}). \tag{3.42}$$

This implies that $L$ is nonincreasing in expectation and in turn SBSMU converges. □

### 3.2.4 Remarks

In this section we present some remarks on Theorem 3.2.1 First, we discuss the meaning and implications of the convergence condition. In one sense the condition represents a limitation of the proof. However, it reveals insights about how $\alpha$ should be tuned, under what condition SBSMU converges and why the bound-and-scale step is necessary to achieve convergence. Second, we explain how the proof can be relaxed to include different values for the exponential learning rate.

#### Remarks on the Convergence Condition

The meaning of the condition (3.28) can be illustrated by an analogy with velocity. The left side of the condition is the expected cross-change relative to the expected

change of the factor matrix $\boldsymbol{W}$, measured by the bound-and-scale-divergence. In the velocity analogy, the distance traveled is the expected cross-change and the time dimension is expressed in terms of change of the factor matrix. The condition then states that the expected cross-change "speed" is upper bounded by a constant $\frac{\alpha}{1-\alpha}$, depending on $\alpha$.

In general, the condition points out that higher value for the hyperparameter $\alpha$ are more likely to yield convergence. The margin between the smallest possible $\alpha$ and the chosen value leads to a decrease in convergence rate. Indirectly, the optimal value for $\alpha$ depends on the magnitude of the partial derivative components, which determine the change in the factor matrix. Note that the proof also supports the requirement $\alpha < 1$: for $\alpha = 1$, the condition is undefined, while even higher values give a negative upper bound.

When learning signals over different samples share some common pattern, the condition holds for a larger range of values for $\alpha$. That is, learning from $X_{ij}$ probably leads to a decrease in loss for the other entries as well. In turn, the expected cross-change relative to the change in $\boldsymbol{W}$ is smaller and therefore easier to upper bound. In cluster analysis, which we present as an application of NMF in Section 2.2, this is often the case. All entries in the same cluster attract the corresponding rows in $\boldsymbol{W}$ closer.

The theorem derivation procedure reveals the need for the augmentation. The minimization property (P3) guarantees the decrease of expected self-change, but learning from the current sample could result in an increase in cross-change. The augmentation term $\frac{\alpha}{1-\alpha} H(\boldsymbol{W}^{ij,\text{new}}, \boldsymbol{W})$, which introduces the bound-and-scale step to the update, remedies this. As a result, convergence can be guaranteed under certain conditions on the bound-and-scale parameter $\alpha$.

#### Relaxation of the Exponential Learning Rate

The current proof fixes the value of the learning rate parameter to $\eta = \frac{1}{\psi_{\max} - \psi_{\min}}$. While SBSMU with this learning rate is guaranteed to converge, different values can improve the rate of convergence in practice. The proof can be relaxed to include lower values for $\eta$ by altering $\psi_{\max}$ in the stochastic majorization function and the bound-and-scale divergence.

## 3.3 Implementation and Optimization

This section presents our implementation of SBSMU and explains how we optimized execution time. The optimizations we applied can be divided into four categories:

- **Algorithmic modifications.** While the optimized implementation is log-

ically equivalent with Algorithm 6, we make certain changes to improve performance on sparse matrices and reduce the cost of random number generation.

- **Stratified sampling.** When factorizing sparse matrices, it is desirable to increase the sampling rate for nonzero entries compared to zero-valued entries.
- **Compilation.** The execution of compiled code is significantly faster than using an interpreter.
- **Parallelization.** Parallelizing code makes it possible to utilize multi-core architectures.

First, Section 3.3.1 presents how we treat the concept of convergence and determine termination criteria. Each of the remaining sections discuss one of the optimizations described above. Section 3.3.2 presents an overview of algorithmic modifications. Section 3.3.3 shows how we use an adapted version of stratified sampling to increase utilization of the nonzero entries of $\boldsymbol{X}$. Section 3.3.4 introduces Numba, which we use to compile the code instead of running it through the Python interpreter. Finally, Section 3.3.5 explains how we utilize parallelization to enhance performance on architectures with multiple processors.

### 3.3.1 Convergence Criteria

In Algorithm 6, the update loop continues until an unspecified *convergence criterion* is reached. The nature of this convergence criterion depends on the application. Broadly speaking, there are two approaches: run until convergence is detected or run until some computational budget limit is reached. These two approaches might also be combined, so that the algorithm terminates when either criterion is satisfied.

While detecting convergence is straightforward for full-batch MU, this is not the case for stochastic algorithms. The advantage of full-batch MU is that they have the monotone convergence property. The loss is always decreasing, so it is possible to run the algorithm until the decrease in loss between two iterations is sufficiently small. The challenge with stochastic algorithms is that they usually are not guaranteed to monotonically decrease the loss. Even if the expected value of the loss is nonincreasing, there may still be temporary increases in the loss. This is because different entries in the dataset can pull the values in the factor matrix in different directions. A step that reduces the loss with respect to one entry in $\boldsymbol{X}$ may do the opposite for others, and thus increase the total loss. Hence, an approach based on measuring the difference in loss between successive iterations does not work. One alternative is to keep track of the lowest loss so

far, and terminate when a sufficiently long period of time goes by without an improvement.

Computational budget criteria are applicable to both full-batch and stochastic algorithms, and the budgets can be specified in terms of the number of iterations or the running time. The latter has the advantage that it allows fair comparisons between different algorithms. Furthermore, budgets based on running time give the user more control when the time per iteration is unknown.

### 3.3.2 Algorithmic Optimization

Algorithmic optimizations are modifications of the algorithm that improve performance, while keeping the algorithm logically equivalent to the non-optimized version. The optimizations yield a more time efficient execution, for example by avoiding calculation of terms that are later multiplied by zero. In the following we present three algorithmic optimizations. The first two modifications are always valid when factorizing sparse, symmetric matrices and have been used in our implementation. The third modification is specific to NMF applied to clustering, and is not used in this work.

#### Generating Lists of Random Indices

When generating random indices, we improve performance by generating lists of random numbers instead of making calls to the random number generator every iteration. Logically, we need to choose two indices $i$ and $j$ for each iteration. These determine the entry $X_{ij}$ that the update derivation is based on. Instead of performing random number generation separately at the beginning of each iteration, two index lists of size 10 000 are randomly generated and stored every 10 000 iterations. At the beginning of each iteration, the current heads of the two lists are consumed as index $i$ and $j$.

This approach optimizes execution time by minimizing the overhead of random number generation. Due to fixed costs like the initialization of the random generator, the cost per random number is lower when generating 10 000 numbers than when generating a single number. The resulting speedup is greater than the time it takes to store and fetch random numbers. One potential disadvantage is that storing lists of random indices requires extra memory. However, the amount of memory required to store 20 000 integers is negligible compared to the size of the datasets SBSMU is applied to.

#### Utilizing the Sparsity of $\boldsymbol{X}$

Our focus is on factorizing sparse datasets with the I-divergence, and so we consider how SBSMU can be optimized for this specific problem. Since sparse

matrices are dominated by zero-valued entries, simplifying the update when $X_{ij} = 0$ may significantly speed up execution. We derive the stochastic multiplicative update for I-divergence in Section 3.1.2. Its negative component is

$$\nabla_{ik}^{-} = 2\frac{W_{jk}X_{ij}}{\widehat{X}_{ij}}. \tag{3.43}$$

If $X_{ij} = 0$, then $\nabla_{ik}^{-} = 0$ for all $k$. Hence, we do not explicitly calculate this term for the zero entries of $\boldsymbol{X}$. Importantly, this means that we avoid the relatively expensive computation of $\widehat{X}_{ij} = \sum_k W_{ik}W_{jk}$.

#### Omitting Diagonal Entries

In clustering applications, the diagonal entries of the dataset do not need to be included. As discussed in Section 2.2, clustering is the most important application of symmetric NMF. In clustering, $\boldsymbol{X}$ is a similarity matrix where $X_{ij}$ represents the similarity between two entities $i$ and $j$. The diagonal entries thus represent the similarity of each entity with itself. This information is not useful for clustering, and can be left out of the factorization process.

Not including diagonal entries is straightforward and likely to improve performance. To avoid diagonal entries, we can simply enforce $i \neq j$ when randomly selecting indices. By removing information that is not useful, it is likely that the convergence rate will increase. Furthermore, the quality of the factorization may improve because it is not constrained by the diagonal entries.

### 3.3.3 Stratified Sampling

Stratified sampling is a statistical technique for reducing estimation errors in populations with mutually exclusive subgroups. The technique involves allocating the total number of samples among the subgroups, and then sampling each subgroup independently. More samples are allocated to larger subgroups and subgroups with greater internal variability. This ensures that each subgroup is adequately represented. If the subgroups are relatively homogeneous compared to the population as a whole, this will often reduce the overall estimation error [48].

Stratified sampling can be applied to the factorization of sparse datasets, where nonzero elements and zero-valued elements constitute two subgroups. The zero-valued elements outnumber the nonzero elements, but the nonzero elements contain most of the information. Hence, it makes sense to choose a distribution that prioritizes nonzero elements, rather than choosing elements uniformly at random. We achieve this by introducing a sampling variable $\beta \in (0, 1)$, which represents the probability of selecting a nonzero element each iteration.

It is necessary to ensure that we still optimize the correct loss function. Since we apply stratified sampling to sparse matrices, we show how the sampling variable can be introduced to the I-divergence here. However, the same principles can be applied to the Euclidean distance. To ensure that we are optimizing for the I-divergence, we require that $\boldsymbol{X}$ is normalized so that $\sum_{ij} X_{ij} = 1$. This allows us to view $\boldsymbol{X}$ as a probability distribution, which gives

$$L_I(\boldsymbol{W}) = \sum_{ij} \widehat{X}_{ij} - \sum_{ij} X_{ij} \ln \widehat{X}_{ij} + \text{constant} \tag{3.44}$$

$$= N^2 \mathbb{E}_{(i,j)\sim[1..N]^2} \left[ \widehat{X}_{ij} \right] - \mathbb{E}_{(i,j)\sim\boldsymbol{X}} \left[ \ln \widehat{X}_{ij} \right] + \text{constant} \tag{3.45}$$

$$= \frac{1}{\beta} \Bigg( (1-\beta) \frac{\beta}{1-\beta} N^2 \mathbb{E}_{(i,j)\sim[1..N]^2} \left[ \widehat{X}_{ij} \right] - \beta \mathbb{E}_{(i,j)\sim\boldsymbol{X}} \left[ \ln \widehat{X}_{ij} \right] \Bigg) + \text{constant}. \tag{3.46}$$

Optimizing for (3.46) is equivalent to optimizing for the I-divergence, but it allows us to set the sampling rate for nonzero entries using $\beta$. The term $\mathbb{E}_{(i,j)\sim\boldsymbol{X}}[\ln \widehat{X}_{ij}]$ contains only nonzero entries that are sampled according to $\boldsymbol{X}$. The $\beta$ factor is applied implicitly through the sampling rate. The term $\mathbb{E}_{(i,j)\sim[1..N]^2}[\widehat{X}_{ij}]$ includes both nonzero and zero-valued entries, and contains no information from $\boldsymbol{X}$. The $(1-\beta)$ factor is applied implicitly through sampling, but this term must be explicitly multiplied by $\frac{\beta}{1-\beta} N^2$. The leftmost $\frac{1}{\beta}$ factor outside the parentheses is applied to both terms. It can be safely ignored since scaling the loss will not alter the location of the minimum.

The approach differs from how stratified sampling usually works, but update rules based on (3.46) can enforce a higher sampling rate for the nonzero terms. The two terms in (3.46) do not exactly represent the two subgroups, nonzero and zero-valued entries. While $\mathbb{E}_{(i,j)\sim\boldsymbol{X}}[\ln \widehat{X}_{ij}]$ only contains nonzero entries, $\mathbb{E}_{(i,j)\sim[1..N]^2}[\widehat{X}_{ij}]$ contains a uniform distribution over all the entries. However, in a sparse matrix with mostly zero-valued entries, the latter subgroup is very similar to the group of all zero-valued entries. Regardless of sparsity, increasing the sampling rate for the first subgroup increases the sampling rate for nonzero entries relative to zero-valued entries. Hence, although this technique is different from stratified sampling in the standard sense, it achieves the same goal.

The update rules are derived from the stochastic partial derivative of the loss function with respect to a single entry of $\boldsymbol{W}$. Following the procedure in Section 3.1.2, we use the symmetry of $\boldsymbol{X}$ to include both $X_{ij}$ and $X_{ji}$ by multiplying by 2. Since the two terms in (3.46) are distributed differently, we derive two separate

update rules[2]. For the positive term,

$$\nabla^{+}_{ij,al} = \frac{\partial}{\partial W_{al}} \left( 2N^2 \frac{\beta}{1-\beta} \widehat{X}_{ij} \right) \tag{3.47}$$

$$= 2N^2 \frac{\beta}{1-\beta} \frac{\partial}{\partial W_{al}} \left( W_{il} W_{jl} \right), \tag{3.48}$$

which gives the update

$$W_{al} \leftarrow W_{al} \left( \frac{\alpha}{\alpha + (1-\alpha)\nabla^{+}_{ij,al}} \right). \tag{3.49}$$

For the negative term,

$$\nabla^{-}_{ij,al} = \frac{\partial}{\partial W_{al}} \left( \ln \widehat{X}_{ij} \right) \tag{3.50}$$

$$= \frac{1}{W_{il} W_{jl}} \frac{\partial}{\partial W_{al}} \left( W_{il} W_{jl} \right), \tag{3.51}$$

which gives the update

$$W_{al} \leftarrow W_{al} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,al}}{\alpha} \right). \tag{3.52}$$

It is worth noting that since we redefine the loss function to include stratified sampling, the formulas of $\nabla^{+}_{ij,al}$ and $\nabla^{-}_{ij,al}$ change. The resulting pseudocode is shown in Algorithm 7.

This approach to stratified sampling changes how we apply the algorithmic optimizations presented in the previous section. It is still beneficial to generate lists of random numbers instead of calling the random number generator every iteration. However, five separate lists for $\beta$ values, indices sampled uniformly at random, and indices sampled according to $\boldsymbol{X}$. The technique for utilization of sparsity is applied implicitly, since $\nabla^{-}_{ij,al}$ only is calculated for nonzero entries.

### 3.3.4 Compilation

Our implementation of SBSMU is written in Python, which we also used to experiment with different versions of the algorithm. Extensive third-party mathematics libraries like NumPy [21] and SciPy [49] make Python a common choice for scientific computing software. Furthermore, with dynamic typing, garbage collection

[2]For brevity, we omit the four cases introduced by the term $\frac{\partial}{\partial W_{al}} \left( W_{il} W_{jl} \right)$.

**Algorithm 7:** SBSMU with Stratified Sampling for the I-divergence

**Input :** $\boldsymbol{X}$, $\eta$, $\alpha$, $\beta$
**Output:** $\boldsymbol{W}$
$\boldsymbol{W} \leftarrow$ `RandomNonnegativeMatrix()`;
**while** $\neg$*(convergence condition)* **do**
  $b \sim U[0,1]$;
  **if** $b < \beta$ **then**
    $(i,j) \sim U\left[1,...,N\right]^2$;
    **for** $k$ **in** $1 \dots r$ **do**

$$W_{ik} \leftarrow W_{ik} \left( \frac{\alpha}{\alpha + (1-\alpha)\nabla^{+}_{ij,ik}} \right)^{\eta};$$

$$W_{jk} \leftarrow W_{jk} \left( \frac{\alpha}{\alpha + (1-\alpha)\nabla^{+}_{ij,jk}} \right)^{\eta};$$

  **else**
    $(i,j) \sim \boldsymbol{X}$;
    **for** $k$ **in** $1 \dots r$ **do**

$$W_{ik} \leftarrow W_{ik} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,ik}}{\alpha} \right)^{\eta};$$

$$W_{jk} \leftarrow W_{jk} \left( \frac{\alpha + (1-\alpha)\nabla^{-}_{ij,jk}}{\alpha} \right)^{\eta};$$

and emphasis on readability, Python facilitates high development velocity and is a suitable language for prototyping. However, this comes at a significant performance cost. A study comparing the performance of Python, Go and C++ on the N Queens problem, found that the C++ implementation was 1-2 orders of magnitude faster than the Python implementation [17]. Nevertheless, as long as the main bottleneck is development time, Python is a good choice for the reasons stated above. If the goal is to optimize performance however, other tools are needed.

Numba is a third-party compiler that translates Python code into machine code to increase performance. The authors of Numba claim that it can achieve execution times comparable to those of C or C++ in scientific computing domains [27]. This is supported by the previously cited study on the N Queens problem, where Numba was found to perform about 10% worse than the C++ implementation [17]. Numba is based on the LLVM compiler library, which provides the low-level programming language *Intermediate Representation* (IR). LLVM

takes IR code as input, optimizes it and converts it to platform-dependent machine code [28]. Numba is able to parse most of the basic Python language and convert it to IR code. In addition, it supports most of the functionality provided by NumPy. Applying Numba to a function written in Python is as simple as adding a decorator, provided that it only contains code that can be parsed by Numba. Hence, Numba can greatly increase the performance of Python code with minimal effort.

By applying Numba to our Python implementation, we were able to reduce execution time by approximately two orders of magnitude without making significant changes to the code.

### 3.3.5 Parallelization

Parallelizing code may greatly speed up its execution, but sometimes these gains are more than offset by the associated overhead. Parallelization means writing code that can be run on multiple processors in parallel. This can be utilized on systems with multiple processing units. However, spawning new processes and synchronizing their execution may lead to significant overhead. Synchronization is usually required to avoid concurrency issues when handling shared, mutable objects. It entails one process waiting on another process to avoid two processes accessing the shared region of memory at the same time. Aside from the wait itself, this necessitates inter-process communication that also adds extra cost. In particular, synchronization is often implemented using locks that a process must acquire when using the shared object and then release.

