Computational Science and Engineering
(International Master's Program)

Technische Universität München

Guided Research

Solving least-squares problems involving large dense matrices

Angelos Nikitaras

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Guided Research

Solving least-squares problems involving large dense matrices

Author:	Angelos Nikitaras
Examiner:	Univ.-Prof. Dr. Felix Dietrich
Submission Date:	October 4, 2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

October 4, 2024 Angelos Nikitaras

Abstract

With the advent of Deep Learning, training large-scale neural networks has become a crucial task across a wide range of applications. Typically, the training is performed using gradient-based optimization algorithms, with the most popular being Stochastic Gradient Descent and its variants. Random feature models offer a different method for training neural networks, where the parameters of the hidden layers are sampled from a data-agnostic distribution, such as a normal distribution. Building on this concept, a novel approach, known as the Sampling Where It Matters (SWIM) algorithm, was recently proposed. In SWIM, the network parameters of the hidden layers are constructed using a data-driven sampling scheme, followed by solving a linear least squares problem for the output layer. This linear problem is dense and highly ill-conditioned, posing significant challenges for traditional numerical solvers.

In this work, we investigate numerical methods for solving the least-squares problems that arise in the context of the SWIM algorithm, with a focus on scalability and efficiency. We propose using a recently developed iterative solver called LSRN, which leverages randomized preconditioning to significantly improve the convergence rate. In addition, we introduce an alternative approach that divides the problem into smaller subproblems, which are solved sequentially. We demonstrate the effectiveness of these methods through a series of numerical experiments, showcasing substantial improvements in training speed and scalability.

Contents

1 Introduction

Machine Learning (ML) has rapidly advanced to become a cornerstone of modern technology, driving innovations in diverse areas such as computer vision, natural language processing, and scientific computing [11, 17, 22]. The primary goal of ML is to develop algorithms that can learn patterns from data, enabling the automation of complex tasks that are difficult to address using traditional programming techniques [2]. The recent revolution in ML has been largely driven by the advancements in Deep Learning, a subfield of ML that leverages the power of neural networks to model complex relationships in data [13].

Neural networks are a class of models inspired by the structure of the human brain, consisting of interconnected nodes (*neurons*) that process information. These networks are composed of multiple layers that perform a series of parametrized transformations on the input data, with the goal of learning useful representations for making predictions. Training of neural networks involves adjusting the model's parameters, the so-called *weights* and *biases*, to minimize a predefined loss function, which quantifies the discrepancy between the predicted and true values.

Training neural networks is challenging due to the high-dimensional, non-convex nature of the optimization problem. The most popular approach for training these networks is gradient-based optimization, with Adam and other variants of Stochastic Gradient Descent (SGD) being among the most widely used algorithms [12]. These first-order methods rely solely on the gradient of the loss function, and are known for their slow convergence rates and difficulties in escaping local minima [7]. Additionally, training requires tuning numerous hyperparameters to achieve optimal performance, making the process both laborious and time-consuming.

An alternative to the gradient-based optimization, is the random sampling of the parameters of the hidden layers, which can lead to lower computational cost and in some cases, competitive performance. Extreme learning machines and random feature models are examples of such approaches [19, 10]. Despite their promising results, these methods suffer from a few important limitations. First, the hidden layer parameters are sampled from a data-agnostic distribution, meaning the information contained in the data is not fully exploited. Second, these models are typically shallow, with only one hidden layer, which is incompatible with the deep architectures that have demonstrated state-of-the-art performance in many tasks [13].

Recently, Bolager et al. [4] proposed the Sampling Where It Matters (SWIM) algorithm, a novel approach in which all weights and biases before the last linear layer of a neural network are sampled from a data-dependent probability distribution. The last, linear layer

is then obtained by solving a least-squares problem, which is a well-studied problem in numerical linear algebra [3]. This approach leads to a faster and more stable training process, with significantly fewer hyperparameters to tune. In addition, deep networks can be efficiently constructed, enabling the use of more complex architectures.

In this Guided Research, we will investigate numerical methods for solving least-squares problems arising from the SWIM algorithm. Section 2 provides an overview of Sampled Neural Networks and includes a brief discussion of traditional numerical methods used for solving least-squares problems. In Section 3, we introduce two approaches tailored for large scale applications: the LSRN algorithm, a recently developed parallel iterative solver, and the Split and Solve method, which divides the problem into smaller subproblems that can be solved sequentially. Section 4 presents numerical experiments evaluating the effectiveness of these methods, highlighting their main advantages and limitations. Finally, Section 5 summarizes our findings and discusses potential future research directions.

2 Background

2.1 Sampled Networks

Roughly speaking, Sampled Networks are a class of neural networks, in which each pair of weights and biases of the hidden layers is determined by two points in the input space. In the following, we provide the formal definition [4].

Definition 2.1 (Sampled Neural Networks) *Let Φ be a neural network with L hidden layers, and $\mathcal{X}$ be the input space. We call Φ a sampled neural network if for each layer $l \in \{1, \dots, L\}$, and every neuron $i \in \{1, \dots, N_l\}$, the weights $w_{l,i}$ and the biases $b_{l,i}$ are determined by pairs of data points, $\left(x_{0,i}^{(1)}, x_{0,i}^{(2)}\right)$, sampled from $\mathcal{X} \times \mathcal{X}$. In particular, we define the weights and biases as follows:*

$$w_{l,i} = s_1 \frac{x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}}{\left\| x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)} \right\|^2}, \quad b_{l,i} = \langle w_{l,i}, x_{l-1,i}^{(1)} \rangle + s_2, \tag{2.1}$$

where $s_1, s_2 \in \mathbb{R}$ are constants and $x_{l,i}^{(j)} = \Phi^{l-1}\left(x_{0,i}^{(j)}\right)$ for $j = 1, 2$, with Φ^{l-1} denoting the composition of the first $(l-1)$ layers of Φ. We also assume that $x_{0,i}^{(1)} \neq x_{0,i}^{(2)}$. The weights and the biases of the output layer are obtained through the minimization problem $W_{L+1}, b_{L+1} = \arg\min \mathcal{L}\left(W_{L+1}\Phi^L(\cdot) - b_{L+1}\right)$, where $\mathcal{L}$ is a designated loss function.

