Anderson Acceleration For Bioinformatics-Based Machine Learning

Sarwan Ali^{1,†}, Prakash Chourasia^{1,†} and Murray Patterson^{1,*}

¹Georgia State University, Atlanta, USA

Abstract

Anderson acceleration (AA) is a well-known method for accelerating the convergence of iterative algorithms with applications in various fields, including deep learning and optimization. Despite its popularity in these areas, the effectiveness of AA in classical machine learning classifiers has not been thoroughly studied. Tabular data, in particular, presents a unique challenge for deep learning models, and classical machine learning models are known to perform better in these scenarios. However, the convergence analysis of these models has received limited attention. To address this gap in research, we implement a support vector machine (SVM) classifier variant incorporating AA to speed up convergence. We evaluate the performance of our SVM with and without Anderson acceleration on several datasets from the biology domain and demonstrate that the use of AA significantly improves convergence and reduces the training loss as the number of iterations increases. Our findings provide a promising perspective on the potential of Anderson acceleration in training simple machine learning classifiers and underscore the importance of further research in this area. By showing the effectiveness of AA in this setting, we aim to inspire more studies that explore the applications of AA in classical machine learning.

Keywords

Anderson Acceleration, SVM, Sequence Analysis

1. Introduction

Anderson acceleration is a method that can be used to enhance the convergence of gradient descent algorithms. Based on the difference between the current and prior weight vectors, a correction term is added to the weight vector updates at each iteration. When the gradients are changing quickly, or the optimization landscape is very non-convex, this correction term can aid in reducing oscillations and speeding convergence. Consider the optimization issue as a trajectory in the weight space, where the weight vector reflects the position at each iteration, to appreciate this concept better. Without Anderson acceleration, the gradients at each location alone control the trajectory of the optimization process. While Anderson acceleration can smooth out the trajectory and minimize oscillations, the trajectory is also affected by the difference between the current and prior weight vectors.

Solving the convex problem in finding gradient descent is a typical problem in optimization. Newton's methods use the inverse Hessian matrix [1] to accelerate gradient descent, and they are successful in achieving a faster rate of convergence compared to gradient descent or accelerated gradient descent, but it is very expensive. By utilizing knowledge of the curvature of the loss function

KDH@IJCAI'23: Knowledge Discovery From Healthcare Data, August 20, 2023, Macao, S.A.R

≦ sali85@student.gsu.edu (S. Ali); pchourasia1@student.gsu.edu

(P. Chourasia); mpatterson30@gsu.edu (M. Patterson)

O 2023 Copyright for this paper by its authors. Use permitted under Creative Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

landscape and quasi-Newton algorithms [2] that compute a low-rank approximation of the Hessian, it is feasible to accelerate the training of machine learning(ML) models. The approximate replacement matrix for the Hessian Inverse can be found using quasi-Newton methods, described in detail by authors in [3]. The alternative option is to use the existing data that is already available for the fixed point approach, such as natural gradient descent (NGD) [4] and extrapolation methods for vector sequences [5]. Approximation-based approaches such as popular techniques, including momentum acceleration methods [6], and stochastic Newton methods [2], can be used to calculate the approximate replacement matrix. However, as dimensionality and complexity rise, the benefits of these methods are outweighed by the computational expense. Due to the increased dimensionality and complexity of the data, evaluating the training loss function is getting increasingly expensive; consequently, fixed-point [7] approaches are more suited for hastening the training of ML models.

The machine learning (ML) community has become interested in extrapolation techniques like Anderson acceleration (AA) [5]. These techniques have been demonstrated to speed up the training of sequential deep learning models (DL) [8]. Since the optimization landscape for deep learning models can be more complicated to improve than simpler models, Anderson acceleration is typically more successful when used in optimization methods for deep learning [9]. Although it might not offer as big of a gain in terms of convergence speed compared to more complex models, Anderson acceleration may still be effective for optimizing simpler classical ML models.

^{*}Corresponding author.

[†]These authors contributed equally.

Ultimately, the precise characteristics of the optimization problem being solved will determine how effective Anderson acceleration is in each given scenario.

In this work, we propose a robust approach to perform Anderson acceleration (AA) to speed up the training of SVM classifier models for multi-dataset training from the domain of biological sequencing. We regularize AA by including it in the loss optimization of simple linear classifier models (SVMs) and classical ML training, in contrast to previous work in complex deep learning models. We numerically demonstrate the effectiveness of the proposed acceleration by comparing the training loss with an increasing number of iterations on different sets of biological sequences. The results show that using AA significantly improves convergence and efficiently accelerates the training of traditional ML models.