*Hogwild* is an update scheme that allows parallelized SGD to be implemented without any object locks [41]. Stochastic optimization algorithms like SGD use fast update loops that may only take nanoseconds to calculate. The updates require reading from and writing to memory, but those operations can also be completed in a matter of tens of nanoseconds on modern systems. This means that locking can become a significant bottleneck. Hogwild avoids this by giving the processes unhindered access to the shared memory. Each process updates the components without any synchronization. This will inevitably lead to some incorrect updates. However, the errors introduced by such updates are minor, and they are rare when data access is sparse. Sparse data access means that each update only affects a small part of the shared memory. The authors behind Hogwild show that a linear speedup in the number of processors can be theoretically guaranteed as long as data access is sufficiently sparse. Furthermore, they empirically demonstrate that Hogwild outperforms parallelization schemes for SGD that require locking.

The Hogwild scheme can be applied to parallelize SBSMU without locking. A single update of SBSMU affects at most 2 rows of $\boldsymbol{W}$, which means that data

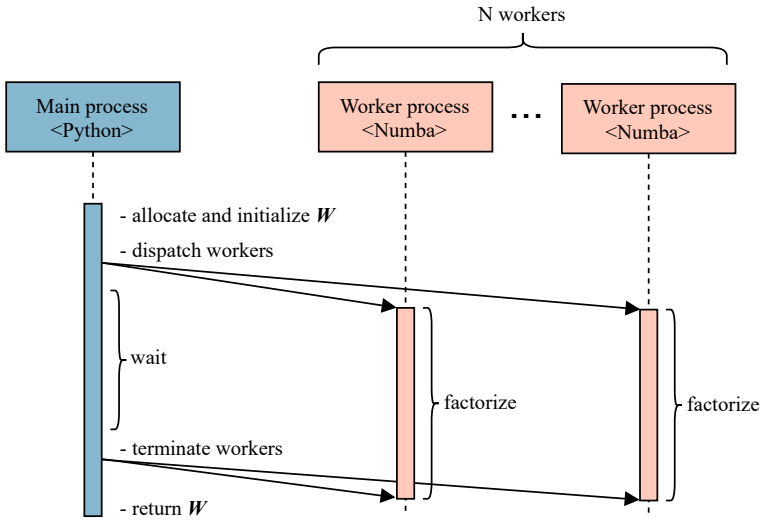



Figure 3.1: Parallelization architecture in SBSMU

access is extremely sparse. The loop in SBSMU can thus be run independently on several cores, which are unaware of each other aside from sharing access to the same factor matrix $\boldsymbol{W}$ in memory. It does not matter in the long run if two processes sometimes choose the same entry $X_{ij}$ at the same time. This may cause one process to update $\boldsymbol{W}$ incorrectly by using rows of $\boldsymbol{W}$ from different time steps. However, as long as the number of entries is far greater than the number of processors, this will be a highly infrequent event. The erroneous updates are drowned out by correct updates, and also become less significant as $\widehat{\boldsymbol{X}}$ converges.

Because we require very little communication between the worker processes, we can implement parallelization using a simple fork-join design pattern [36]. The algorithm first allocates memory for and randomly initializes $\boldsymbol{W}$, then branches out by dispatching multiple workers that apply the update loop in SBSMU independently. The main process keeps track of the convergence criterion and signals to the workers when they should terminate. This is illustrated in figure 3.1.

# 4 Experiments

This section presents the empirical experiments we conducted to assess the performance of SBSMU, including the setup, methodology and the results. Our overarching approach was to test SBSMU on sparse datasets from the real world, with the I-divergence as the loss function. Section 4.1 presents the setup, including hardware, software, random number generation and datasets. The remaining sections contain method and results of the experiments. Section 4.2 presents the hyperparameter experiment, where we explore how SBSMU performs with different configurations by performing grid searches on several datasets. Section 4.3 presents the benchmark experiment, which compares the performance of SBSMU and three other matrix factorization algorithms on the MNIST dataset. Finally, Section 4.4 presents the big data experiment, which compares the performance of SBSMU and full-batch MU on the Higgs dataset. Due to the size of the dataset, running four benchmark algorithms was infeasible, so we prioritized MU.

## 4.1 Setup and Datasets

This section details the setup for the experiments in order to facilitate reproducibility. By using this setup, the datasets we list here, and our source code[1], a researcher should be able to reproduce all the results presented in this work.

#### Hardware and Software

We ran some of our experiments on a local desktop computer, while the most resource-intensive tasks were run on the remote cluster system IDUN. The desktop computer is a Dell OptiPlex 9020 that runs Windows 10, has 16GB of memory and has a 3.40GHz Intel Core i7-4770 processor with 8 threads. The IDUN cluster is maintained by the High Performance Computing Group at NTNU and contains multiple Intel Xeon processors [44]. We used the Python 3 interpreter and libraries that come bundled with Anaconda 2020.07 on the cluster and Anaconda 2020.11 on the desktop computer [1].

[1] The source code is available for download at `https://github.com/schlueter-riege/sbsmu`.

### Random Number Generation

The seeds we used to generate random numbers in our experiments are listed in Appendix C. The implementations of SBSMU and the benchmark algorithms take the seed as an optional argument, and use the default random number generators provided by Numpy and Numba. This ensures that the initialization of $\boldsymbol{W}$ and the indices chosen by each worker are reproducible. Furthermore, all our optimization algorithms have the initialization of $\boldsymbol{W}$ as the first call to the random number generator. They also use the same initialization method, choosing the entries of $\boldsymbol{W}$ randomly from a uniform distribution over $(0, 1]$. This ensures that two algorithms with the same seed will start with the same randomized $\boldsymbol{W}$. Nevertheless, due to parallelism it is not possible to guarantee the order of the updates in SBSMU. Furthermore, we ran the experiments with a time limit. This means that there were small fluctuations in the number of iterations between runs. Hence, there will always be some variation between experiments with the same algorithm and the same seed.

### Convergence Criteria

Ideally, we would be able to let each algorithm run until it converges. However, as discussed in Section 3.3.1, the concept of convergence is problematic when working with stochastic algorithms. Because they may never settle completely in a local minimum, it is not possible to detect convergence from the change in loss in consecutive iterations. Furthermore, limited computational resources prevented us from running slow-converging algorithms until they stabilized completely. For these reasons we opted to use computational budgets based on execution time, and we only consider the minimum loss and rate of convergence within the time limit.

Additionally, for some experiments we detected convergence by considering the time since the last recorded improvement in minimum loss. If a specified time interval went by without any improvement in loss, the algorithm was terminated. This ensured that diverging configurations were stopped early, thus saving computational resources. This was used in combination with a time limit.

### Datasets and Preprocessing

The datasets used in this work are listed in Table 4.1. We included data from a variety of sources in order to test the performance of SBSMU in many different domains. All the datasets are publicly available in their original form through the provided sources.

We applied some preprocessing steps in order to make the datasets symmetric and sparse. The network datasets, Dolphins and Football, consist of symmetric

Table 4.1: Datasets used in experiments

| Dataset | Size | Rank | Domain | Source |
|---|---|---|---|---|
| Higgs | 11 000 000 | 5 | Physics | [3] |
| MNIST | 70 000 | 10 | Handwriting images | [10] |
| 20Newsgroups | 19 938 | 20 | Text documents | [38] |
| Gisette | 7 000 | 2 | Handwriting images | [29] |
| CUReT | 5 612 | 61 | Texture images | [9] |
| Wine | 178 | 3 | Chemistry | [16] |
| Iris | 150 | 3 | Biology | [15] |
| Football | 115 | 12 | Sports league network | [18] |
| Dolphins | 62 | 2 | Social network | [33] |
| AML/ALL | 38 | 3 | Gene expression | [19] |

adjacency matrices. Both are also naturally sparse. For the remaining datasets, the *k-nearest neighbor algorithm* ($k$-NN) was applied to generate sparse adjacency matrices. These adjacency matrices represent directed graphs, and to enforce symmetry we calculated the matrix to be factorized as

$$\boldsymbol{X} \leftarrow \boldsymbol{X} + \boldsymbol{X}^\top.$$

We scaled all the datasets, including the networks, so that $\sum_{ij} X_{ij} = 1$. For more information about the preprocessing, see table D.3 in Appendix D. The processed datasets are available on request.

We used the I-divergence in all experiments. As mentioned in Section 2.1.2, the I-divergence is often the loss function of choice when factorizing sparse matrices. We focused exclusively on sparse matrices in our experiments, and so we used the I-divergence throughout. Hence, when we discuss loss in the experiments, we are always referring to the I-divergence.

## 4.2 Hyperparameter Experiment

The aim of this experiment was to determine how sensitive SBSMU is to different hyperparameters, and in particular whether there exists a configuration that performs well across all datasets. To do so, we performed a grid search over the bound-and-scale variable $\alpha$, the sampling variable $\beta$ and the learning rate $\eta$. To some extent this relates to RQ2.2, since we investigated whether SBSMU converges in various settings. However, the main focus was RQ2.1, as we evaluated how performance varies with different hyperparameters and datasets.

Table 4.2: Parameter values tested in the hyperparameter experiment.

| Parameter | Domain | Values tested |
|---|---|---|
| $\alpha$ | $[0, 1)$ | [0.9, 0.99, 0.999, 0.9999, 0.99999] |
| $\beta$ | $(0, 1)$ | [0.1, 0.3, 0.5, 0.7, 0.9] |
| $\eta$ | $(0, 1]$ | [0.1, 0.3, 0.5, 0.7, 0.9] |

### 4.2.1 Setup and Method

To perform a grid search it is necessary to determine the ranges of the hyperparameters, the desired granularity and how the values should be distributed. As previously mentioned, $\alpha \in [0, 1)$ controls the bound-and-scale step, $\beta \in (0, 1)$ the determines the sampling rate for nonzero entries and $\eta \in (0, 1]$ is the exponential learning rate. The domains of all three variables are thus constrained to the real numbers between 0 and 1, with some variation in whether the endpoints are included. Running a grid search is computationally intensive, so we limited our search to 5 values per parameter. This gave a total of $5 \times 5 \times 5 = 125$ configurations to test. For $\alpha$, preliminary experiments indicated that values very close to 1 gave the best results. The proof also suggested that high values of $\alpha$ could be needed to achieve convergence. Hence, we chose a distribution of values that was skewed towards the upper limit of its domain. For $\beta$ and $\eta$, preliminary experiments did not indicate a strong tendency towards either end of their domains. For these variables we thus chose values that were evenly distributed between 0 and 1. The specific values we tested for each parameter are listed in Table 4.2.

In order to assess the robustness of SBSMU in a variety of scenarios, we ran the grid search on several datasets from a variety of domains. Specifically, we used all datasets from Table 4.1 except for MNIST and Higgs. In addition to variation in domain, these datasets gave us significant variation in size and rank. The smallest dataset was AML/ALL with dimensions $38 \times 38$, while the largest was 20Newsgroups at $19\,938 \times 19\,938$. Dolphins and Gisette both have a rank of 2, while CUReT has a rank of 62.

We performed 10 trials with different random seeds per configuration, for a total of 1250 SBSMU runs per dataset. Aside from the configuration, the outcome of a run depends on the random initialization and the order in which the indices are chosen. We performed multiple runs in order to average out the effect of these random factors. Furthermore, doing so allows us to calculate the standard deviation of the minimum loss across runs with different random seeds. Hence we can estimate how sensitive SBSMU is to the random factors.

We ran SBSMU with a time limit of 10 minutes per optimization run, while calculating the loss every 0.1 seconds. Furthermore, we terminated the run if 5 seconds went by without an improvement in minimum loss. These parameters

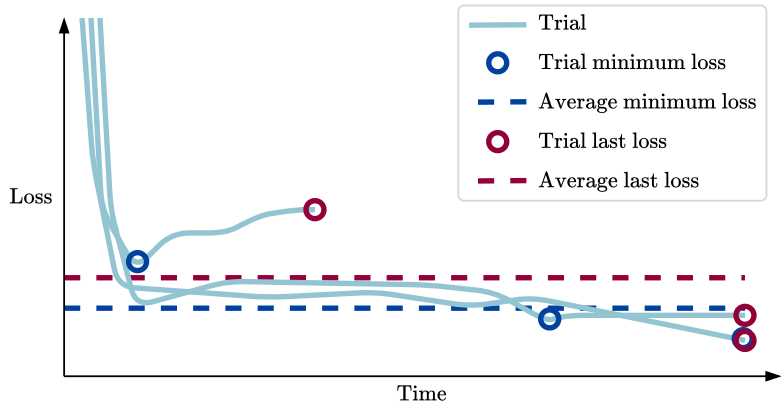



Figure 4.1: Illustration of average minimum loss and average last loss in a fictitious experiment with 3 trials. One trial diverges and is terminated early.

were chosen based on preliminary experiments.

We considered both the average minimum loss and the average last loss for each configuration. The average minimum loss was calculated by finding the minimum loss for each trial, and then taking the average over the trials. The average last loss was calculated by finding the last recorded loss for each trial, and then taking the average over the trials. The minimum loss is the most important of the two, as it represents the best factorization found. However, the last loss provides some information about the stability of the configuration. If the last loss is significantly higher than the minimum loss, it suggests that the configuration diverges in the long run. For clarity, the concepts of average minimum loss and average last loss are illustrated in Figure 4.1.

The experiment was run on the IDUN cluster due to its computational requirements, with 24 GB of memory allocated to each run. With 1250 runs per dataset, 8 datasets and a time limit of 10 minutes per run, it would have taken too long to execute all runs sequentially on a single computer. Through the cluster, we were able to run the experiment in parallel on 30 computers simultaneously. This did not interfere with the parallelization of SBSMU, as described in Section 3.3.5. Thus each instance of SBSMU was itself parallelized, running on multiple cores within the assigned computer.

The cluster is less of a controlled environment compared to the desktop computer. Although all the CPUs on the cluster are from the Intel Xeon series, we do not know the exact types of the processors the experiment ran on. Furthermore, the trials were not run in exclusive mode on the nodes on the cluster. That means that our processes may have had to share certain resources with other

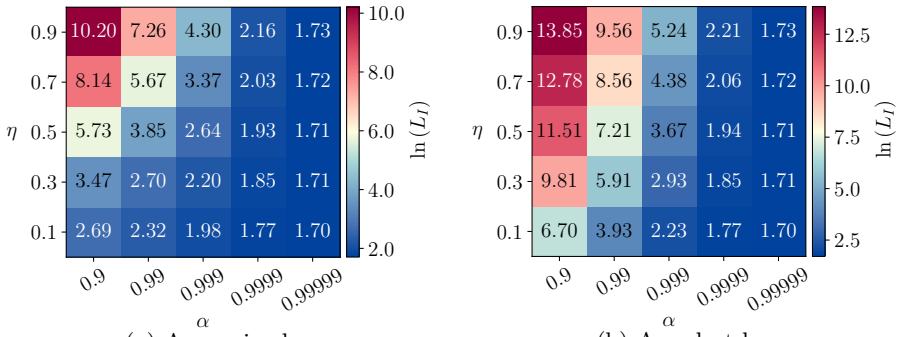
jobs. However, we considered this acceptable, since the evaluation of the results from this experiment does not depend on exact time measurements.

### 4.2.2 Results

The results from the hyperparameter experiment are summarized in Table 4.3. The table shows the best configuration found for each dataset, as determined by the average minimum loss. It also shows the associated average minimum loss and average last loss, including standard deviations. From the table, it is clear that the average last loss is near the average minimum loss for the best configurations. For all the datasets, the last loss is less than 1 standard deviation away from the minimum loss. Furthermore, it is clear that the best configurations all have relatively high values of $\alpha$. For all datasets, the top configuration has one of the two highest values tested in the hyperparameter search. Beyond this, there is no clear tendency for $\beta$ and $\eta$. While SBSMU achieves the lowest loss on 20Newsgroups with the lowest values of both $\beta$ and $\eta$, the optimum for Dolphins and Football is near the highest parameter values. For CUReT the best configuration has a low value of $\beta$ and a high value of $\eta$, while the exact opposite is true for Iris.

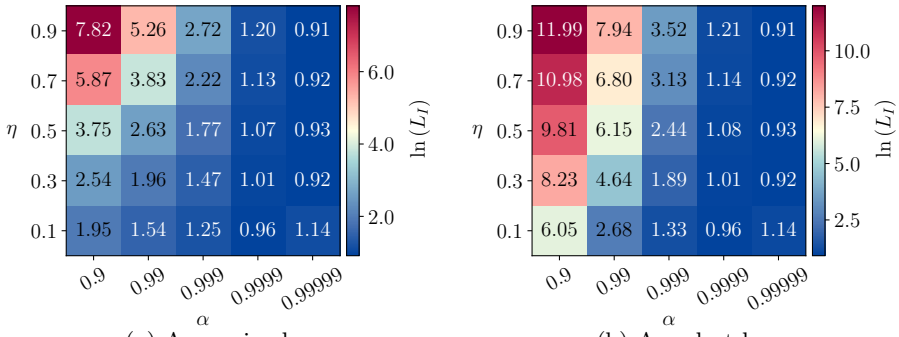
Figure 4.2, 4.3 and 4.4 show heat maps of $\alpha$ and $\eta$ for a given $\beta$. Each cell contains the natural logarithm of the average minimum loss for that configuration. The cells are color coded so that red represents relatively high average minimum loss, while blue represents a relatively low average minimum loss. These particular plots were chosen because they illustrate the most important trends we wish to highlight. They contain the best configuration found for three of the datasets, 20Newsgroups, CUReT and Dolphins. Similar plots for all other configurations tested can be found in Appendix F.