The constants s_1 and s_2 control the behavior of the activation functions, when applied to the points $x^{(1)}$ and $x^{(2)}$. For example, for the ReLU activation function, we can set $s_1 = 1$ and $s_2 = 0$.

The above definition, does not specify the sampling strategy for the pairs of data points. Bolager et al. [4] proposed the Sampling Where It Matters (SWIM) algorithm, which introduces a data-dependent probability distribution for sampling the pairs of points. The main idea of the SWIM algorithm is to prefer pairs of points which are close to each other in the input space, but far away in the output space. After determining the weights and biases of the hidden layers, the final layer is obtained by solving a least-squares problem, which is the main focus of this work.

2.2 Numerical methods for solving least-squares problems

Numerical methods for solving linear least-squares problems can be divided into two main categories: direct and iterative methods.

Direct methods perform a finite number of operations to obtain a solution, often using appropriate matrix factorizations. While these methods provide a theoretically exact solution (considering round-off errors), their computational cost can be prohibitive for large-scale problems. The most popular direct methods for solving least-squares problems rely on the Singular Value Decomposition (SVD) and the QR decomposition, with high-performance implementations being available in libraries such as LAPACK [3, 1].

In contrast, iterative methods approximate the solution by iteratively refining an initial guess until a convergence criterion is met. These methods are particularly advantageous for large-scale problems because they only require matrix-vector products, eliminating the need to explicitly form the matrix. However, this advantage comes at the cost of reduced accuracy and, in some cases, slower convergence rates. Popular iterative algorithms for solving least-squares problems include the CGLS and LSQR [9, 18]. The main idea behind these methods is to apply the Conjugate Gradient (CG) algorithm to the normal equations of the least-squares problem, with some modifications for improved stability. CG's convergence rate, however, is significantly affected by the condition number, leading to slow convergence for ill-conditioned matrices, which are the focus of this work [20].

3 Methods

3.1 LSRN

Recently, randomized methods have gained popularity in the field of numerical linear algebra, showcasing significant improvements over their deterministic counterparts [14, 16]. These methods leverage random numbers to compute approximate solutions, with their main advantage being the reduced computational cost. For solving least-squares problems, Meng et al. [15] proposed LSRN, a parallel iterative solver designed for strongly over-determined or under-determined linear systems. LSRN computes the minimum-norm solution of the minimization problem $\min_x \|\|Ax-b\|\|_2$, where $A \in \mathbb{R}^{m\times n}$, with $m \gg n$ or $m \ll n$. The requirement that $m \gg n$ is well-aligned with the structure of our problem, as m, the number of data points, is typically much larger than n, the number of neurons in the hidden layer. One of LSRN's key advantages is that its convergence rate is unaffected by rank deficiency, making it particularly suitable for ill-conditioned problems.

The algorithm consists of a preconditioning phase followed by the application of an iterative solver, such as the LSQR algorithm or the Chebyshev semi-iterative method [6]. To construct the preconditioner, a random normal projection is first applied to the matrix A, followed by an SVD decomposition of the projected matrix. The preconditioned system is almost surely well-conditioned, enabling the subsequent iterative solver to converge rapidly. For concreteness, pseudocode of the LSRN algorithm is provided in Algorithm 1.

Algorithm 1 LSRN (computes $\hat{x} \approx A^\dagger b$ when $m \gg n$).

1: Choose an oversampling factor $\gamma > 1$ and set $s = \lceil \gamma n \rceil$.
2: Generate $G = \text{randn}(s, m)$, i.e., an s-by-m random matrix whose entries are independent random variables following the standard normal distribution.
3: Compute $\tilde{A} = GA$.
4: Compute $\tilde{A}$'s compact SVD $\tilde{U}\tilde{\Sigma}\tilde{V}^T$, where $r = \text{rank}(\tilde{A})$, $\tilde{U} \in \mathbb{R}^{s\times r}$, $\tilde{\Sigma} \in \mathbb{R}^{r\times r}$, $\tilde{V} \in \mathbb{R}^{n\times r}$, and only $\tilde{\Sigma}$ and $\tilde{V}$ are needed.
5: Let $N = \tilde{V}\tilde{\Sigma}^{-1}$.
6: Compute the min-length solution to $\min_y \|ANy - b\|_2$ using an iterative method. Denote the solution by $\hat{y}$.
7: Return $\hat{x} = N\hat{y}$.

A Python 2 implementation of the LSRN algorithm is provided by Meng et al. [15]. For our experiments, we adapted their code to ensure compatibility with Python 3. Our im-

plementation is based on the Numpy [8] and Scipy [21] libraries, with the iterative solver being the LSQR algorithm, as implemented in Scipy. The code is compatible with the LinearOperator class, an interface for performing matrix-vector and matrix-matrix products without explicitly forming the matrix. This is particularly advantageous for large-scale problems, where the matrix A may not fit into memory.

An important consideration in the implementation is managing the memory requirements for generating the random matrix G. Naively, the generation of G would require storing $s \times m$ entries, significantly increasing the overall memory footprint. To address this, the matrix G is generated on blocks, and the matrix-matrix product GA is computed block-wise, reducing memory consumption. In addition, the block size is tunable, and should be chosen based on the available memory and the architecture of the system. For smaller problems in which memory is not a limiting factor, the full matrix G can be generated at once, which is more efficient than the block-wise approach.

3.2 Split and Solve

An alternative approach to the solution of the minimization problem is to decompose it into smaller subproblems. The main advantage of this method is that the smaller problems may fit into memory, enabling the use of direct methods.