2. Related Work

Iterative optimization methods like gradient descent and its variants are widely used for training ML models, but convergence can be slow, especially for high-dimensional problems. Anderson acceleration (AA) is a technique for speeding up the convergence of these methods by exploiting the geometry of the search space. It was first introduced by Anderson [10] as a way to accelerate the convergence of the conjugate gradient method and has since been applied to a variety of optimization techniques, such as Newton's method [11], stochastic gradient descent [12], and the Nelder-Mead simplex algorithm [13].

In recent years, there has been a growing interest in using Anderson acceleration for training deep neural networks, where it has been applied to a variety of tasks, such as image classification [14], natural language processing and reinforcement learning [15]. Anderson acceleration is particularly well-suited for deep learning problems, where it has been shown to improve convergence and generalization performance [16, 17]. A method to estimate a sparse generalized linear model with convex or non-convex separable penalties using Anderson acceleration is also proposed in [17]. In these approaches, Anderson acceleration has been shown to improve convergence and generalization performance compared to traditional optimization methods, such as gradient descent. In addition, it has also been applied to logistic regression [18] and other ML models.

However, despite these advances, Anderson acceleration has not been widely applied to classical machine learning classifiers, such as support vector machines (SVM), despite the potential for improved convergence rates. This is particularly relevant for tabular data, where classical machine learning classifiers are widely used. The limited exploration of Anderson acceleration in classical machine learning classifiers is surprising, given that

the technique is effective in improving convergence in other optimization problems [10, 11, 12, 13].

Another area where Anderson acceleration has shown promise is in training sparse models, such as sparse coding and dictionary learning [19]. In these applications, Anderson acceleration effectively improves convergence and achieves sparsity, an essential consideration in many machine-learning models.

In recent years, researchers have also explored the use of Anderson acceleration in the training of generative adversarial networks (GANs) [20]. In these applications, Anderson acceleration has been shown to improve convergence and stability and to produce high-quality synthesized images.

Finally, it's worth noting that Anderson acceleration has also been applied to the training of robust models that are robust to outlier examples and to adversarial attacks [21]. In these applications, Anderson acceleration effectively improves the robustness of machine learning models and defends against adversarial attacks.

3. Proposed Approach

This section first discusses the algorithm we use for the proposed method. Later, we discuss the theoretical understanding of Anderson Acceleration and the assumptions considered.

Anderson Acceleration (AA) attempts to make greater use of previous data than the fixed-point iteration, which only takes the most recent iteration to produce a new estimate, $y_{k+1} = g(y_k)$. The proposed method's algorithmic pseudocode is provided in Algorithm 1, and the model training flow chart is shown in Figure 1. For model training, given the feature embedding X (or ϕ) made from SARS-CoV-2 sequences and its lineage (variants) as labels Y, the first step involves the embedding generation using the methods discussed in Section 3.2, the feature vector generated and the labels for the sequences are then supplied to the algorithm. In the algorithm, firstly, the weight vector is initialized with random values (1 \times length of sequence). We then initialize the iterLoss and gradient values (lines 2-3 in Algorithm 1) for each iteration. Afterward, for each input sample X and its label y, we predict using weight vector \vec{w} (line 5 in Algorithm 1), the predicted value is normalized, and the gradient is updated (lines 5 and 6 in Algorithm 1). Sample loss is updated, and the iteration loss list iterLoss is maintained (lines 8 to 9 in Algorithm 1). After every sample is processed, the gradient is averaged out, and weight history is maintained for the iteration (lines 11 and 12 in Algorithm 1), also shown in Figure 1-e. Anderson acceleration is used to update the weight vector from the third iteration since we need at least two weight histories. The difference between the last two weight histories is computed and is multiplied with Anderson factor α as shown in lines 14 and 15 in Algorithm 1, also shown in Figure Figure 1-ii. The loss and accuracy for the iteration are saved, and the next iteration is performed to do the same steps. Finally, after all iterations, the loss list is returned for the given input feature vectors. The loss for each Iteration is captured and argued to be the better option for faster convergence using Anderson Acceleration.

3.1. Anderson Acceleration

One way to formally prove the convergence of Anderson acceleration is to use the concept of "linear convergence", which refers to the rate at which the optimization process approaches the optimal solution. Specifically, we can show that under certain conditions, the Anderson acceleration optimization process converges linearly, meaning that the error decreases by a constant factor at each iteration. This contrasts standard gradient descent, which converges at a sublinear rate (e.g., the error decreases by a factor less than 1 at each iteration).