All plots indicate that higher values of $\alpha$ reduce both the average minimum loss and the average last loss. Furthermore, it seems that the value of $\eta$ becomes less important for higher values of $\alpha$. Consider the plot of the average minimum losses for 20Newsgroups in Figure 4.2. Across the values of $\alpha$, decreasing $\eta$ leads to lower minimum losses. For $\alpha = 0.9$ for example, decreasing $\eta$ from 0.9 to 0.1 reduces the average minimum loss by 99.9%. For $\alpha = 0.99999$, however, the reduction is only 3.0%. Note that these percentages are calculated based on the average minimum loss, not the logarithm of the average minimum loss which is shown in the plot.



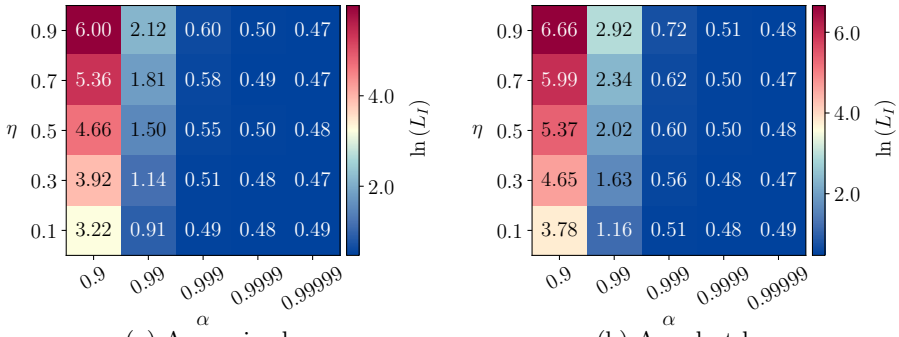

(a) Avg. min. loss (b) Avg. last loss

Figure 4.2: Hyperparameter experiment results for 20Newsgroups with $\beta = 0.1$.




(a) Avg. min. loss (b) Avg. last loss

Figure 4.3: Hyperparameter experiment results for CUReT with $\beta = 0.1$.




(a) Avg. min. loss (b) Avg. last loss

Figure 4.4: Hyperparameter experiment results for Dolphins with $\beta = 0.7$.

Table 4.3: Best configuration, determined by average minimum loss, for each dataset from the hyperparameter experiment.

| **Dataset** | **Best configuration** $(\alpha, \beta, \eta)$ | **Avg. min. loss** | **Avg. last loss** |
|---|---|---|---|
| 20Newsgroups | $(0.99999, 0.1, 0.1)$ | $5.497 \pm 0.005$ | $5.497 \pm 0.005$ |
| AML/ALL | $(0.9999, 0.1, 0.5)$ | $0.802 \pm 0.007$ | $0.804 \pm 0.007$ |
| CUReT | $(0.99999, 0.1, 0.9)$ | $2.490 \pm 0.023$ | $2.491 \pm 0.023$ |
| Dolphins | $(0.99999, 0.7, 0.7)$ | $1.602 \pm 0.007$ | $1.603 \pm 0.007$ |
| Football | $(0.99999, 0.7, 0.9)$ | $0.842 \pm 0.034$ | $0.844 \pm 0.034$ |
| Gisette | $(0.99999, 0.3, 0.1)$ | $5.406 \pm 0.032$ | $5.407 \pm 0.032$ |
| Iris | $(0.9999, 0.7, 0.1)$ | $2.139 \pm 0.046$ | $2.143 \pm 0.047$ |
| Wine | $(0.99999, 0.5, 0.3)$ | $2.383 \pm 0.032$ | $2.384 \pm 0.032$ |

## 4.3 Benchmark Experiment

In this experiment, we compare the performance of SBSMU and benchmark algorithms on MNIST. One important benchmark is the performance of full-batch MU. In addition, we consider exponentiated gradient descent (EGD) and projected stochastic gradient descent (PSGD). EGD is an optimization algorithm that naturally maintains nonnegativity since the exponentiated gradient is always positive. PSGD is an extension of SGD that maintains nonnegativity by using a projection step after each iteration. For more details on EGD and PSGD, see Appendix E.

### 4.3.1 Setup and Method

When comparing SBSMU to benchmarks, one important question is to what extent the benchmark algorithms should be optimized. With the aim of providing a fair comparison, we implemented MU, EGD and PSGD with the same libraries we used for SBSMU. Their core loops are compiled using Numba, but unlike SBSMU we did not attempt to parallelize their execution.

Another important question is what configuration to use for each algorithm. For SBSMU, we performed a two-step hyperparameter search on MNIST. Firstly, we explored combinations of a broad range of settings in a coarse search. Secondly we ran a fine-grained search, exploiting the data from the first step to investigate the most promising areas. The best configuration we found, measured in average minimum loss, was $\alpha = 0.999999, \beta = 0.25, \eta = 0.8$. For the benchmark algorithms, we chose hyperparameter configurations either from the literature or from preliminary experimentation. For MU we used an exponential learning rate of $\eta = 0.5$, which was derived using the unified development procedure from Section 2.3.3. For EGD, $\omega = 0.1$ was determined to work well through preliminary experiments. For PSGD, we achieved the best results with a learning rate of $\eta = 10^{-5}$.

The benchmark experiment was run on the Dell desktop computer. For each algorithm, 5 trials were run with different random seeds to allow us to assess variance and average out the effect of random initialization. Each trial was run with a time limit of 10 minutes. Unlike the hyperparameter experiment, we did not calculate losses during runs and did not terminate any runs before the time limit. Instead, temporary copies of $\boldsymbol{W}$ were saved once per second and processed after the optimization was complete. The time it took to initialize the algorithms was not taken into account. This includes compilation of functions using Numba, initialization of $\boldsymbol{W}$ and spawning of worker processes. Since SBSMU uses multiprocessing, it takes 3-8 seconds to initialize it.

Besides minimum loss, the other key criterion is convergence time. This

Table 4.4: Average minimum loss and standard deviation for SBSMU and the benchmark algorithms applied to MNIST.

| **Algorithm** | **Minimum loss** |
|---|---|
| SBSMU | $(6.483 \pm 0.015) \times 10^0$ |
| EGD | $(6.398 \pm 0.024) \times 10^0$ |
| MU | $(6.424 \pm 0.021) \times 10^0$ |
| PSGD | $(6.437 \pm 0.089) \times 10^9$ |

is somewhat tricky when comparing algorithms that achieve different minimum losses. As discussed in Section 3.3.1, convergence generally represents the point at which an algorithm no longer significantly reduces the loss. Consider an algorithm that quickly levels off at a high loss, compared to one that takes longer to converge, but does so at a substantially lower loss. It is too simplistic to conclude that the former performs best in terms of convergence time. We cannot completely separate the concept of minimum loss from the concept of convergence rate when evaluating performance. Hence, we measure convergence time with respect to a loss threshold. The algorithm that most quickly reaches the threshold, is deemed to perform the best in terms of convergence time. This way we can judge algorithms based on how quickly they reduce the loss, without penalizing algorithms that achieve lower minimum losses.

### 4.3.2 Results

Like in the hyperparameter experiment, a key metric for each algorithm is the minimum loss. The minimum losses and standard deviations for each algorithm are shown in Table 4.4. The numbers in the table were calculated by first finding the minimum loss for each of the 5 trials, and then computing the average and standard deviation over these values.

Additionally, the average I-divergence is plotted against time for each algorithm in Figure 4.5 and 4.6. However, in Figure 4.6 PSGD has been excluded since the losses reached by PSGD are several orders of magnitude higher than those achieved by the other algorithms. The y-axis in this plot was clipped in order to show how the remaining algorithms perform near convergence.

It is worth noting that Table 4.4 shows the average minimum across the trials, while the minimum of the plots represents the minimum average. This is due to the fact that each data point in the plots shows the average loss of the trials at that point in time. For this reason, the minima of the plots are in general not equal to the minima listed in the table

EGD performs best in terms of minimum loss, with MU and SBSMU relatively close at second and third place. PSGD is a distant fourth. Table 4.4 shows

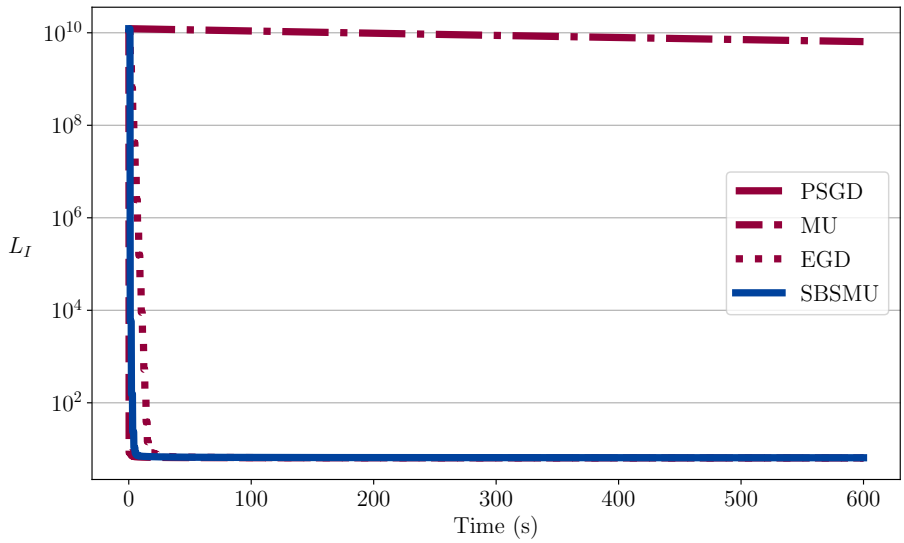



Figure 4.5: Average loss vs time for SBSMU and the benchmark algorithms applied to MNIST.

that the average minimum loss of PSGD is approximately 9 orders of magnitude greater than that of the other algorithms. The same applies to its standard deviation. Conversely, the average minimum losses achieved by SBSMU, MU and EGD differ by less than 1%. As the numbers are relatively close, we use Welch's t-test to determine whether the means are significantly different. This is an extension of Student's t-test, designed to be more reliable for populations with different standard deviations [50]. We use Welch's t-test to estimate the probability that the true average minimum loss of SBSMU is smaller than or equal to that of MU or EGD. This gives a probability of 0.062% for MU, and 0.016% for EGD. Hence it is evident that the average minimum loss of SBSMU is significantly greater than that of both MU and EGD.

Considering the convergence time, either MU or EGD performs best depending on the threshold we set for convergence. SBSMU follows as either second or third. Once again, PSGD is a distant fourth, regardless of the threshold. After 10 minutes, PSGD has reduced the average loss by less than one order of magnitude. The other algorithms reduce the average loss by more than 9 orders of magnitude in the first 30 seconds. If we set the convergence threshold at about 6.7 or greater, MU converges first, followed by SBSMU and then EGD. For thresholds below this, EGD and SBSMU switch place. If the threshold is lower than 6.45, EGD also surpasses MU.

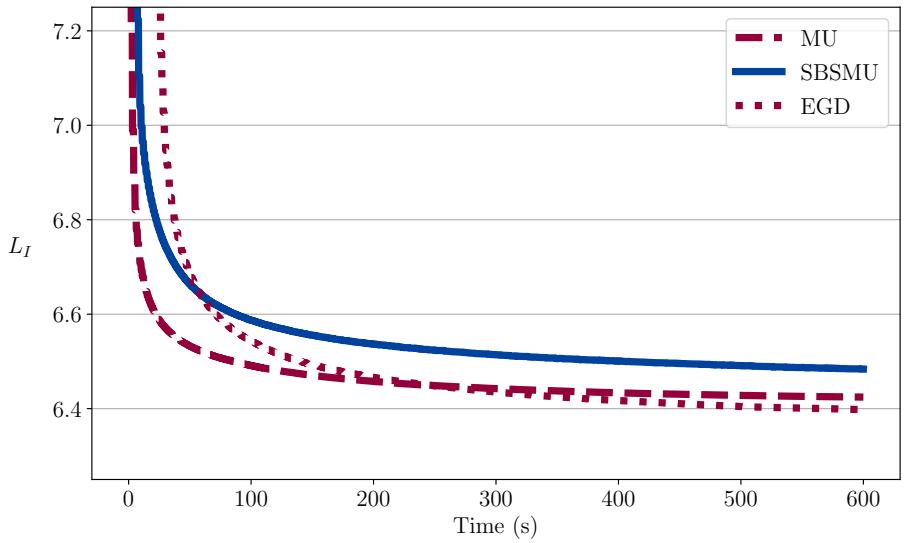



Figure 4.6: Average loss vs time for SBSMU, MU and EGD applied to MNIST. The y-axis has been clipped to visualize trends near convergence.

## 4.4 Big Data Experiment

In this experiment, we compared the performance of SBSMU and MU on Higgs. Higgs is the largest dataset in our database. It consists of 11 000 000 $\times$ 11 000 000 entries, compared to 70 000 $\times$ 70 000 entries for MNIST, our second largest dataset. One important potential application of SBSMU is the factorization of very large datasets. Hence, the purpose of this experiment was to assess the performance of SBSMU in this setting. This addresses the questions that fall under RQ2.

### 4.4.1 Setup and Method

The experiment was run on the Dell desktop computer. Although this meant limited computational resources compared to the cluster, it allowed us to ensure nearly identical runtime environments for the two algorithms. Each algorithm was run only once, with a time limit of 225 minutes. Like in the experiment on MNIST, we saved temporary copies of $\boldsymbol{W}$ and performed loss calculation after the runs were completed. We sampled $\boldsymbol{W}$ once every 15 minutes.

For both SBSMU and MU, we used the same configuration as in the MNIST experiment. Due to the size of Higgs, a hyperparameter search would have been

Table 4.5: Minimum loss for SBSMU and MU applied to Higgs.

| **Algorithm** | **Minimum loss** |
|---|---|
| SBSMU | 82.630 |
| MU | 13.214 |

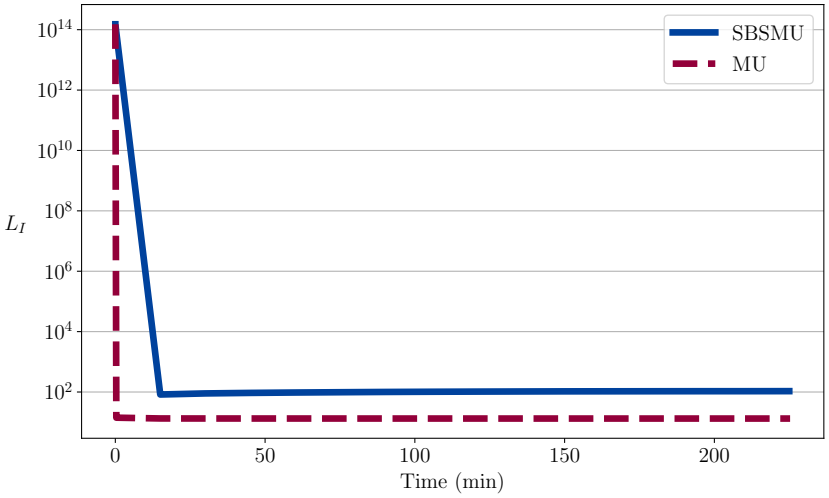



Figure 4.7: Loss vs time for SBSMU and MU applied to Higgs.

prohibitively expensive, even on the cluster.

### 4.4.2 Results

The minimum loss for both algorithms is shown in Table 4.4. MU outperforms SBSMU in terms of minimum loss, with about 13 compared to 83. Since we only conducted a single run, we cannot estimate the standard deviation for either algorithm. Neither can we use statistical tools to determine whether the difference is significant.

The loss is plotted against time in Figure 4.7. MU outperforms SBSMU in terms of the rate of convergence as well, regardless of the time interval under consideration. Beyond considerations of convergence rate, the plot shows that both algorithms are able to reduce the initial loss by about 12-13 orders of magnitude.

# 5 Discussion

In this section we assess the results from our theoretical analysis and experiment and discuss the research questions put forward in the introduction. Therefore, the structure of the discussion follows the research questions. In Section 5.1 we discuss if SBSMU can be proven to be convergent, and if it is possible to parallelize it. This covers RQ1. In Section 5.2, we address RQ2.1 by evaluating the experiments with regards to robustness to hyperparameters and datasets. Lastly, Section 5.3 covers RQ2.2 and RQ2.3 by considering how SBSMU performs in terms of minimum loss and time to convergence.

## 5.1 Convergence and Parallelization

RQ1 asks whether it is possible to prove the convergence of stochastic MU, and whether the updates can be parallelized. In short, we are able to prove the convergence of SBSMU for sufficiently high values of $\alpha$, and we are also able to parallelize the algorithm.

### 5.1.1 Conditional Proof of Convergence

Using the well-known MM algorithm, we are able to prove that the loss is non-increasing under SBSMU. The proof is presented in Section 3.2 and relies on a certain condition on the bound-and-scale parameter. This confirms RQ1.1.

The bound-and-scale step in SBSMU is parameterized by the hyperparameter $\alpha$, which plays a central role in the convergence proof. The key challenge in achieving convergence is to counteract the impact of what we call cross-change: the changes in the loss for all entries of the ground truth except the one we are currently learning from. The bound-and-scale step accomplishes this as long as the convergence condition holds. Specifically, the term $\frac{\alpha}{1-\alpha}$ upper bounds the rate of cross-change relative to the magnitude of change in the factor matrix. In practice, this means that when the learning signals from different entries point in the same direction, lower values of $\alpha$ are sufficient. Conversely, when different entries suggest different directions, the value of $\alpha$ needs to be big enough to counteract the potential impact of cross-change.