In particular, we consider the least squares problem:

$$\min_{w} |||Aw - b|||_2. \tag{3.1}$$

The matrix A can be seen as column-wise concatenation of smaller matrices $A_1, \ldots, A_k$, i.e.

$$A = \begin{bmatrix} A_1 & \cdots & A_k \end{bmatrix}. \tag{3.2}$$

Similarly, the vector w can be decomposed as $w = \begin{bmatrix} w_1 & \cdots & w_k \end{bmatrix}^T$, where w_i is a row vector, with dimensions corresponding to the number of columns in A_i.

Given this decomposition, the minimization problem (3.1) can be written as:

$$\min_{\substack{w_i \\ i=1,\ldots,k}} |||\sum_{i=1}^{k} A_i w_i - b|||_2 \tag{3.3}$$

The key idea is to iteratively solve these subproblems. We begin by solving the first subproblem $\min_{w_1} \|A_1 w_1 - b\|_2$. Defining the residual $r_1 = A_1 w_1 - b$, we can rewrite the remaining minimization problem as:

$$\min_{\substack{w_i \\ i=2,\ldots,k}} |||\sum_{i=2}^{k} A_i w_i - r_1|||_2. \tag{3.4}$$

This process is repeated, resulting in a sequence of k least squares problems of the form

$$\min_{w_i} \|A_i w_i - r_{i-1}\|_2,$$

where $r_i = A_i w_i - r_{i-1}$, and $r_0 = b$.

In particular for our problem, the matrix A is the output of the hidden layers of the neural network, when the input data is passed through the network. For example, for a single hidden layer network with a $\tanh$ activation function, the matrix A is given by:

$$A = \tanh(XW_h + b_h), \tag{3.5}$$

where X is the input data, W_h is the weight matrix of the hidden layer, and b_h is the bias vector of the hidden layer, assuming a suitable broadcasting operation. To avoid the need to store the whole matrix A in memory, we can generate the submatrices A_i, by partioning the weight matrix W_h into smaller submatrices, i.e $W_h = \begin{bmatrix} W_h^1 & \cdots & W_h^k \end{bmatrix}$, and multiplying the input data X with each submatrix W_h^i.

An important insight here is that in the decomposition of the weight matrix W_h, we have the flexibility to permute its columns, which can enhance the efficiency of solving the resulting subproblems. For instance, by first sorting the columns of W_h based on their norm, and then groupping them into k submatrices, we can achieve substantial improvements. As we will demonstrate in the next section, the ordering of the columns can significantly affect the accuracy of the method.

4 Numerical Experiments

4.1 Direct and Iterative methods

To assess the effectiveness of the proposed methods, we will consider a toy regression problem. In particular, we will try to approximate the function

$$f(x, y) = \sin\left(\alpha(x + y)\right), \tag{4.1}$$

with $\alpha \in \mathbb{R}$. The data points will be generated by sampling x, y from a uniform distribution in the interval $[0, 1]$, and the corresponding labels will be obtained by evaluating the function f at the sampled points.

In our first numerical experiment, we compare the accuracy of the direct and iterative methods. In particular, we consider the direct solvers based on the SVD and QR factorizations (*gelsd* and *gelsy* routines in LAPACK, respectively), and the iterative solvers LSQR and LSRN. We consider the problem (4.1), with $\alpha = 1$ and $N = 5000$ training points. We use one hidden layer with $M = 1000$ neurons and the $\tanh$ activation function. Given that, the matrix A has dimensions 5000×1001, including the bias term. In addition, we employ regularization, by using a cutoff value ϵ for small singular values of the matrix, i.e we consider as zero all singular values smaller than $\epsilon \times \sigma_{\max}$, where $\sigma_{\max}$ is the largest singular value of the matrix. The value of ϵ is set to 10^{-8}.

Figure 4.1, illustrates the residual norm per iteration for the aforementioned solvers. The dashed horizontal lines represent direct solvers, while the solid lines represent iterative solvers. First, we observe the LSQR suffers from slow convergence, which is expected given the high condition number of the matrix. Specifically, the condition number was numerically estimated to be approximately 10^{31}, indicating that the matrix is effectively rank deficient. The LSRN algorithm though, converges rapidly, since the preconditioning step ensures that the system to be solved is well-conditioned. We also notice that its accuracy is comparable to the direct methods, with the QR-based solver being slightly more accurate than the SVD-based solver.

The next step is to compare the execution time of the solvers. In Table 4.1, we provide the average execution time, along with the test and train Mean Absolute Error (MAE) for each method. We notice that the SVD-based solver is the fastest, followed by the LSRN algorithm. The QR-based solver is the slowest, but it provides the most accurate solution. However, the deviations in both the execution time and the accuracy are relatively small. Given that direct methods are impractical for large-scale problems that exceed available memory, LSRN emerges as a highly promising solver for such scenarios. It is worth noting