To prove this result, we can start by considering the optimization problem in the form of a series of updates to the weight vector, where the update at each iteration is given by:

$$w_{t+1} = w_t - \alpha \nabla f(w_t) \tag{1}$$

where w_t is weight vector at iteration t, α is the learning rate, and $\nabla f(w_t)$ is the gradient of the objective function f at w_t . Now, we can add the Anderson acceleration term to the update, resulting in:

$$w_{t+1} = w_t + \alpha(w_t - w_{t-1}) - \alpha \nabla f(w_t)$$
 (2)

Next, we can define the error at each iteration as:

$$e_t = w_t - w^* \tag{3}$$

where w^* is the optimal weight vector. Now, we can substitute the expression for the update into the expression for the error and rearrange it to get:

$$e_{t+1} = (1 - \alpha)e_t + \alpha e_{t-1} - \alpha \nabla f(w_t) \tag{4}$$

where we have used the fact that $w^* = w_t - \nabla f(w_t)$. Now, we can define the "damping factor" as:

$$\delta = 1 - \alpha \tag{5}$$

and rewrite the expression for the error as:

$$e_{t+1} = \delta e_t + (1 - \delta)e_{t-1} \tag{6}$$

This expression has the form of a weighted average, where the weight of the current error is given by δ , and the weight of the previous error is given by $1 - \delta$. Now, we can make the following assumptions:

 The objective function f is continuously differentiable and has a Lipschitz continuous gradient, i.e., there exists a constant L such that

$$|\nabla f(x) - \nabla f(y)| \le L|x - y| \tag{7}$$

for all x, y.

- 2. The objective function f is bounded below, i.e., there exists a constant f_{\min} such that $f(x) \ge f_{\min}$ for all x.
- The optimization algorithm is using a fixed step size α, and the sequence of points x_k generated by the algorithm satisfies

$$|x_{k+1} - x_k| \le R \tag{8}$$

for some constant R and all k.

These assumptions are typically made in the analysis of gradient descent algorithms. They allow us to establish certain convergence properties of the algorithm. Specifically, under these assumptions, it can be shown that the sequence of points generated by gradient descent with Anderson acceleration converges to a stationary point (a point where the gradient is zero) of the objective function f at a rate of O(1/k), where k is the iteration number. This convergence rate is faster than $O(1/k^2)$ achieved by plain gradient descent without Anderson acceleration.

Intuitively, Anderson acceleration can be thought of as a way to incorporate information from past iterations into the current iteration to improve the convergence rate of the optimization algorithm. This is achieved using a weighted combination of the current gradient and the difference between the current and previous iterates. The weights are chosen such that the resulting update direction better approximates the true gradient at the current iterate, leading to faster convergence.

To compute the loss, we use "cross-entropy loss" using the following expression:

Cross Entropy Loss =
$$-sum(y \times log(yPred + 1e - 10))$$
 (9)

where y is the true label, and yPred is the predicted label. The 1e-10 is added to the yPred to avoid the log of zero, which will cause an infinity error. The negative sign ensures the optimization problem is formulated as a minimization problem (hence, our loss can be negative).

The flowchart for training with Anderson Acceleration(AA) is shown in Figure 1. We provide the Feature vectors X (or ϕ) as input along with the labels Y. Few parameter initializations are required, such as the number of iterations and the weight vector initialized with random values. Anderson acceleration factor α , for which we tried several values to study its impact and select the best value. An empty list for loss is also shown in Figure 1-b. For a given number of iterations, we process the samples

to compute the gradient and loss for the sample. The gradient is averaged out, and we update the weight using Anderson Acceleration for that iteration, also shown in Figure 1-ii. The process is repeated for the given number of iterations.

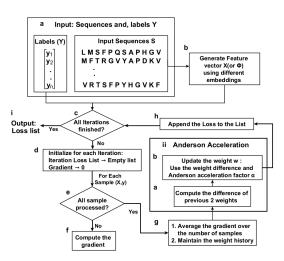


Figure 1: Flow chart for Anderson Acceleration training. Given the input data (X,y), we First, predict with initial weight w; this w is updated using SGD and modified with AA in consecutive steps to compute the loss and accuracy for the prediction.