This condition is more likely to hold if SBSMU is applied to clustering. Clustering datasets based on similarity matrices is presented in Section 2.2. It is the main application for symmetric NMF. In a clustering problem, the entries in the

ground truth matrix represent similarities between samples. The ground truth matrix is constructed by applying the same similarity metric to every pair of entries in the original dataset. This process imposes some structure on the ground truth matrix. For example, if $a$ and $b$ are similar and $b$ and $c$ are similar, then $a$ and $c$ should be somewhat similar as well. All samples that belong in one cluster hence attract the corresponding rows in the factor matrix closer. As a result, learning from one entry in the similarity matrix is likely to reduce the loss for other entries as well. As a result, the cross-change is smaller and in turn, smaller values can be chosen for $\alpha$.

The convergence condition gives us some hints on how to tune the bound-and-scale parameter $\alpha$ when applying SBSMU in practice. Since the term $\frac{\alpha}{1-\alpha}$ represents an upper bound, higher values are more likely to converge. This is apparent in the hyperparameter experiment, where high values of $\alpha$ lead to lower minimum losses. However, lower values increase the effect of the current sample on the update and in turn increase the convergence rate. Thus the optimal setting is the lowest possible $\alpha$ that still yields convergence. Here, it may be an advantage that the convergence condition is not only a theoretical construct in the proof, but computable at every iteration. It can therefore provide feedback on the choice of $\alpha$.

The proof confirms the need for an algorithm that augments the naive attempt to apply stochastic partial derivatives. In Section 3.1.3, we give an intuitive explanation for why the naive stochastic approach fails to converge. In the proof, the issue is formalized as the need for counteracting the cross-change. SBSMU does this by applying the bound-and-scale step. To reflect this additional step in the proof, we introduce the bound-and-scale-divergence. The bound-and-scale-divergence formalizes how the bound-and-scale step counteracts the cross-change as long as the convergence condition on $\alpha$ holds. Hence SBSMU successfully augments the naive approach to produce a converging algorithm.

### 5.1.2 Parallelization

Section 3.3.5 shows how we parallelize SBSMU. We base our approach on the Hogwild algorithm. The authors behind Hogwild showed that algorithms with sparse data access can be parallelized without locking objects. Sparse data access means that the algorithm only accesses a fraction of the variables that are updated every iteration. SBSMU fulfills this criterion, as each update accesses at most two rows in the factor matrix $\boldsymbol{W}$. This allows us to use workers that perform the update loop completely independently of each other, while they still share the same factor matrix.

An important advantage of the Hogwild algorithm is that it does not require object locking or any other communication between the workers. This greatly

simplifies implementation, because it reduces the need for special objects and communication protocols. For the same reason it reduces the risk of error, and parallelization issues are notoriously hard to debug.

## 5.2 Robustness

RQ2.1 asks how robust SBSMU is to different datasets, hyperparameter configurations and random factors. The random factors are the initialization of $\boldsymbol{W}$ and the order in which the indices are chosen. We can evaluate RQ2.1 based on the experimental results and on the theoretical work. In particular, the hyperparameter experiment tests how SBSMU performs with different random seeds, datasets and configurations. In this experiment, we performed a grid search with different configurations on 8 different datasets. Because each configuration and dataset was tested with 10 different random seeds, we can also assess how sensitive SBSMU is to the random factors. In addition, the proof provides some information on the robustness to different values for the bound-and-scale variable and the learning rate.

#### Different Hyperparameter Configurations within Datasets

SBSMU is parameterized by three different values: the bound-and-scale parameter $\alpha$, the learning rate $\eta$ and the stratified sampling threshold $\beta$. The proof provides some hints on the robustness to different values for the bound-and-scale parameter and fixes the learning rate to a specific value. $\beta$ is an optimization variable and not included in the proof. In the experiments, SBSMU seems to be robust to different configurations of $\eta$ and $\beta$ for high values of $alpha$.

The proof suggests that high values of $\alpha$ are more likely to converge, but yield a slower convergence rate. As discussed previously, $\alpha$ must upper bound the cross-change speed. SBSMU is therefore more likely to converge for higher values. At the same time the rate of convergence is higher for lower values.

In the theoretical section, the value of the learning rate $\eta$ is fixed to a value depending on the loss function. For the Euclidean distance, $\eta = 1/3$ and for the I-divergence $\eta = 1/2$. With these values, convergence is guaranteed. However, the proof can be relaxed to include lower values, and other settings may be beneficial in practice.

Experimentally, SBSMU is sensitive to the configuration, but it appears that this is only the case for relatively low values of $\alpha$. For all of the datasets tested in the hyperparameter experiment, performance varied greatly depending on the configuration. The worst configuration produced an average minimum loss that was several orders of magnitude greater than that of the best configuration.

However, SBSMU appears more robust if we only consider the configurations where $\alpha = 0.99999$. For these configurations, the difference between the worst and best is always within one order of magnitude. In general, the higher the value of $\alpha$, the smaller the impact of $\eta$ and $\beta$ on the average minimum loss. This is an important insight because the highest values of $\alpha$ also give the lowest average minimum losses overall.

It seems that a good standard configuration is $\alpha = 0.99999$, $\beta = 0.5$ and $\eta = 0.5$. $\beta$ is here simply set to the middle of its domain and $\eta$ is set to the value derived in the proof. With this configuration, the average minimum loss is within 15% of the best result for all datasets. Whether 15% is significant depends on the application, and the discrepancy may be larger for other datasets. Performing a hyperparameter search to find a better configuration could therefore still be necessary. In that case the data suggests that the search should be limited to $\alpha \geq 0.9999$.

#### Optimal Hyperparameter Configurations across Datasets

Another factor that potentially impacts how different hyperparameter configurations perform in comparison is the dataset. SBSMU is robust to the tested datasets for high values of $\alpha$. Nevertheless, it is evident that the optimal configuration varies significantly depending on the dataset, with the exception that the best configurations consistently include a high $\alpha \geq 0.9999$.

In the convergence condition in the proof, $\alpha$ indirectly depends on the magnitude of the partial derivative components. As a result, $\alpha$ would be heavily dependent on the dataset at hand if their magnitude varies significantly between datasets. To remedy this, we normalize the data in the experiments to reduce variation in the magnitude of partial derivative components.

Experimentally, it seems that the optimal configuration for SBSMU to a large extent depends on the dataset to be factorized. For the 8 datasets tested, there were large differences in the best configurations we found. In particular, there appears to be no general trend for the best configurations when it comes to $\beta$ and $\eta$. For some datasets, the best configuration had high values of both $\beta$ and $\eta$. For others, the best configuration had low values of both, or a high value for one and a low value for the other.

However, high values of $\alpha$ gave consistently better performance than lower values of $\alpha$. For all the datasets tested, the best configuration included $\alpha \geq 0.9999$. This was also true in the MNIST experiment, where the best configuration we found used $\alpha = 0.999999$. We therefore believe that a relatively good performance can be expected by using a standard configuration, without any tuning of hyperparameters.

#### Random Factors

The data from both the hyperparameter experiment and the MNIST experiment suggests that SBSMU is not particularly sensitive to the random factors. The impact of these factors can be evaluated by considering how much the minimum loss varies with different random seeds. In the hyperparameter experiment, the standard deviation of the minimum loss is $< 5\%$ for all datasets. In the MNIST experiment, the standard deviation of the average minimum loss of SBSMU is lower than those of EGD and MU, both in absolute numbers and as percentages. These results suggest that SBSMU is relatively stable to different random seeds. This is important, because high sensitivity to initialization is undesirable in most applications. If SBSMU depended heavily on how it is initialized, it would be necessary to run it multiple times to ensure a good outcome.

## 5.3 Performance Comparison

Based on RQ2.2 and RQ2.3, we hoped to achieve two improvements by introducing stochasticity:

- **Lower minimum loss by escaping local minima.** A stochastic algorithm is not monotonically decreasing with the respect to the loss, which may allow it to escape local minima. We lose the theoretically guaranteed convergence, but may be able to achieve a lower loss in practice.

- **Increased convergence rate by constant time updates.** Stochasticity may increase the rate of convergence. For large datasets, each update is calculated and applied in a fraction of the time it takes for a full-batch algorithm.

Here we evaluate these potential areas for improvement, mostly based on empirical data from the experiments.

Focusing first on minimum loss, results from the experiments with MNIST suggest that SBSMU outperforms PSGD but not MU or EGD. In the MNIST experiment, the average minimum loss achieved by SBSMU was 9 orders of magnitude lower than that achieved by PSGD. Thus SBSMU outperformed the only stochastic benchmark by a wide margin. Nevertheless, we also found that the average minimum loss of SBSMU was greater than that of both MU and EGD. It is worth noting that the relative difference was small, as the average minimum losses of SBSMU, MU and EGD differed by $< 1\%$. However, it is not possible to say whether this difference would be of importance in applications. It may for example translate to a greater difference in homogeneity in a clustering problem.

Similarly, SBSMU was outperformed by MU in terms of minimum loss in the Higgs experiment. The relative difference was much larger than in the MNIST experiment. The minimum loss achieved by SBSMU was more than 5 times greater than that of MU. Since we only performed a single run with each algorithm, it was not possible to perform a statistical analysis on these results. However, in the hyperparameter experiment and the MNIST experiment, the standard deviations we calculated were all $< 5\%$. Assuming that this also holds for Higgs, the difference between SBSMU and MU is clearly significant.

It is unclear whether the proposed mechanism, whereby stochasticity allows SBSMU to escape local minima, has merit. Clearly, SBSMU was not able to find better local optima than the deterministic algorithms for MNIST or Higgs. This does not support the hypothesis, but neither does it completely refute it, as we only considered two datasets.

Regardless of whether SBSMU is able to escape local minima in practice, stochasticity may prevent it from achieving complete convergence. While the full-batch MU is monotonically decreasing, this is not the case for SBSMU. There may be entries in the dataset that pull the values of $\boldsymbol{W}$ in different directions. Since the learning rate is constant, there is no mechanism that forces SBSMU to settle down completely. Note that this explanation does not conflict with the proof of convergence. The proof shows that the loss is nonincreasing in expectation. However, the minimum that this expected value represents may be higher than that found by MU.

In terms of time to convergence, the results from the MNIST experiment showed MU outperforming SBSMU, while SBSMU outperformed PSGD. The rank of EGD depends on the threshold in loss where we consider the algorithms to have converged. EGD eventually achieved the lowest average loss. With a sufficiently low threshold it thus outperformed both MU and SBSMU. The first 50-60 seconds saw most of the decrease in loss for SBSMU, MU and EGD, with a reduction in average loss of about 9 orders of magnitude. Conversely, the remaining 9 minutes produced less than 10% in decrease for any of the three. If we focus on the period of rapid decline over the first minute, MU reached convergence fastest, followed by SBSMU, EGD and then PSGD. PSGD performed far worse than the other algorithms, regardless of the threshold. Over the full 10 minutes it reduced the loss by less than one order of magnitude.

The results from the Higgs experiment were similar in that MU outperformed SBSMU in time to convergence. We only sampled losses every 15 minutes in this experiment, so it is not possible to determine how close the two algorithms were. The data shows that MU reduced the loss by 12-13 orders of magnitude after a single iteration, and that SBSMU had reached a similar level after 15 minutes. However, SBSMU never reached an average loss lower than or equal to that of MU. Hence, regardless of any threshold, MU had a lower convergence time than

SBSMU.

When discussing convergence time, it is also worth considering the question of initialization cost. This is not included in the plots in the MNIST experiment and the Higgs experiment. SBSMU takes 7-8 seconds to initialize, mostly due to the time it takes to start the processes used for parallelization. As a result, the initialization is near constant with respect to the size of the dataset and negligible when the factorization runs for several minutes or hours. However, the initialization cost is far from negligible for small datasets that may take $< 1$ second to converge. Hence, SBSMU is unlikely to be a suitable choice for factorizing small datasets.

# 6 Evaluation

Here we evaluate our work, including the merits and limitations of what we have presented in the previous chapters. In Section 6.1, we summarize our answers to the research questions based on the discussion. Section 6.2 addresses the limitations of our work. Finally, Section 6.3 outlines possible avenues for future research.

## 6.1 Conclusions

The research questions presented in Section 1.1 ask whether stochasticity can be introduced to MU for symmetric NMF, and if so, how the algorithm would perform. More specifically, RQ1.1 asks if the algorithm can be proven to converge and RQ1.2 if it can be parallelized. RQ2.1 asks how robust the algorithm is to different settings. Finally, RQ2.2 and RQ2.3 asks how it performs in terms of minimum loss and convergence time, respectively. We were indeed able to develop a stochastic multiplicative algorithm for symmetric NMF, namely SBSMU. In the remainder of this section, we answer each of the research questions with respect to SBSMU.

SBSMU can be proven to converge under a certain condition on the bound-and-scale parameter. We provide the proof in Section 3.2. The convergence condition links the bound-and-scale parameter to the speed of cross-change. For a given iteration of SBSMU, the cross-change is the change in loss for all entries except those in the current mini-batch. The proof from Section 3.2 demonstrates how the bound-and-scale step in SBSMU counteracts the cross-change. As long as $\alpha$ is sufficiently close to 1, so that $\frac{\alpha}{1-\alpha}$ upper bounds the speed of cross-change, the algorithm will converge.

Furthermore, SBSMU can easily be parallelized without object locking. We show how parallelization can be implemented in Section 3.3.5, and use this approach in the experiments. The approach is based on the Hogwild framework, and utilizes the fact that data access to the factor matrix is sparse.

The data suggests that SBSMU is generally robust to the random factors and relatively robust to datasets and configurations. Although the optimal configuration appears to vary significantly between datasets, it consistently contains $\alpha \geq 0.9999$. Furthermore, there is relatively little variation in minimum loss for different values of the two other parameter, $\eta$ and $\beta$, as long as $\alpha \geq 0.9999$.

The performance of SBSMU is not on par with the best of the benchmarks. In terms of minimum loss, the experiments on MNIST and Higgs suggest that SBSMU is better than PSGD, but significantly worse than EGD and MU. Considering time to converge, the data again suggests that SBSMU is better than PSGD and worse than MU, while the comparison with EGD is unclear.

## 6.2 Limitations

In this section we seek to address the main limitations of our work. We first discuss limitations regarding the scope of this thesis and assumptions made in our theoretical work. We intentionally kept the scope narrow in order to be able to investigate the research questions in-depth. The second group of limitations are those related to our experiments. Some stem from the inherent drawbacks of empirical research, while others are due to practical trade-offs we made because of limited time and resources.

### 6.2.1 Scope and Theoretical Analysis

The scope of our work is limited to symmetric NMF with either the Euclidean distance or the I-divergence. The theoretical analysis and proof in Section 3 have only been evaluated in this context. It may be that SBSMU can be extended to other NMF problems, and it is likely that the algorithm can be used with other separable loss functions. Similarly, it may be that the proof holds for other loss functions in its current form. However, we make no claims about the validity of these conjectures.

For the optimization of SBSMU and the experimental work, we have further limited our scope to the factorization of sparse matrices. This is because the most likely applications of a stochastic algorithm for symmetric NMF involves very large, sparse matrices. Furthermore, we only consider the I-divergence because that is the loss function most often used to factorize sparse datasets. We introduce several techniques for optimizing the execution time of SBSMU in Section 3.3. Among these techniques, stratified sampling and simplified update calculation for zero-valued terms are only applicable when factorizing sparse matrices. The remaining techniques, parallelization, compilation and random number generation in lists, should in theory work for dense matrices as well. Preliminary experiments confirmed this.

The convergence proof in Section 3.2 relies on a condition on the bound-and-scale parameter $\alpha$, stated in Theorem 3.2.1. The theorem assumes that $\frac{\alpha}{1-\alpha}$ upper bounds the speed of cross-change, specifically the rate of cross-change relative to the change in the factor matrix. This means that convergence is only guaranteed for values of $\alpha$ that are sufficiently close to 1. We have not been able to

theoretically upper bound the speed of cross-change, and so we cannot determine a specific minimum for $\alpha$. However, the speed of cross-change is computable, and in practice it appears that the optimal value of $\alpha$ is relatively consistent for different datasets.

### 6.2.2 Experimental Results

Regarding the conclusions we draw from our experimental results, there are some limitations that apply to all empirical research. The problem of induction is a central issue in empirical research, as it is inherently problematic to induce general rules from specific observations. The conclusions we draw attempt to generalize from a handful of datasets and a limited number of randomly initialized trials. It is impossible to know with certainty whether the patterns we observe would hold for every possible dataset or random initialization. However, we have taken two steps to mitigate this. Firstly, we ran our experiments on multiple datasets and ensured that these represent a number of different domains. Secondly, we ran each experiment multiple times with different random seeds and recorded the variance. The exception was the experiment on Higgs, where doing multiple runs would have been prohibitively expensive.

Aside from the unavoidable problem of induction, there are some limitations that stem from how we designed the experiments:

- **Unoptimized benchmark algorithms.** In the benchmark experiment and the big data experiment, we relied on our own implementations of the benchmark algorithms. We used the same libraries as for the SBSMU implementation and applied straightforward optimizations, but did not parallelize the benchmark algorithms or implement problem-specific improvements. Furthermore, we did not perform hyperparameter searches for EGD and MU. We performed a limited hyperparameter search for PSGD during preliminary experiments. In conclusion, the benchmark algorithms may have performed better if we had spent more time on their implementation and configuration.

- **Time limits.** Throughout the experiments, we ran the factorization algorithms with a time limit that sometimes terminated them before convergence. The time limits were imposed due to limited computational resources, and were chosen to allow most of the algorithms to converge. Nevertheless, the algorithms may have been able to achieve lower minimum losses given more time. This is particularly relevant for PSGD, which clearly did not have enough time to find a local minimum for any of the large datasets in the benchmark experiment.

- **Limited loss sampling.** We found the minimum losses and generated the plots by sampling $\boldsymbol{W}$ during the factorization process. Calculating the loss after every iteration would not have been computationally feasible, particularly for the stochastic algorithms and the large datasets. The limited number of samples means that we may have missed patterns. Furthermore, for the algorithms without monotonically decreasing loss, we may not have sampled the true minimum loss. In practice, this means that the true lowest losses of SBSMU and PSGD are likely to be lower than they appear in the data.