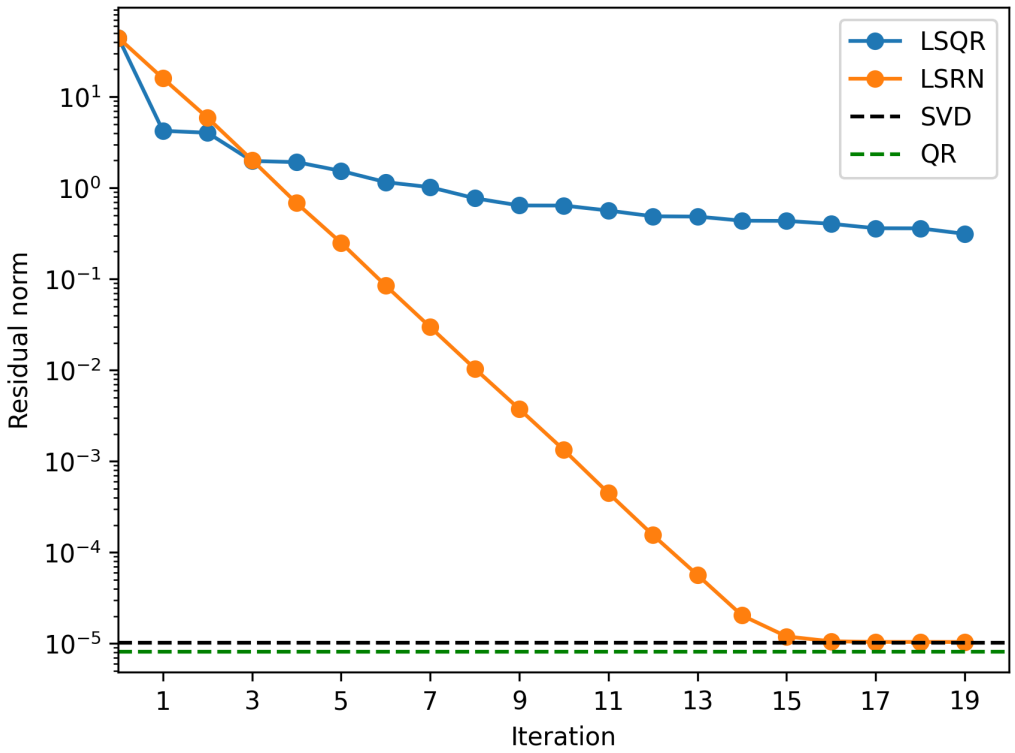


Figure 4.1: Comparison of the direct and iterative solvers for the regression problem (4.1). The dashed lines represent the direct solvers, while the solid lines represent the iterative solvers. It is evident that the randomized preconditioning enables the rapid convergence of the LSRN algorithm, with accuracy comparable to the direct methods.

here, that we implemented the LSRN algorithm in Python 3, following the original Python 2 implementation provided by Meng et al. [15]. However, our code is not fully optimized, so there is room for further improvements in execution time.

Method	Time (s)	Test MAE	Train MAE
SVD	0.84 ± 0.09	1.29×10^{-7}	1.18×10^{-7}
QR	1.45 ± 0.06	1.10×10^{-7}	1.01×10^{-7}
LSRN	1.09 ± 0.03	1.30×10^{-7}	1.19×10^{-7}

Table 4.1: Comparison of the standard LAPACK solvers with the LSRN algorithm. The results are averaged over 10 runs. We observe that all solvers provide accurate solutions, with comparable execution times.

To further investigate the scalability of LSRN, we consider a more realistic problem, the

classification of handwritten digits from the MNIST dataset [5]. This dataset consists of 70000 images of handwritten digits, with 28×28 pixels each. We randomly sampled 63000 images for training and 7000 for testing, while we varied the number of neurons in the hidden layer from 1000 to 20000. As before, we used the $\tanh$ activation function and regularization, with $\epsilon = 10^{-5}$. The main difference with the previous experiments is that the matrix A was not stored in memory. Instead, it was stored in the disk, and the matrix-vector and matrix-matrix products were computed on the fly, using memory-mapped files.

In Figure 4.2, we show the misclassification rate in the training and test sets, with respect to the number of neurons in the hidden layer. We see that that with an increasing number of neurons, the accuracy of the model improves significantly, with a misclassification rate of approximately 0.2% in the training set for 20000 neurons. It is worth mentioning that our primary interest lies in the training set accuracy, as our main objective is to obtain an accurate solution to the optimization problem, rather than focusing on the generalization performance of the model Nonetheless, the model's performance on the test set is also quite good, with a misclassification rate of approximately 2% for 15000 neurons.

As previously mentioned, the LSRN algorithm is particularly well-suited for strongly over-determined or under-determined problems. As a final step, we examine the effect of the ratio N/M on the algorithm's performance, in comparison to the standard SVD-based solver. Figure 4.3 presents the ratio of the run time of the LSRN algorithm to the SVD-based solver is shown, with respect to N/M. It is observed that the SVD-based solver consistently outpeforms the LSRN algorithm, though the performance gap narrows as the ratio N/M increases. This behavior is expected, as the LSRN algorithm is designed for cases where $N \gg M$ or $N \ll M$. Nevertheless, even for ratios close to 1, LSRN remains competitive, with the execution time being approximately 2 times slower than the SVD-based solver.

It is important to note that when the matrix is almost square, memory limitations may hinder the use of LSRN due to the construction of the projected matrix $\tilde{A}$ during the pre-conditioning step. For the default value of $\gamma = 2$, the dimensions of $\tilde{A}$ are $2M \times M$, potenially making it twice as large as the original matrix A. Additionally, the computation of the SVD of $\tilde{A}$ can be more challenging than that of the original matrix A, rendering the preconditioning step inefficient, as it becomes more costly than directly solving the system.

4.2 Split and Solve

In the next set of experiments, we will investigate the effectiveness of the Split and Solve approach. As described in the previous section, the main idea is to decompose the matrix A into smaller submatrices, and solve the corresponding subproblems one by one.

As a first step, we will examine the accuracy of the method, i.e the effect of the number of splits n on the accuracy of the solution. Again, we consider the same setup as before, with the function f defined in (4.1), $N = 5000$ training points, and $M = 1000$ neurons in the hidden layer. The solver used for the subproblems is the SVD-based solver, with regularization parameter $\epsilon = 10^{-8}$. In Figure 4.4, we present both the test and train MAE

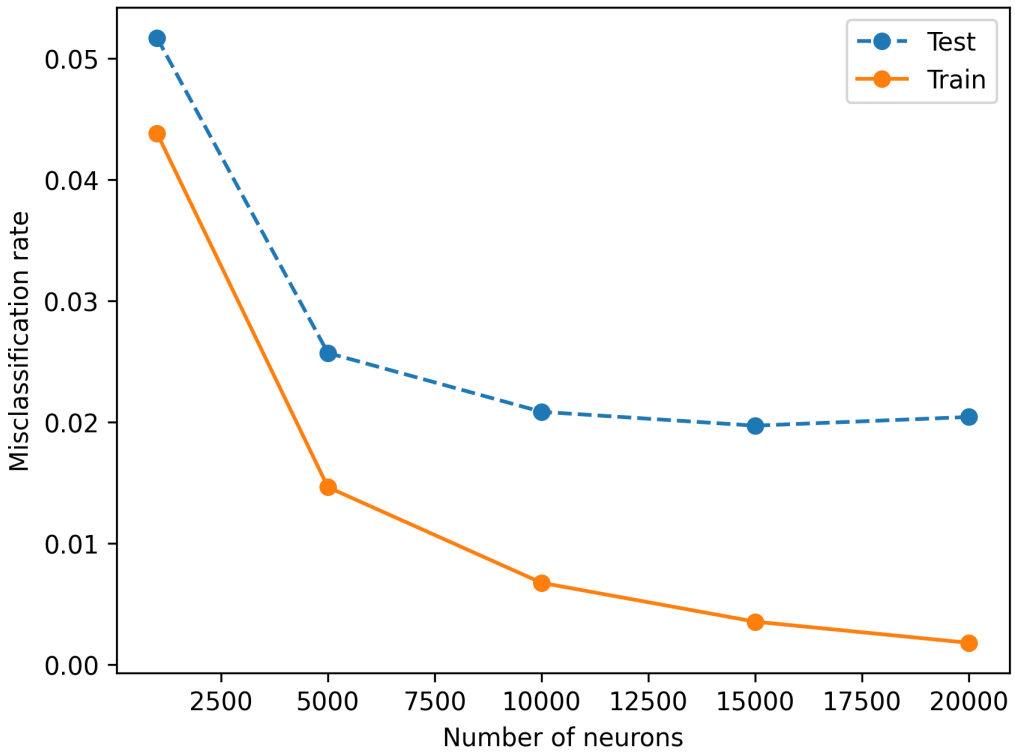


Figure 4.2: Misclassification rate with respect ot the number of neurons in the hidden layer, for the MNIST dataset. The orange line represents the training set, while the blue line represents the test set. As expected, the training accuracy improves with the number of neurons.

with respect to the number of splits. Interestingly, we observe that for small values of n, the accuracy of the method is even better than solving the full problem directly. However, as n increases, the accuracy gradually decreases, with $n = 9$ splits showing essentially the same MAE as the standard approach.

Additionally, in Figure 4.5, we show the execution time of the method with respect to n. Since the execution time decreases with the number of splits, this approach could be very promising for large-scale problems, especially for small n, where the accuracy is comparable to the standard approach. However, we should note here that, at least with the current formulation, the method is inherently sequential, since the solution of each subproblem depends on the previous one.

To further elucidate the reasons behind the effectiveness of the approach, we will repeat the same experiment, but with different strategies for splitting the matrix A. As described in the previous section, the splitting of A ultimately depends on the splitting of the weight matrix W of the hidden layer. In particular, we will consider the following strategies:

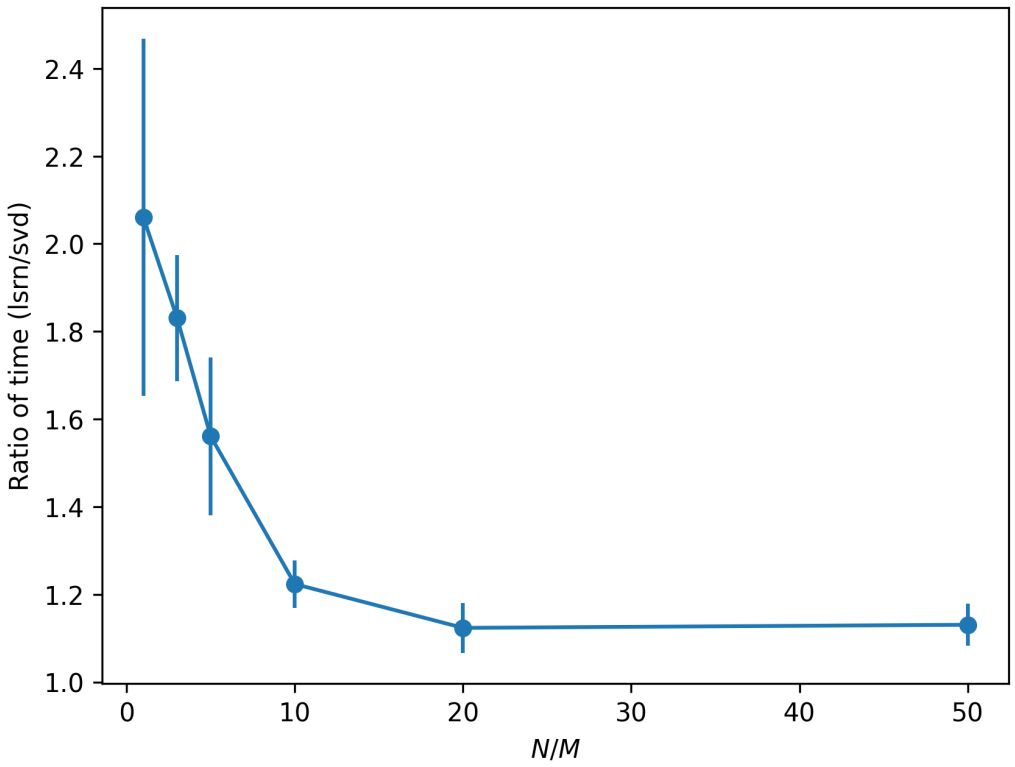


Figure 4.3: Ratio of the run time of the LSRN algorithm to the SVD-based solver, with respect to the ratio N/M. As N/M increases, the execution times become more comparable, though the SVD solver consistently remains faster.

(i) *Ordered splitting:* The weight matrix is split into k submatrices, following the prefined order of the columns, i.e without any reordering. (ii) *Sorted splitting:* The columns of the weight matrix are sorted based on their norm, and then grouped into k submatrices. This is the strategy used in the previous experiment. (iii) *Random splitting:* The columns of the weight matrix are randomly permuted, and then grouped into k submatrices.

In Figure 4.6, we show the test and train MAE with respect to the number of splits, for the different strategies. We see that the accuracy is increased only for the sorted splitting strategy, while the other strategies lead to a gradual decrease, with some fluctuations. This indicates that the ordering of the columns is an important factor for the effectiveness of the method. However, we should note that the sorting of the columns can be computationally expensive, especially for large-scale problems. Since the MAE of the ordered splitting is comparable to the standard approach, there are scenarios where this strategy could be a promising alternative.