These limitations mean that we cannot draw strong conclusions from the empirical experiments. For example, we cannot state with certainty that the configuration $\alpha = 0.99999, \beta = 0.5, \eta = 0.5$ will always close to optimal results for SBSMU. We can only state that the experiments support this hypothesis.

## 6.3 Future Work

There are several potential avenues for future research. Here we present the ones we believe show the greatest promise.

### Heuristics for Adaptive Learning Rates

Optimization algorithms often need to find a balance between exploration and exploitation. Exploration is a coarse search in the solution space, implemented by updates that represent large steps. This is typically useful in the first part of the search, where the goal is to explore as much of the solution space as possible. Towards the end of the search, the focus shifts to exploitation of the most promising areas. This is achieved by performing fine-grained searches in the areas surrounding the best known solutions.

In SBSMU, this trade-off is mainly controlled by the learning rate $\eta$. Higher values of $\eta$ mean larger steps and thus more exploration of the solution space. Lower values of $\eta$ put more emphasis on exploitation. In the current version of the algorithm $\eta$ is fixed, which means that it is not possible to prioritize exploration initially and exploitation later on. It is likely that an adaptive learning rate, enabling this shift, would be beneficial. Preliminary experiments with gradually decreasing learning rates did not lead to lower minimum losses or faster convergence. However, finding the right heuristic could yield significant improvements.

### Application of SBSMU to Other NMF Problems

This work only considers symmetric factorization of sparse matrices. SBSMU could be applied to other NMF problems, such as linear NMF and projective

NMF. The pseudocode in Section 3.1.4 is specific to symmetric NMF. However, the manner in which stochasticity is introduced and the bound-and-scale step itself can in theory be applied directly to other NMF problems. Furthermore, the application of SBSMU to other loss functions beyond Euclidean distance and I-divergence should only require changing the partial derivative components.

### Extension of the Empirical Work

By extending our empirical work, other researchers would either strengthen or refute the conclusions we put forward. We discussed the most important limitations of our experiments in Section 6.2. They include the fact that we only tested SBSMU on 10 different datasets. Furthermore, while we included 8 datasets in the hyperparameter experiment, we only evaluated performance for 2 datasets. While these results give some indication of how SBSMU performs, it would be useful to perform more experiments. Beyond applying SBSMU to more datasets, the results would be strengthened by increasing the time limits and the frequency of loss sampling.

### Adaptive Tuning of the Bound-and-Scale Parameter

The proof of convergence in Section 3.2 relies on a computable assumption on the bound-and-scale parameter $\alpha$. It provides a lower bound for the value of alpha in each iteration, where values closer to the bound are preferable because they result in a higher rate of convergence. Checking this convergence condition is computationally expensive and cannot be done at every iteration. However, an extension of SBSMU that checks and adapts $\alpha$ at certain intervals could improve performance. This approach would adjust $\alpha$ based on the dataset and the progression of the factorization. In turn, it could reduce the need to perform expensive hyperparameter searches with $\alpha$.

### Proving a Minimum for the Bound-and-Scale Parameter

Although our proof shows that SBSMU is more likely to converge for higher values of $\alpha$, we do not prove the existence of a minimum value for $\alpha$. We require that $\frac{\alpha}{1-\alpha}$ upper bounds the speed of cross-change, which is the rate of cross-change relative to the rate of change in the factor matrix. Proving the existence of an upper bound on the cross-change and quantifying it would strengthen the proof, and provide a theoretical minimum for the value of $\alpha$.

# Bibliography


[1] *Anaconda Software Distribution*. 2020. URL: https://anaconda.com/.

[2] Bill Andreopoulos et al. "A roadmap of clustering algorithms: Finding a match for a biomedical application". In: *Briefings in bioinformatics* 10 (Mar. 2009), pp. 297–314. DOI: 10.1093/bib/bbn058.

[3] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for Exotic Particles in High-Energy Physics with Deep Learning". In: *Nature communications* 5 (July 2014), p. 4308. DOI: 10.1038/ncomms5308. URL: https://archive.ics.uci.edu/ml/datasets/HIGGS.

[4] Léon Bottou. "Stochastic Gradient Learning in Neural Networks". In: *Proceedings of Neuro-Nimes*. Vol. 8. Nov. 1991.

[5] S. S. Bucak and B. Gunsel. "Incremental subspace learning via non-negative matrix factorization". In: *Pattern Recognition* 42.5 (2009), pp. 788–797. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2008.09.002. URL: http://www.sciencedirect.com/science/article/pii/S0031320308003725.

[6] Anil Chitade and Sunil Katiyar. "Color based image segmentation using K-means clustering". In: *International Journal of Engineering Science and Technology* 2 (Oct. 2010).

[7] A. Cichocki et al. "Non-negative matrix factorization with $\alpha$-divergence". In: *Pattern Recognition Letters* 29.9 (2008), pp. 1433–1440. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2008.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0167865508000767.

[8] Andrzej Cichocki et al. *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation*. Wiley, Oct. 2009. ISBN: 978-0-470-74666-0. DOI: 10.1002/9780470747278.

[9] Kristin Dana et al. "Reflectance and texture of real-world surfaces". In: *ACM Transactions on Graphics (TOG)* 18 (Aug. 1999), pp. 1–34. DOI: 10.1145/300776.300778. URL: https://www.cs.columbia.edu/CAVE/software/curet/index.php.

[10] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477. URL: http://yann.lecun.com/exdb/mnist/.

[11] Inderjit S. Dhillon and Suvrit Sra. "Generalized Nonnegative Matrix Approximations with Bregman Divergences". In: NIPS'05. Vancouver, British Columbia, Canada: MIT Press, 2005, pp. 283–290.

[12] Chris Ding, Xiaofeng He, and Horst D. Simon. "On the Equivalence of Nonnegative Matrix Factorization and Spectral Clustering." In: *SIAM International Conference on Data Mining*. Jan. 2005.

[13] Chris Ding et al. "Orthogonal Nonnegative Matrix T-Factorizations for Clustering". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 126–135. ISBN: 1595933395. DOI: 10.1145/1150402.1150420. URL: https://doi.org/10.1145/1150402.1150420.

[14] Wei Dong, Moses Charikar, and Kai Li. "Efficient K-nearest neighbor graph construction for generic similarity measures". In: Jan. 2011, pp. 577–586. DOI: 10.1145/1963405.1963487.

[15] Ronald A. Fisher. "THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS". In: *Annals of Human Genetics* 7 (1936), pp. 179–188. URL: https://archive.ics.uci.edu/ml/datasets/iris.

[16] Michele Forina et al. *PARVUS: An Extendable Package of Programs for Data Exploration*. Jan. 1998. ISBN: 0-444-43012-1. URL: https://archive.ics.uci.edu/ml/datasets/wine.

[17] Pascal Fua and Krzysztof Lis. "Comparing Python, Go, and C++ on the N-Queens Problem". In: *arXiv e-prints* (Jan. 2020), arXiv:2001.02491. arXiv: 2001.02491 [cs.MS].

[18] Michelle Girvan and Mark Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99 (Nov. 2001), pp. 7821–7826. URL: http://networkdata.ics.uci.edu/data/football/.

[19] T. R. Golub et al. "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring". In: *Science* 286.5439 (1999), pp. 531–537. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.531. URL: http://leo.ugr.es/elvira/DBCRepository/Leukemia/ALLAML.html.

[20] N. Guan et al. “Online Nonnegative Matrix Factorization With Robust Stochastic Approximation”. In: *IEEE Transactions on Neural Networks* 23 (July 2012), pp. 1087–1099. DOI: 10.1109/TNNLS.2012.2197827.

[21] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[22] J. Huang et al. “Robust Manifold Nonnegative Matrix Factorization”. In: *ACM Transactions on Knowledge Discovery from Data* 8 (June 2014), pp. 1–21. DOI: 10.1145/2601434.

[23] David R. Hunter and Kenneth Lange. “Quantile Regression via an MM Algorithm”. In: *Journal of Computational and Graphical Statistics* 9.1 (2000), pp. 60–77. ISSN: 10618600. URL: http://www.jstor.org/stable/1390613.

[24] H. Kasai. “Accelerated stochastic multiplicative update with gradient averaging for nonnegative matrix factorizations”. In: *2018 26th European Signal Processing Conference (EUSIPCO)*. 2018, pp. 2593–2597. DOI: 10.23919/EUSIPCO.2018.8553610.

[25] H. Kasai. “Stochastic Variance Reduced Multiplicative Update for Nonnegative Matrix Factorization”. In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2018, pp. 6338–6342. DOI: 10.1109/ICASSP.2018.8461325.

[26] Jyrki Kivinen and Manfred K. Warmuth. “Exponentiated Gradient versus Gradient Descent for Linear Predictors”. In: *Information and Computation* 132.1 (1997), pp. 1–63. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.2612. URL: https://www.sciencedirect.com/science/article/pii/S0890540196926127.

[27] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[28] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[29] Yann Lecun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86 (Dec. 1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: http://archive.ics.uci.edu/ml/datasets/Gisette.

[30] D. Lee and H. Seung. "Algorithms for Non-negative Matrix Factorization". In: *Adv. Neural Inform. Process. Syst.* 13 (Feb. 2001).

[31] Xiaolong Liu, Zhidong Deng, and Yuhan Yang. "Recent progress in semantic image segmentation". In: *Artificial Intelligence Review* 52 (Aug. 2019), pp. 1–18. DOI: 10.1007/s10462-018-9641-3.

[32] Stuart P. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/tit.1982.1056489. URL: http://www.cs.toronto.edu/~roweis/csc2515-2006/readings/lloyd57.pdf.

[33] David Lusseau et al. "The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations". In: *Behavioral Ecology and Sociobiology* 54 (Jan. 2003), pp. 396–405. DOI: 10.1007/s00265-003-0651-y. URL: http://www.casos.cs.cmu.edu/computational_tools/datasets/external/dolphins/index11.php.

[34] James B. Macqueen. "Some methods for classification and analysis of multivariate observations". In: *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*. 1967, pp. 281–297.

[35] Andrew Kachites McCallum. "Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering". 1996. URL: https://www.cs.cmu.edu/~mccallum/bow/rainbow/.

[36] Michael McCool, Arch D. Robison, and James Reinders. "Chapter 8 - Fork–Join". In: *Structured Parallel Programming*. Ed. by Michael McCool, Arch D. Robison, and James Reinders. Boston: Morgan Kaufmann, 2012, pp. 209–251. ISBN: 978-0-12-415993-8. DOI: https://doi.org/10.1016/B978-0-12-415993-8.00008-6. URL: https://www.sciencedirect.com/science/article/pii/B9780124159938000086.

[37] Peter R. Mercer. "Convex Functions and Taylor's Theorem". In: *More Calculus of a Single Variable*. New York, NY: Springer, 2014, pp. 171–208. ISBN: 978-1-4939-1926-0. DOI: 10.1007/978-1-4939-1926-0_8. URL: https://doi.org/10.1007/978-1-4939-1926-0_8.

[38] Tom M. Mitchell. "Chapter 6 - Bayesian Learning". In: *Machine Learning*. Boston: McGraw-Hill, 1997, pp. 182–184. ISBN: 978-0-07-115467-3. URL: https://kdd.ics.uci.edu/databases/20newsgroups/.

[39] James M. Orthega and Werner C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. New York: Academic Press, 1970, pp. 253–255. ISBN: 978-0-12-528550-6.

[40] J. W. Paisley, D. B., and M. I. Jordan. "Bayesian Nonnegative Matrix Factorization with Stochastic Variational Inference". In: *Handbook of Mixed Membership Models and Their Applications*. 2014.

[41] Benjamin Recht et al. “Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

[42] Bernhard Schölkopf, Alex Smola, and Klaus-Robert Müller. “Nonlinear Component Analysis as a Kernel Eigenvalue Problem”. In: *Neural Computation* 10 (July 1998), pp. 1299–1319. DOI: 10.1162/089976698300017467.

[43] R. Serizel, S. Essid, and G. Richard. “Mini-batch stochastic approaches for accelerated multiplicative updates in nonnegative matrix factorisation with beta-divergence”. In: *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2016, pp. 1–6. DOI: 10.1109/MLSP.2016.7738818.

[44] Magnus Själander et al. *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. 2019. arXiv: 1912.05848.

[45] Pang-Ning Tan et al. *Introduction to Data Mining*. 2nd ed. Pearson, May 2005, pp. 487–490. ISBN: 978-0-32-132136-7.

[46] “Taylor expansions and applications”. In: *Mathematical Analysis I*. Milano: Springer Milan, 2008, pp. 223–255. ISBN: 978-88-470-0876-2. DOI: 10.1007/978-88-470-0876-2_7. URL: https://doi.org/10.1007/978-88-470-0876-2_7.

[47] Sergios Theodoridis. *Pattern Recognition*. 4th ed. Academic Press, Oct. 2008, p. 203. ISBN: 978-1-59-749272-0.

[48] Steven K. Thompson. “Chapter 11 - Stratified Sampling”. In: *Sampling*. Ed. by Walter A. Shewhart and Samuel S. Wilks. Hoboken, New Jersey: John Wiley & Sons, 2012, pp. 139–156. ISBN: 978-1-11-816293-4.

[49] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[50] B. L. Welch. “The Generalization of ‘Student’s’ Problem when Several Different Population Variances are Involved”. In: *Biometrika* 34.1/2 (1947), pp. 28–35. ISSN: 00063444. URL: http://www.jstor.org/stable/2332510.

[51] Zhirong Yang and Erkki Oja. “Unified Development of Multiplicative Algorithms for Linear and Quadratic Nonnegative Matrix Factorization”. In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 22 (Dec. 2011), pp. 1878–91. DOI: 10.1109/TNN.2011.2170094.

[52] Zhirong Yang et al. "Kullback-Leibler Divergence for Nonnegative Matrix Factorization". In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Ed. by Timo Honkela et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 250–257. ISBN: 978-3-642-21735-7.

[53] R. Zhao and V. Y. F. Tan. "Online Nonnegative Matrix Factorization With Outliers". In: *IEEE Transactions on Signal Processing* 65.3 (2017), pp. 555–570. DOI: 10.1109/TSP.2016.2620967.

[54] G. Zhou et al. "Online Blind Source Separation Using Incremental Nonnegative Matrix Factorization With Volume Constraint". In: *IEEE Transactions on Neural Networks* 22.4 (2011), pp. 550–560. DOI: 10.1109/TNN.2011.2109396.

# Appendices

## A Mathematical Notation

Table A.1 contains an overview of the mathematical notation we use in this thesis.

| Notation | Definition |
|---|---|
| $\boldsymbol{X}$ | Ground truth matrix, with $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{n \times n}$ in symmetric NMF and $\boldsymbol{X} \in \mathbb{R}_{\geq 0}^{n \times m}$ in linear NMF |
| $\boldsymbol{W}$ | Factor matrix, with $\boldsymbol{W} \in \mathbb{R}_{\geq 0}^{n \times r}$ |
| $\boldsymbol{H}$ | 2nd factor matrix in linear NMF, with $\boldsymbol{H} \in \mathbb{R}_{\geq 0}^{r \times m}$ |
| $\widehat{\boldsymbol{X}}$ | Approximation of $\boldsymbol{X}$, where $\widehat{\boldsymbol{X}} = \boldsymbol{W}\boldsymbol{W}^\top$ in symmetric NMF and $\widehat{\boldsymbol{X}} = \boldsymbol{W}\boldsymbol{H}$ in linear NMF |
| $\widetilde{\boldsymbol{W}}$ | Factor matrix as a variable |
| $\widetilde{\boldsymbol{X}}$ | Approximation matrix as a variable |
| $X_{ij}$ | Entry at row $i$ and column $j$ in matrix $\boldsymbol{X}$ |
| $r$ | Rank of the factor matrix |
| $\alpha$ | Bound-and-scale variable |
| $\beta$ | Stratified sampling variable |
| $\eta$ | Learning rate |
| $L_{EU}$ | Euclidean distance |
| $L_I$ | I-divergence |
| $L$ | Unspecified loss function |
| $L^{(ij)}$ | Stochastic approximation of $L$ based on $X_{ij}$ |
| $\nabla_{al}$ | $\dfrac{\partial L}{\partial W_{al}}(\boldsymbol{W})$ |
| $\nabla_{al}^+, \nabla_{al}^-$ | Positive and negative components of $\nabla_{al}$ |
| $\nabla_{ij,al}$ | $\dfrac{\partial L^{(ij)}}{\partial W_{al}}(\boldsymbol{W})$ |
| $\psi_{\min}, \psi_{\max}$ | Constants used in the unified development procedure |
| $\omega$ | Exponentiated gradient descent variable |

Table A.1: Overview of notation

# B Stochastic Update Rule Derivation

Here we show how the update rule in SBSMU can be derived from $F^{(ij)}$ as defined in Section 3.2. The partial derivative of the stochastic majorization function is

$$\begin{aligned}\frac{\partial G^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} &= \frac{\partial}{\partial \widetilde{W}_{al}} \left[ \frac{W_{al}}{\psi_{\max}} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}} \nabla^{+}_{ij,al} \right. \\ &\qquad \left. - \frac{W_{al}}{\psi_{\min}} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}} \nabla^{-}_{ij,al} \right] + \text{constant} \qquad \text{(B.1)} \\ &= \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}-1} \nabla^{+}_{ij,al} - \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}-1} \nabla^{-}_{ij,al}. \qquad \text{(B.2)}\end{aligned}$$