As a next step, we will investigate how the parameter α in the function f affects the

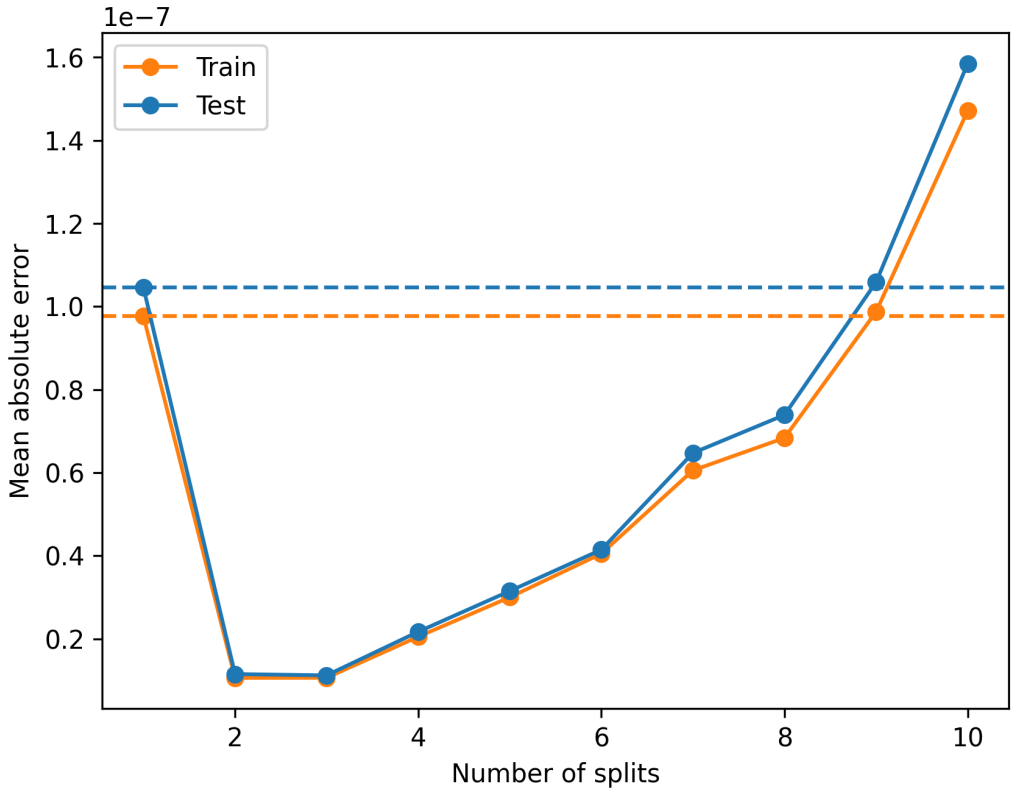


Figure 4.4: MAE with respect to the number of splits for the Split and Solve approach. The orange line represents the training set, while the blue line represents the test set. The dashed horizontal line represents the accuracy of the standard approach. An important observation is that for a small number of splits, the method is more accurate than the standard approach.

behavior of the Split and Solve approach. In Figure 4.7(a) we show the test and train MAE with respect to the number of splits, for different values of α, with $M = 1000$ neurons in the hidden layer. As expected, as α increases, the accuracy of the method decreases, since the function to be approximated becomes more complex. An interesting observation is that the phenomenon of increased accuracy for a small number of splits is dependent on the value of α. More specifically, for $\alpha = 5, 10$, we do not see any improvement in the accuracy, compared to the standard approach. To further investigate this, we repeat the same experiment, but with $M = 10000$ neurons in the hidden layer, as shown in Figure 4.7(b). In this case, for $\alpha = 5$ but not for $\alpha = 10$, we observe the same phenomenon. We conclude that this behavior is dependent on the interplay between the complexity of the function to be approximated and the approximation power of the model.

As a final experiment, we will consider the LSRN algorithm as the solver for the sub-

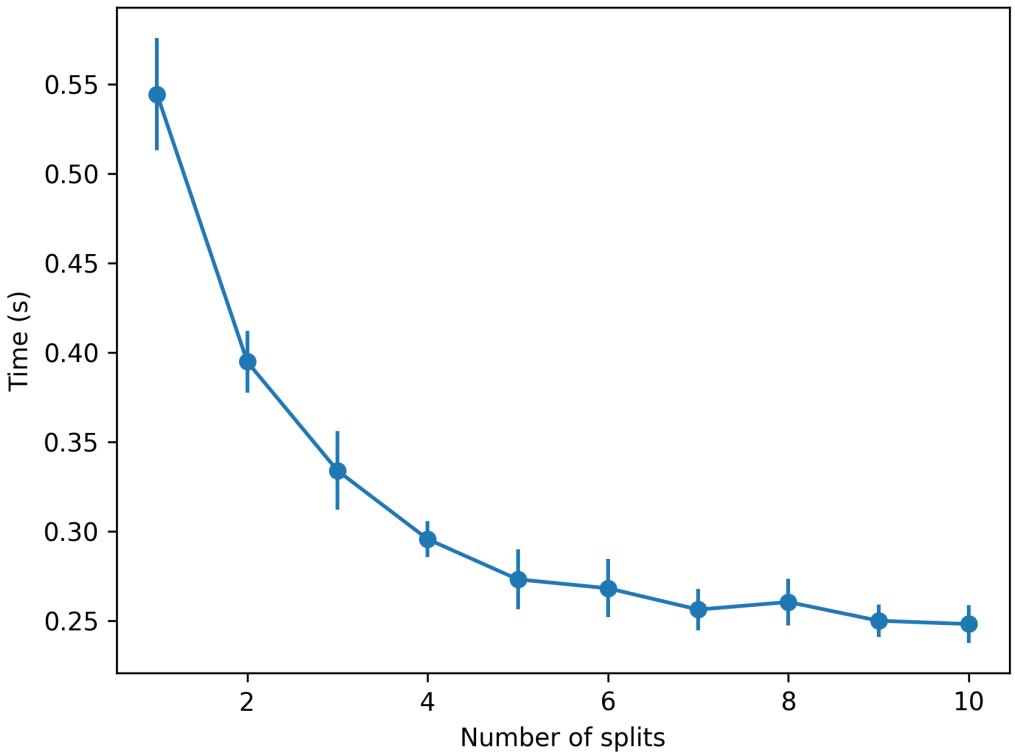


Figure 4.5: Execution time of the Split and Solve approach with respect to the number of splits. It is clear that the execution time decreases with the number of splits, with significant improvements for a small number of splits.

problems, and compare its performance with the SVD-based solver. In Figure 4.8, we show the test and train MAE with respect to the number of splits, for the two solvers, for different values of α. The hidden layer has $M = 1000$ neurons. The solutions obtained by the two solvers are almost identical, with a few fluctuations. Since SVD tends to be the fastest solver, it seems reasonable to use it as a default option. However, LSRN could be useful, in case we want to keep n small, and even the subproblems do not fit into memory.

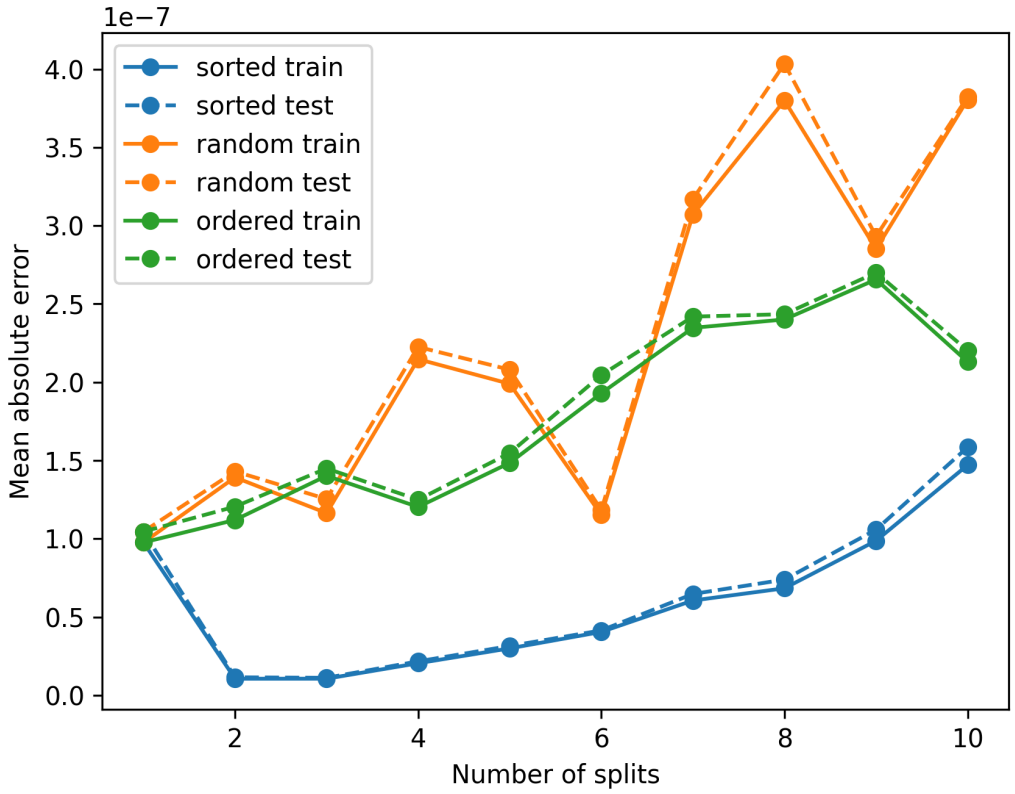
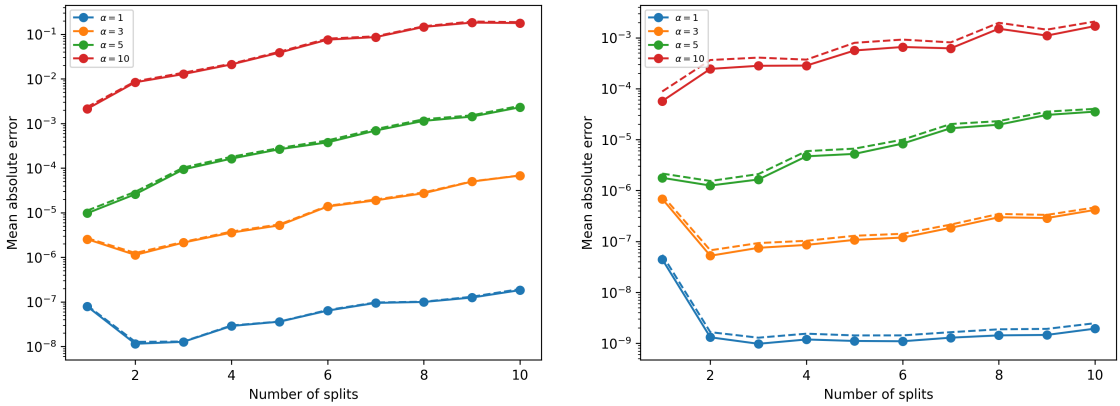


Figure 4.6: MAE with respect to the number of splits for the Split and Solve approach, for different splitting strategies, as described in the text. The solid lines represent the training set, while the dashed lines represent the test set. We see that the increase in accuracy depends on the splitting strategy, with the sorted splitting strategy being the most effective.



(a) $M = 1000$ (b) $M = 10000$

Figure 4.7: MAE with respect to the number of splits for the Split and Solve approach, for (a) $M = 1000$ and (b) $M = 10000$ neurons in the hidden layer. The different curves correspond to different values of α, with the solid lines representing the training set, and the dashed lines representing the test set. We notice that the phenomenon of increased accuracy for a few splits, is affected by both the value of α and the approximation power of the model.