The partial derivative of the bound-and-scale-divergence is

$$\begin{aligned}\frac{\partial H(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} &= \frac{\partial}{\partial \widetilde{W}_{al}} \left[ \frac{\left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}} - 1}{\psi_{\max}} - \frac{\left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}} - 1}{\psi_{\min}} \right] \qquad \text{(B.3)} \\ &= \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}-1} - \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}-1}. \qquad \text{(B.4)}\end{aligned}$$

Combining the above results, the partial derivative of $F^{(ij)}$ is

$$\begin{aligned}&\frac{\partial F^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} \\ &= \frac{\partial}{\partial \widetilde{W}_{al}} \frac{1}{2} \left[ G^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) + \frac{\alpha}{1-\alpha} H(\widetilde{\boldsymbol{W}}, \boldsymbol{W}) \right] \qquad \text{(B.5)} \\ &= \frac{1}{2} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\max}-1} \left( \nabla^{+}_{ij,al} + \frac{\alpha}{1-\alpha} \right) - \frac{1}{2} \left( \frac{\widetilde{W}_{al}}{W_{al}} \right)^{\psi_{\min}-1} \left( \nabla^{-}_{ij,al} + \frac{\alpha}{1-\alpha} \right). \qquad \text{(B.6)}\end{aligned}$$

Setting (B.6) equal to zero and solving for $\widetilde{W}_{al}$ gives the stochastic update rule,

$$\frac{\partial F^{(ij)}(\widetilde{\boldsymbol{W}}, \boldsymbol{W})}{\partial \widetilde{W}_{al}} = 0 \tag{B.7}$$

$$\Leftrightarrow \left(\frac{\widetilde{W}_{al}}{W_{al}}\right)^{\psi_{\max}-1} \left(\nabla^{+}_{ij,al} + \frac{\alpha}{1-\alpha}\right) = \left(\frac{\widetilde{W}_{al}}{W_{al}}\right)^{\psi_{\min}-1} \left(\nabla^{-}_{ij,al} + \frac{\alpha}{1-\alpha}\right) \tag{B.8}$$

$$\Leftrightarrow \left(\frac{\widetilde{W}_{al}}{W_{al}}\right)^{\psi_{\max}-\psi_{\min}} = \frac{\nabla^{-}_{ij,al} + \frac{\alpha}{1-\alpha}}{\nabla^{+}_{ij,al} + \frac{\alpha}{1-\alpha}} \tag{B.9}$$

$$\Leftrightarrow \widetilde{W}_{al} = W_{al} \left(\frac{\alpha + (1-\alpha)\nabla^{-}_{ij,al}}{\alpha + (1-\alpha)\nabla^{+}_{ij,al}}\right)^{\frac{1}{\psi_{\max}-\psi_{\min}}} . \tag{B.10}$$

# C Random Seeds

The random seeds used in the experiments are listed in table C.2. The seeds were chosen uniformly at random from the range $[0, 2^{31} - 1]$.

Table C.2: Random seeds used in experiments.

| **Experiment** | **Random seeds** | | |
|---|---|---|---|
| Hyperparameter | 1573942128 | 0274259859 | 1978143047 |
| | 1059581820 | 0923533011 | 1243444734 |
| | 0976315392 | 1283209697 | 0347312645 |
| | 0403974349 | | |
| MNIST | 1244102535 | 2063541227 | 1096982489 |
| | 1916869805 | 2004766057 | 1872543914 |
| Higgs | 1244102535 | | |

# D Datasets

Table D.3 summarizes the preprocessing applied to every dataset used in this work.

Table D.3: Datasets preprocessing

| **Dataset** | **Preprocessing** |
|---|---|
| 20Newsgroups | Choose a vocabulary of 10 000 words based on maximum information gain using the Rainbow software [35]. Apply TF-IDF to the resulting term frequency matrix. Generate similarity matrix using cosine similarity, make it sparse by applying 5NN. Enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| AML/ALL | Generate sparse matrix by applying 5NN, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| CUReT | Downsample from $200 \times 200$ to $100 \times 100$ pixels per image, extract scattering features and apply PCA to reduce the dimensionality further. Generate sparse matrix by applying 10NN, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| Dolphins | Use adjacency matrix. Scale to $[0, 1]$. |
| Football | Use adjacency matrix. Scale to $[0, 1]$. |
| Gisette | Generate sparse matrix by applying 10NN, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| Higgs | Generate similarity matrix using KGraph [14] with 5 neighbors. Enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| Iris | Generate sparse matrix by applying 5NN, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| MNIST | Generate similarity matrix using KGraph [14] with 10 neighbors, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |
| Strike | Use adjacency matrix. Scale to $[0, 1]$. |
| Wine | Generate sparse matrix by applying 5NN, enforce symmetry by adding its transpose. Scale to $[0, 1]$. |

# E Benchmark Algorithms

### Projected Stochastic Gradient Descent

Stochastic gradient descent was originally proposed by Léon Bottou [4]. Applied to matrix factorization, the updates are of the form

$$W_{al} \leftarrow W_{al} - \eta \nabla_{ij,al}. \tag{E.11}$$

In order to use updates of this form for NMF, a projection step is needed to ensure non-negativity [8]. This can be done by simply taking the maximum of 0 and the updated value. The update rule for NMF is then given by

$$W_{al} \leftarrow \max\left(0, W_{al} - \eta \nabla_{ij,al}\right). \tag{E.12}$$

Combining Projected SGD with the I-divergence, negative values have to be projected to a positive constant instead of 0 as in Equation (E.12). This is a result of the form of the gradient. As derived in Section 3.1.2, gradients of the I-divergence contain the term

$$-\frac{X_{ij}}{\widehat{X}_{ij}}. \tag{E.13}$$

If $X_{ij}$ is positive and $\widehat{X}_{ij} = \sum_k W_{ik} W_{jk}$ approaches zero, this term approaches negative infinity. Subtracting it from $W_{ik}$ leads to infinitely big entries in $\boldsymbol{W}$, which in turn lead to divergence of the algorithm.

Having normalized the data, we empirically set the positive constant to $10^{-16}$. With this adapted projection step, the final update rule used in the experiments is

$$W_{al} \leftarrow \max\left(10^{-16},\ W_{al} - \eta \nabla_{ij,al}\right). \tag{E.14}$$

### Exponentiated Gradient Descent

Exponentiated gradient descent is an optimization algorithm where the updates take the form [26]

$$W_{al} \leftarrow W_{al} \exp\left(-\eta_{al} \nabla_{al}\right). \tag{E.15}$$

To ensure stability, the learning rate $\eta_{al}$ is typically set so that it negatively correlates with the magnitude of $\boldsymbol{W}$

$$\eta_{al} = \frac{\omega}{\sum_i W_{il}}, \tag{E.16}$$

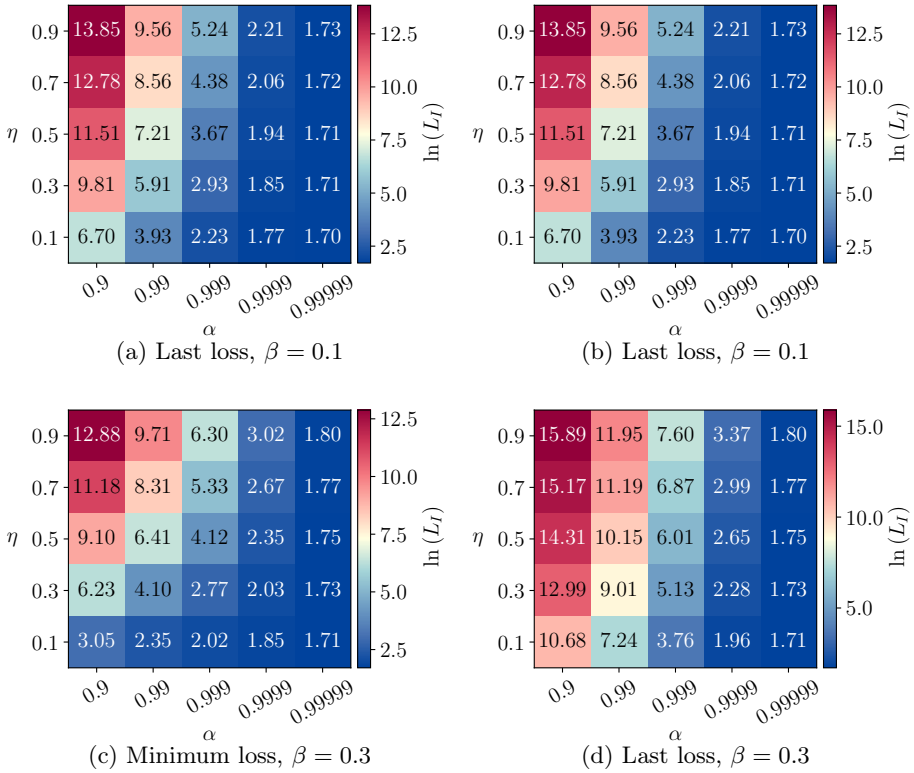
where $\omega \in (0, 2)$ [8].

The update rule (E.15) is multiplicative and maintains the nonnegativity of $\boldsymbol{W}$. It is equivalent to gradient descent applied in the log space

$$\ln(W_{al}) \leftarrow \ln(W_{al}) - \eta_{al} \nabla_{al}. \tag{E.17}$$

# F Hyperparameter Experiment Results

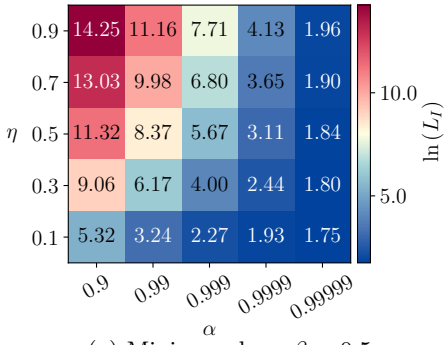
**20Newsgroups**



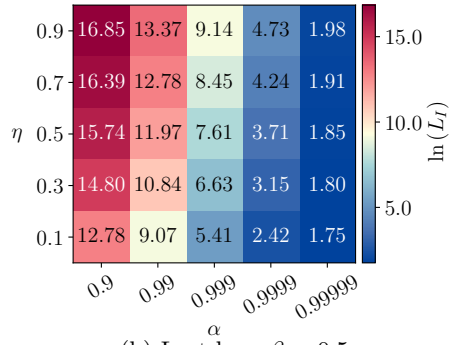

(a) Last loss, $\beta = 0.1$ (b) Last loss, $\beta = 0.1$

(c) Minimum loss, $\beta = 0.3$ (d) Last loss, $\beta = 0.3$

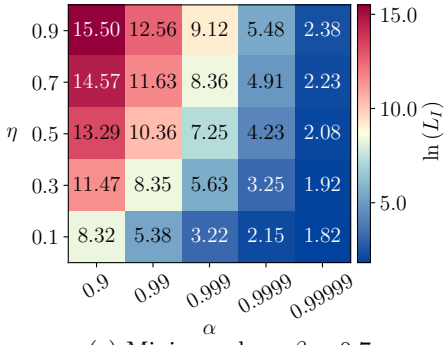
Figure F.1: Hyperparameter experiment results for 20Newsgroups (1/2).



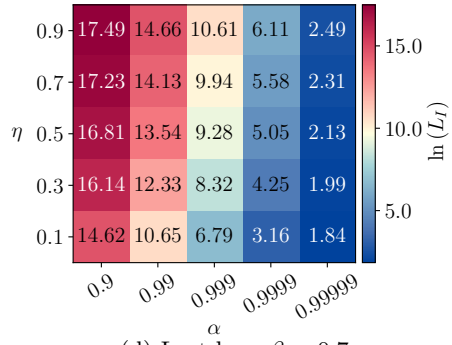

(a) Minimum loss, $\beta = 0.5$



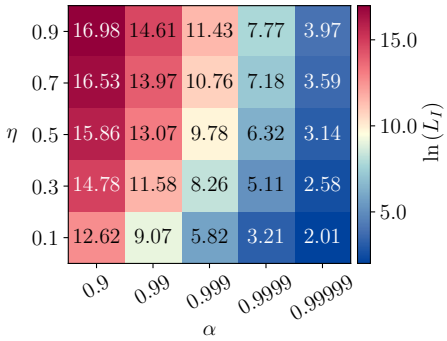

(b) Last loss, $\beta = 0.5$



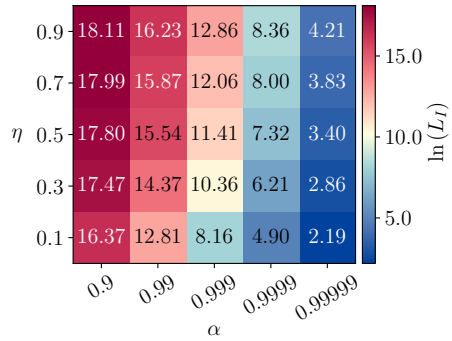

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



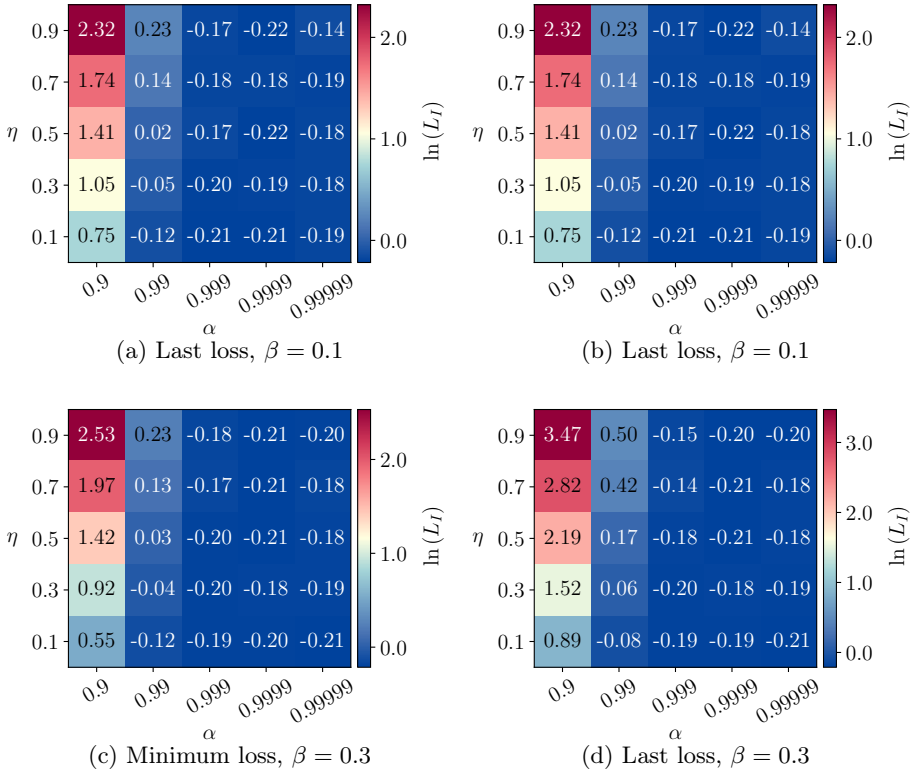

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.2: Hyperparameter experiment results for 20Newsgroups (2/2).

**AML/ALL**



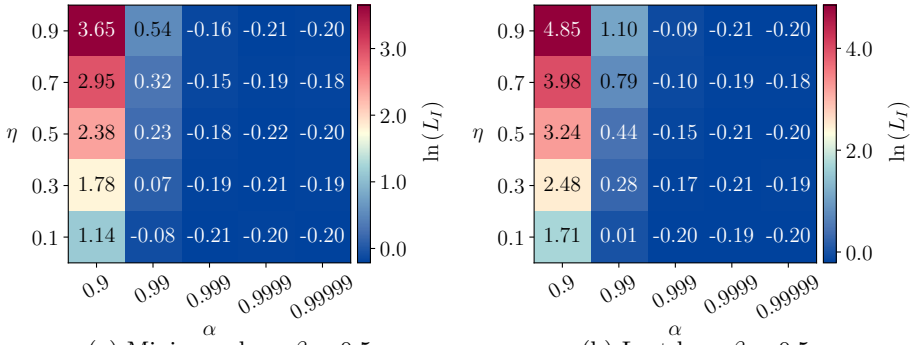

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

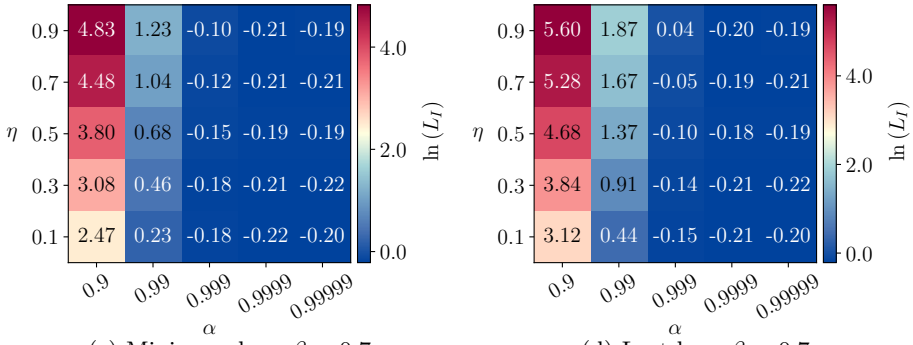
Figure F.3: Hyperparameter experiment results for AML/ALL (1/2).