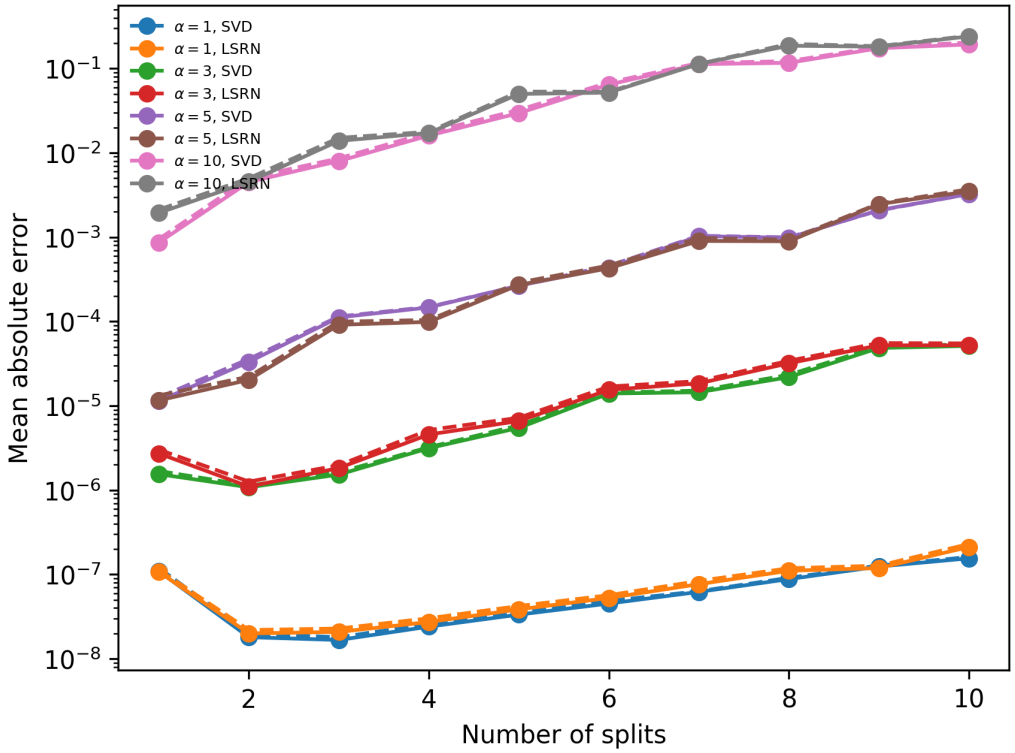


Figure 4.8: MAE with respect to the number of splits, for different values of α and different solvers. The solid lines represent the training set, while the dashed lines represent the test set. We conclude that the solutions obtained by the two solvers are almost identical, with some expected fluctuations.

5 Conclusion

In this Guided Research, we explored numerical methods for solving large-scale dense and ill-conditioned least squares problems, specifically in the context of the SWIM algorithm. We proposed two approaches, the LSRN algorithm and the Split and Solve method, and demonstrated their effectiveness. The LSRN algorithm, in particular, is a versatile solver, with the potential to scale in high-performance computing environments, involving multiple nodes. In constrant, the Split and Solve method is a more specialized approach, well-suited for small to medium-scale problems.

A potential area for future work is scaling up these methods to tackle larger problems by leveraging supercomputing resources. This would require careful optimization of the algorithms to align with the specific hardware architecture. Additionally, a promising avenue is the use of Graphics Processing Units (GPUs), which have demonstrated high efficiency in solving numerical linear algebra problems.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al. *LAPACK users' guide*. SIAM, 1999.

[2] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[3] Å. Björck. *Numerical methods for least squares problems*. SIAM, 2024.

[4] E. L. Bolager, I. Burak, C. Datar, Q. Sun, and F. Dietrich. Sampling weights of deep neural networks. *Advances in Neural Information Processing Systems*, 36, 2024.

[5] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[6] G. H. Golub and R. S. Varga. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order richardson iterative methods. *Numerische Mathematik*, 3(1):157–168, 1961.

[7] I. Goodfellow. *Deep Learning*. MIT Press, 2016.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[9] M. R. Hestenes, E. Stiefel, et al. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS Washington, DC, 1952.

[10] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. In *2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541)*, volume 2, pages 985–990. Ieee, 2004.

[11] A. A. Khan, A. A. Laghari, and S. A. Awan. Machine learning in computer vision: a review. *EAI Endorsed Transactions on Scalable Information Systems*, 8(32):e4–e4, 2021.

[12] D. P. Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[14] M. W. Mahoney et al. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

[15] X. Meng, M. A. Saunders, and M. W. Mahoney. Lsrn: A parallel iterative solver for strongly over-or underdetermined systems. *SIAM Journal on Scientific Computing*, 36 (2):C95–C118, 2014.

[16] R. Murray, J. Demmel, M. Mahoney, N. Erichson, M. Melnichenko, O. Malik, L. Grigori, P. Luszczek, M. Dereziński, M. Lopes, et al. Randomized numerical linear algebra: A perspective on the field with an eye to software (2023). *DOI: https://doi.org/10.48550/arXiv*, 2302.

[17] T. P. Nagarhalli, V. Vaze, and N. Rana. Impact of machine learning in natural language processing: A review. In *2021 third international conference on intelligent communication technologies and virtual mobile networks (ICICV)*, pages 1529–1534. IEEE, 2021.

[18] C. C. Paige and M. A. Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software (TOMS)*, 8(1):43–71, 1982.

[19] A. Rahimi and B. Recht. Uniform approximation of functions with random bases. In *2008 46th annual allerton conference on communication, control, and computing*, pages 555–561. IEEE, 2008.

[20] J. R. Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain. 1994.

[21] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[22] X. Zhang, L. Wang, J. Helwig, Y. Luo, C. Fu, Y. Xie, M. Liu, Y. Lin, Z. Xu, K. Yan, et al. Artificial intelligence for science in quantum, atomistic, and continuum systems. *arXiv preprint arXiv:2307.08423*, 2023.