(a) Minimum loss, $\beta = 0.5$



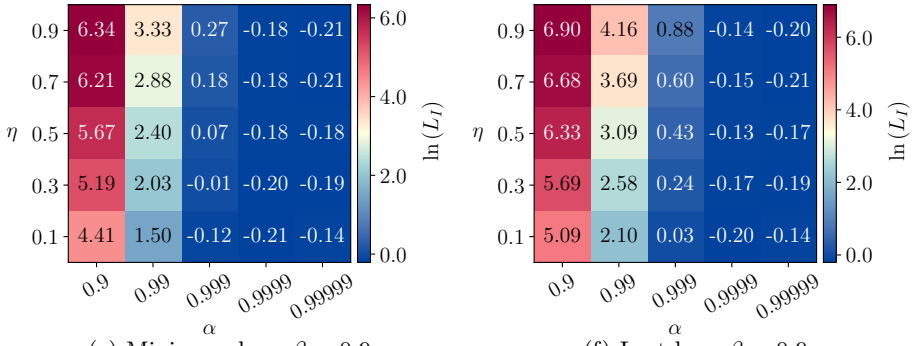

(b) Last loss, $\beta = 0.5$




(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



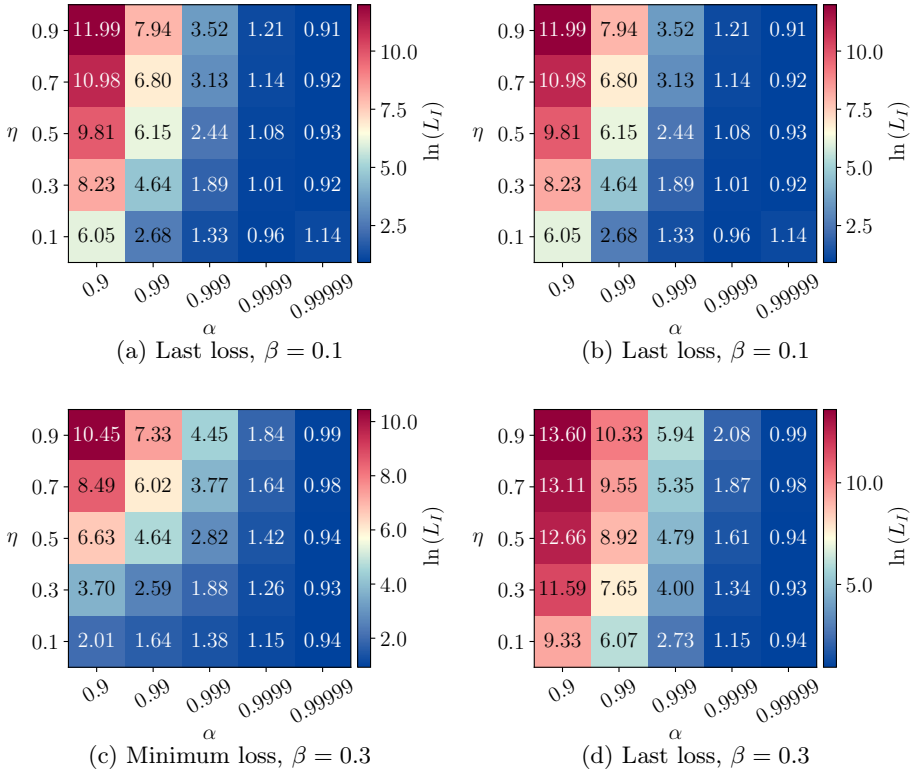

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.4: Hyperparameter experiment results for AML/ALL (2/2).

**CUReT**



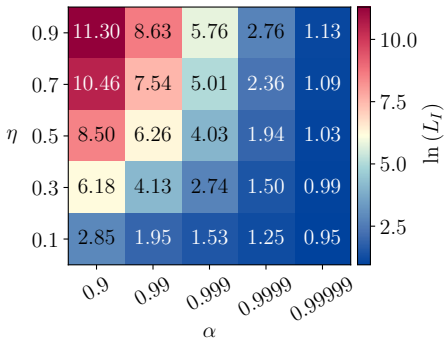

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

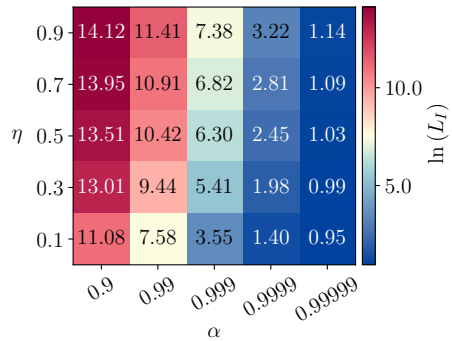
(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

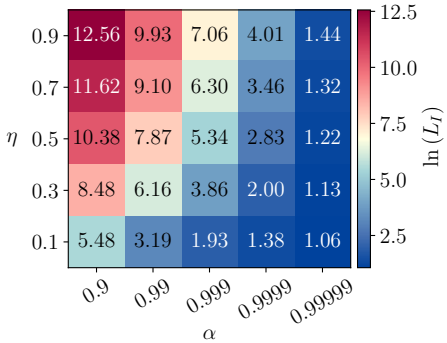
Figure F.5: Hyperparameter experiment results for CUReT (1/2).



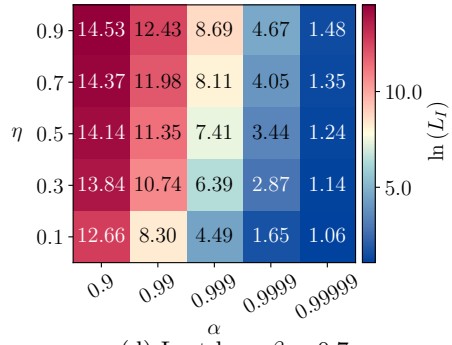

(a) Minimum loss, $\beta = 0.5$



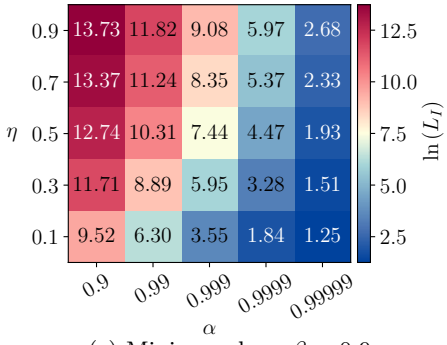

(b) Last loss, $\beta = 0.5$



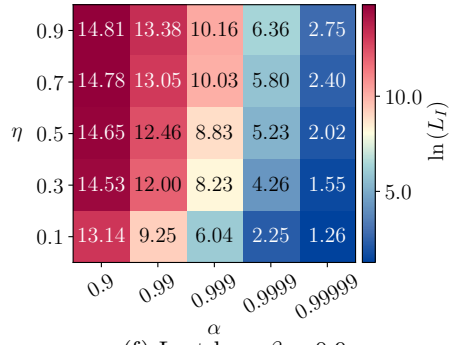

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



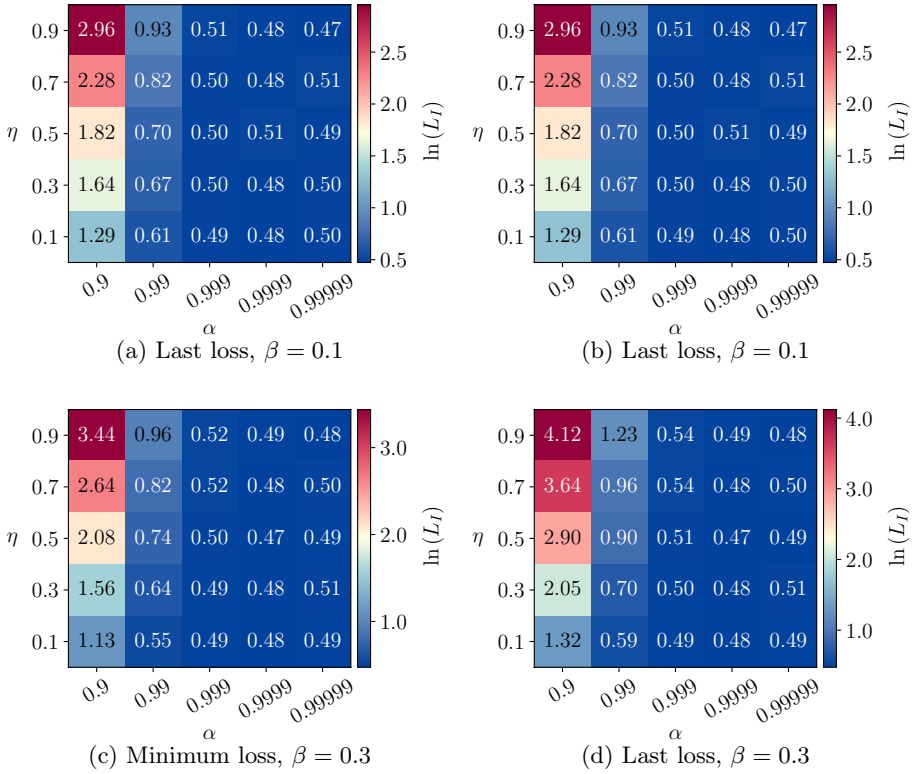

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.6: Hyperparameter experiment results for CUReT (2/2).

**Dolphins**



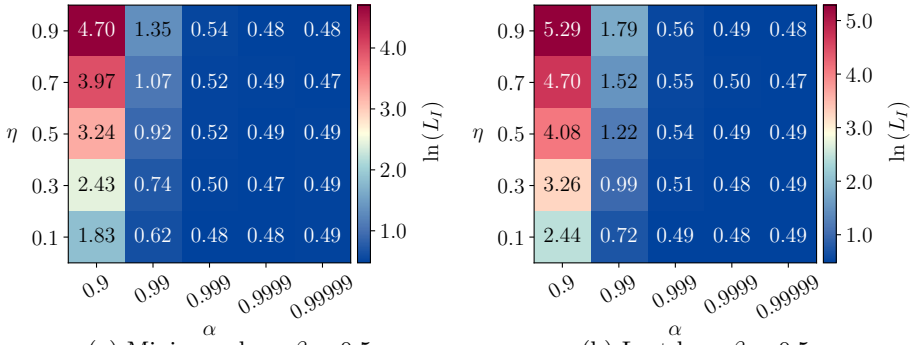

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

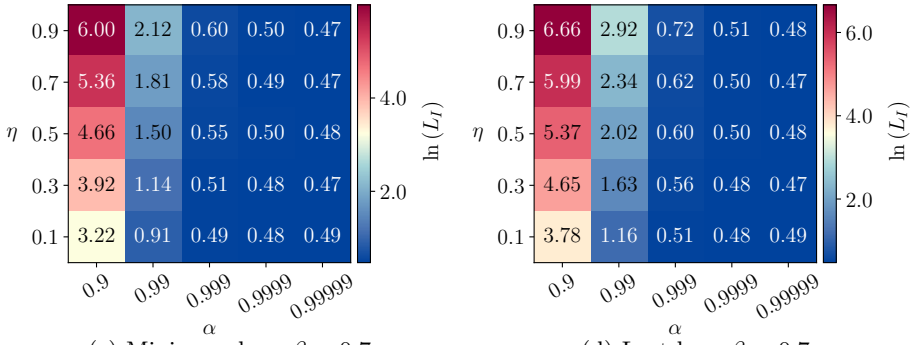
Figure F.7: Hyperparameter experiment results for Dolphins (1/2).




(a) Minimum loss, $\beta = 0.5$



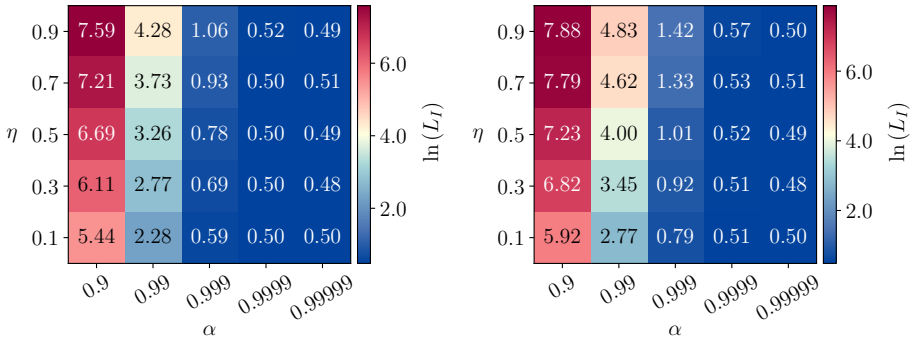

(b) Last loss, $\beta = 0.5$




(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



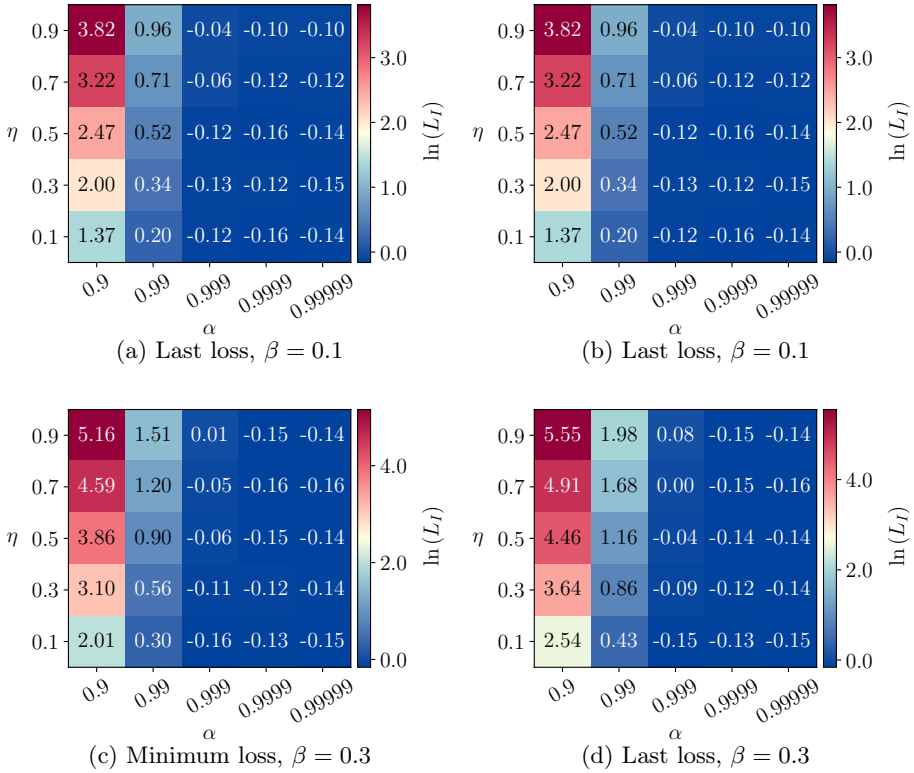

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.8: Hyperparameter experiment results for Dolphins (2/2).

**Football**



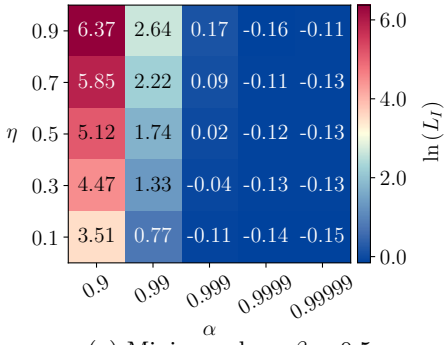

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

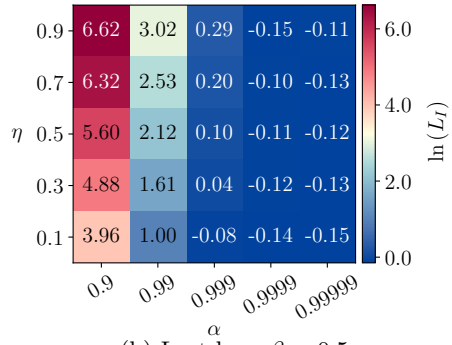
(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

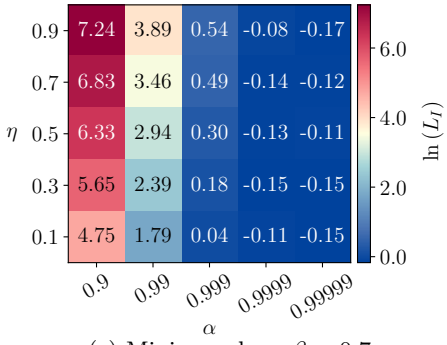
Figure F.9: Hyperparameter experiment results for Football (1/2).



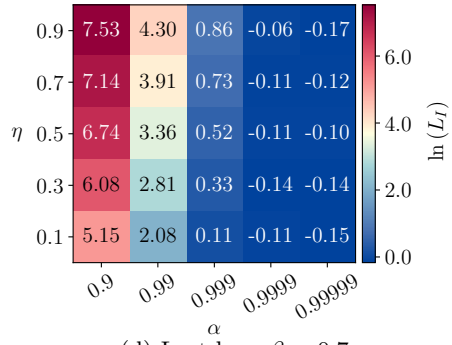

(a) Minimum loss, $\beta = 0.5$



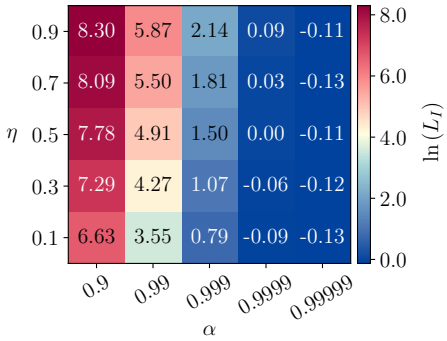

(b) Last loss, $\beta = 0.5$



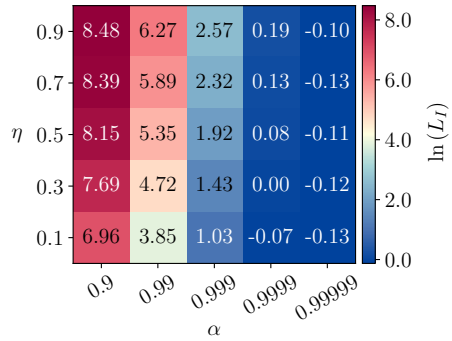

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



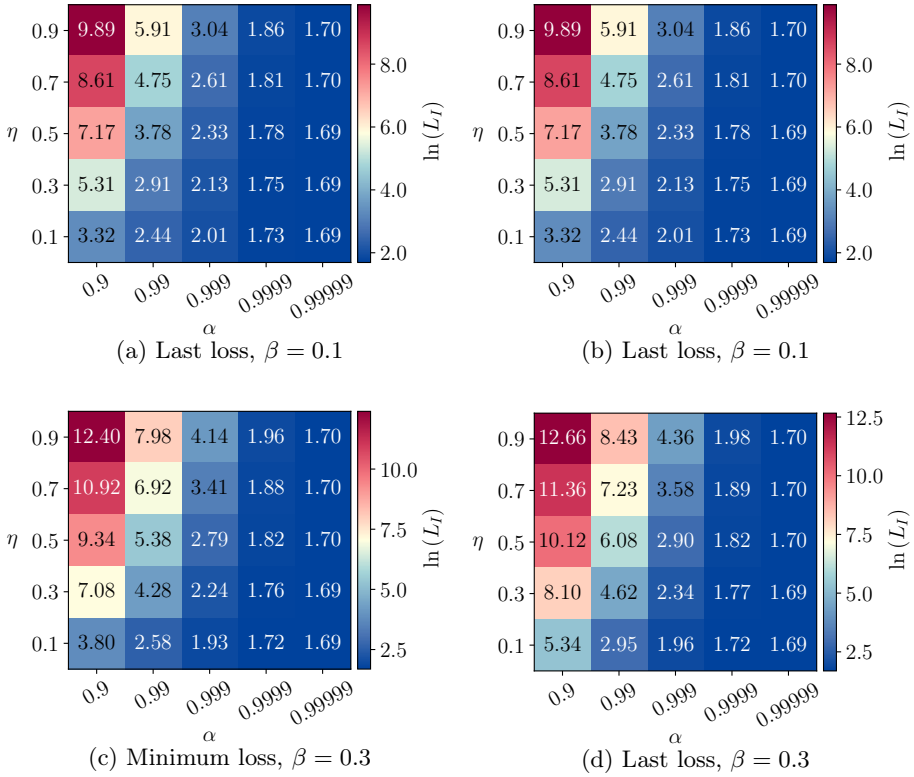

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.10: Hyperparameter experiment results for Football (2/2).

**Gisette**



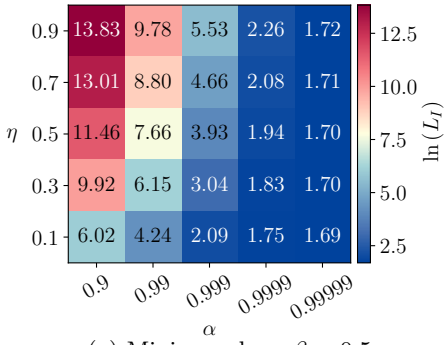

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

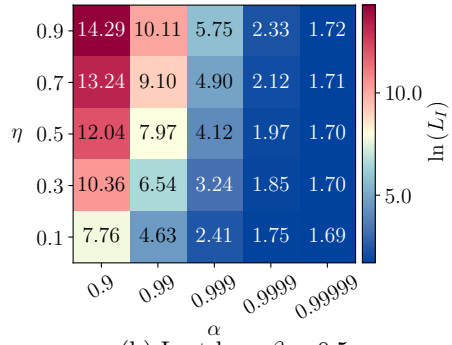
(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

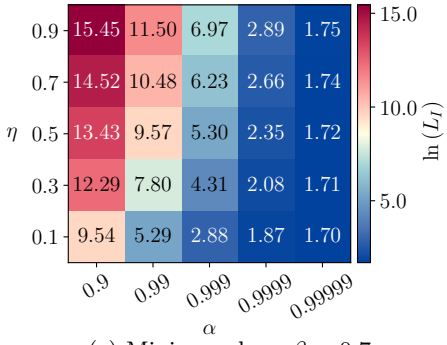
Figure F.11: Hyperparameter experiment results for Gisette (1/2).



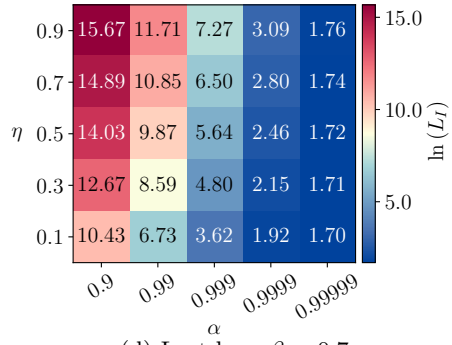

(a) Minimum loss, $\beta = 0.5$



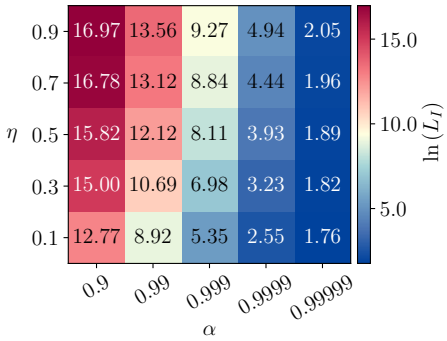

(b) Last loss, $\beta = 0.5$



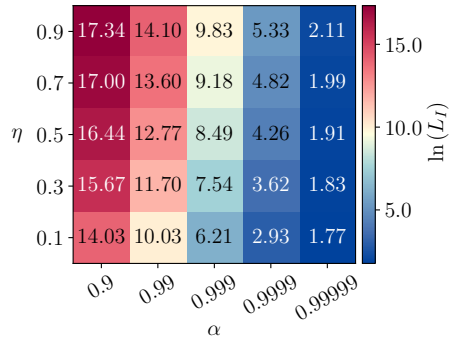

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



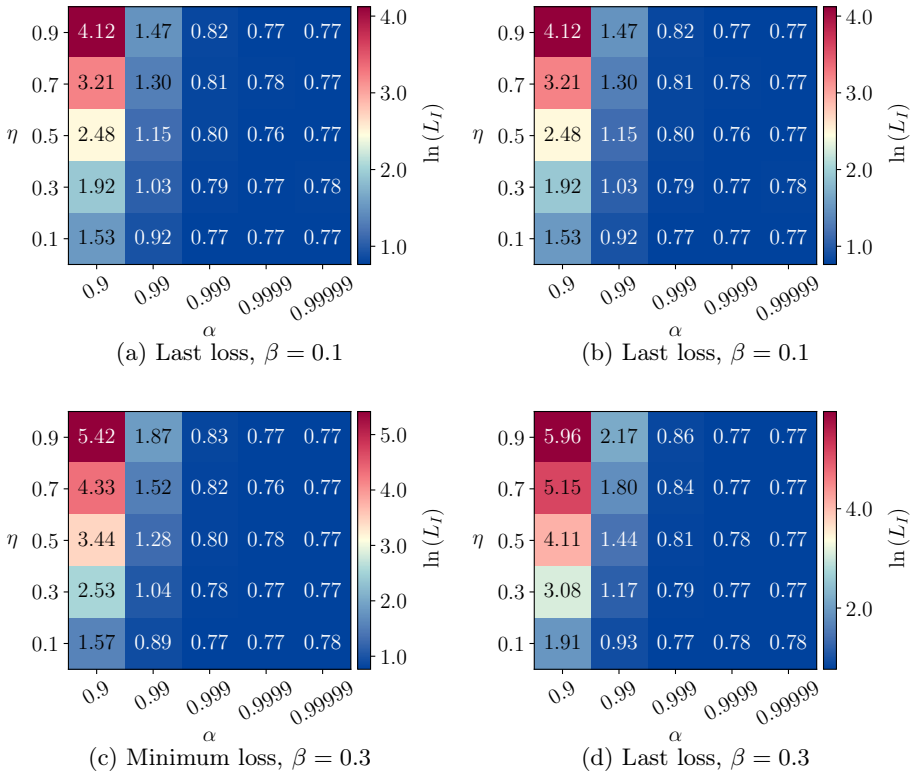

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.12: Hyperparameter experiment results for Gisette (2/2).

**Iris**



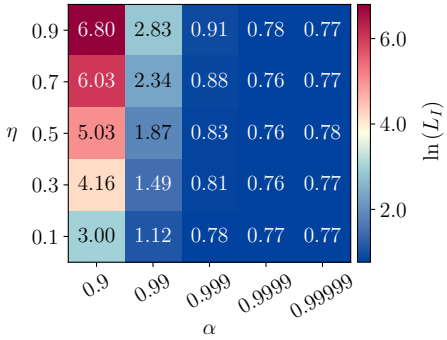

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

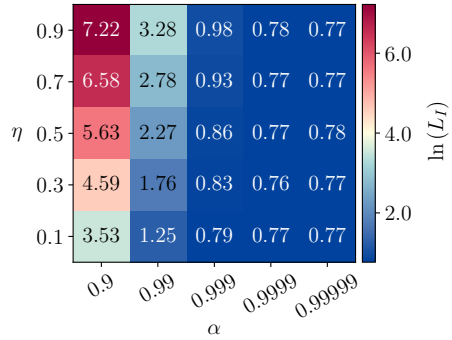
(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

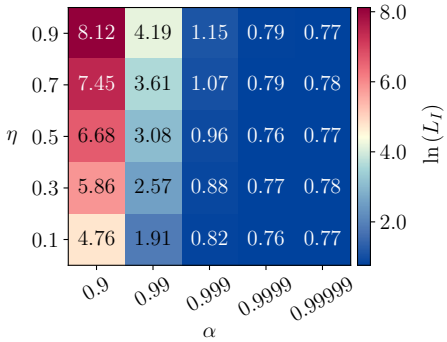
Figure F.13: Hyperparameter experiment results for Iris (1/2).



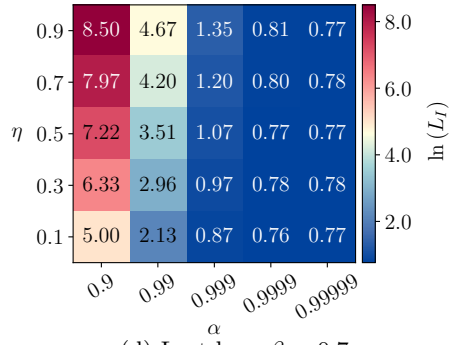

(a) Minimum loss, $\beta = 0.5$



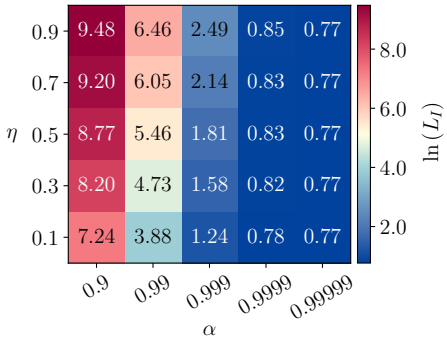

(b) Last loss, $\beta = 0.5$



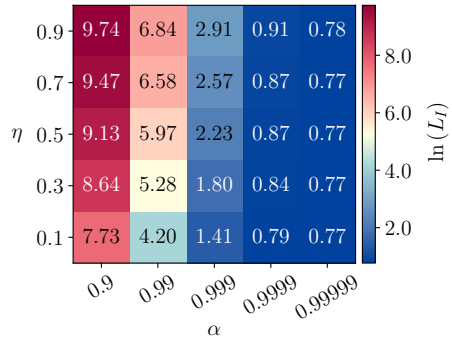

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$



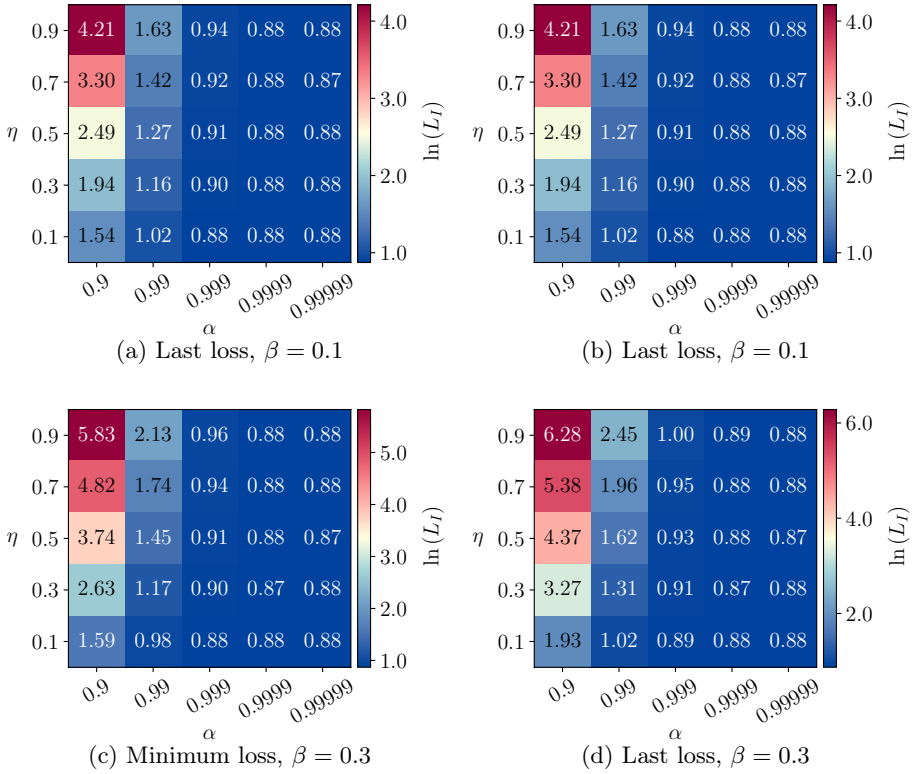

(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.14: Hyperparameter experiment results for Iris (2/2).

**Wine**



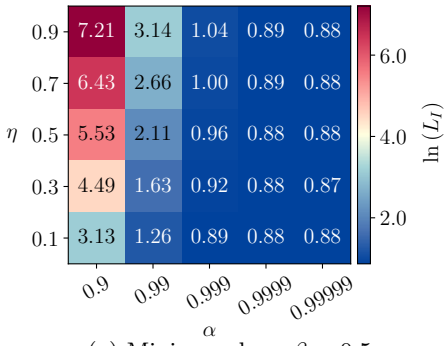

(a) Last loss, $\beta = 0.1$

(b) Last loss, $\beta = 0.1$

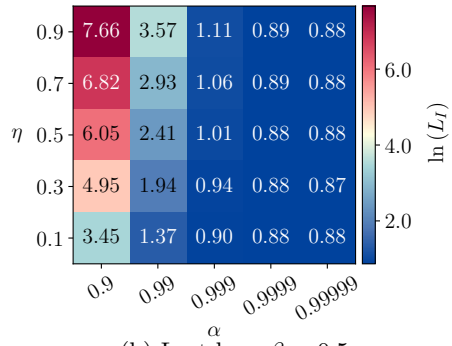
(c) Minimum loss, $\beta = 0.3$

(d) Last loss, $\beta = 0.3$

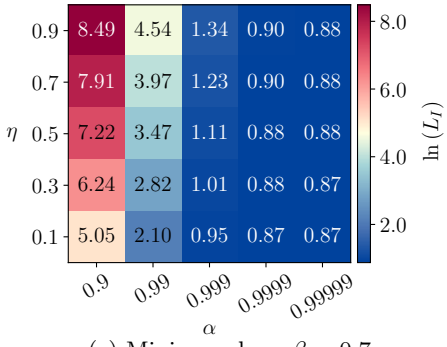
Figure F.15: Hyperparameter experiment results for Wine (1/2).



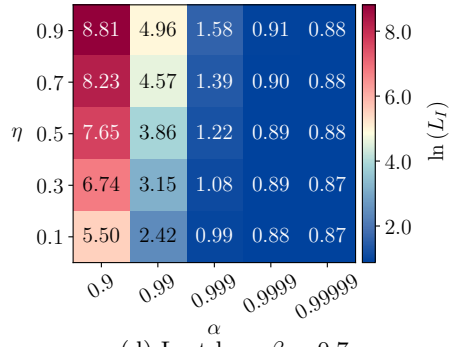

(a) Minimum loss, $\beta = 0.5$



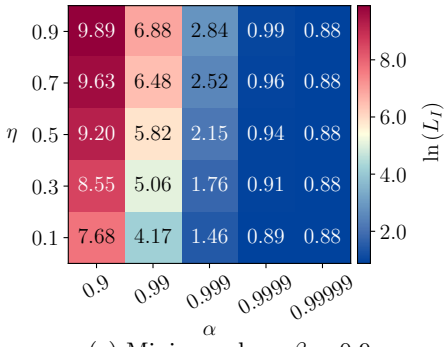

(b) Last loss, $\beta = 0.5$



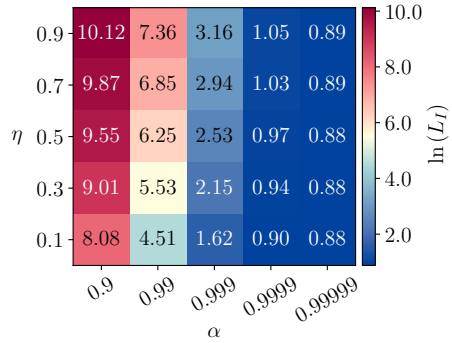

(c) Minimum loss, $\beta = 0.7$




(d) Last loss, $\beta = 0.7$




(e) Minimum loss, $\beta = 0.9$




(f) Last loss, $\beta = 0.9$

Figure F.16: Hyperparameter experiment results for Wine (2/2).

Klara Schlüter and Jon Riege

# Stochastic Multiplicative Updates for Symmetric NMF

NTNU
Norwegian University of
Science and Technology