Input: Embedded training data X, target labels y, no. of iterations iter, Anderson

Algorithm 1 SVM With Anderson Acceleration

```
Initialization
        \vec{w} \leftarrow \text{randn}(X.\text{shape}[1])
                                                                                 ▷ Initial random weight
       loss \leftarrow []
                                                                                                 ⊳ Empty list
       wHistory \leftarrow \begin{bmatrix} \overrightarrow{w} \end{bmatrix}
                                                                                 ▶ Weight vector history
      for i \in \{1, 2, \dots, iter\} do
                                                                                                 ⊳ Initial loss
           iterLoss \leftarrow 0
  3:
            grad \leftarrow \mathbf{0}
for (x, y) \in (X, y) do
                                                                                                   ⊳ Gradient
  4:
                 yPred \leftarrow w \cdot x
                                                                                         ⊳ Predict the class
                 yPred \leftarrow \frac{yPred}{\text{sum}(yPred)}
                                                                             ⊳ Normalize the prediction
  7:
                 grad \leftarrow grad + y - yPred

sampleLoss - = sum(y \times log(yPred))
  8:
  9:
                 iterLoss+ = sampleLoss
            end for
11:
            grad \leftarrow \frac{grad}{len(X)}
                                                                    ⊳ Avg the grad over total samples
12:
            wHistorv + = [\overrightarrow{w}]
                                                                             ⊳ Add undated w to history
13:
            if len(wHistory) > 2 then
14:
                 diff \leftarrow wHistory[i-1] - wHistory[i-2]
                  \vec{w} \leftarrow \vec{w} + \alpha \times diff + grad
                                                                                     ⊳ Update wusing AA
15
16:
17:
            else
                 \vec{w} \leftarrow \vec{w} + grad
                                                                             18
            end if
            loss.append([\frac{iterLoss}{len(X)}])
19:
                                                                                               ⊳ Append loss
20: end for
21: return loss
```

3.2. Embedding Methods

We employ the three representation learning techniques below to convert the biological sequences into lowdimensional embeddings.

3.2.1. Spike2Vec [22]

This technique offers numerical embedding of the supplied input spike sequences to facilitate the use of ML models. Initially, it produces k-mers of the supplied spike sequence because *k*-mers are known to maintain the sequence's ordering information. For a sequence of length N, the total number of k-mers produced is N - k + 1. For every particular sequence, k-mers is a collection of (contiguous) amino acids (also known as mers) of length k. (also called nGram in the NLP domain). To convert the k-mers alphabetical data into a numerical representation, the Spike2Vec computes the frequency vector based on *k*-mers. This vector comprises the counts of each *k*-mer in the sequence. A fixed-length feature vector is then made using the generated k-mers and their frequencies in a sequence. The character alphabet $\boldsymbol{\Sigma}$ and the length of the *k*-mers are used to calculate the length of this feature vector, which is $|\Sigma|^k$.

3.2.2. Minimizer [23]

A minimizer, also known as an *m*-mer, is the substring of consecutive letters (amino acids) of length m from a given k-mer that is lexicographically the smallest in both forward and backward order of the *k*-mer, where m < kand is fixed. The repetition of k-mers in a long sequence, which increases computing and storage costs, is one of the fundamental issues with k-mers. Minimizers could be used to get rid of this redundant information. The lexicographically lowest *m*-mer from both the forward and backward *k*-mers is then used to calculate a minimizer. Two parameters (k,m) are provided to the minimizer. The k-mer's length is k, where k = 9 in our example, and the *m*-mer's size is m, where m = 3. It extracts k-mers of a sequence given to it. Then, it calculates a corresponding m-mer for each k-mer (minimizer). A fixed-length feature vector is then made using the generated minimizers m-mers and their frequencies in a sequence. The character alphabet Σ and the length of the *m*-mers are used to calculate the length of this feature vector, which is $|\Sigma|^m$.

3.2.3. Spaced k-mer [24]

The performance of sequence classification is significantly impacted by the size and sparsity of feature vectors for sequences based on k-mers frequencies. The idea of employing non-contiguous length k sub-sequences (g-mers), proposed by spaced k-mers, to create compact feature vectors with reduced sparsity and size. It first

computed g-mers using a spike sequence as input. We calculate k-mers, where kg, from those g-mers. To conduct the trials, we used k=4 and g=9. The gap's dimensions are determined by g-k. However, this approach still involves bin scanning, which is computationally expensive and generates very high dimensional feature representation. We took 500 Principle components by applying PCA [25] for high dimensional embeddings (feature vector length > 1000).

4. Experimental Evaluation

To perform evaluation, we use datasets including Genome and Host. The details are as follows:

4.1. Dataset Statistics

4.1.1. Genome Dataset

Using the well-known and widely used database of SARS-CoV-2, GISAID [26], we retrieve the full-length nucleotide sequences of the coronavirus. Our dataset includes the COVID-19 variant information and 8220 nucleotide sequences. In our sample, there are 41 different Lineages altogether. The goal is to classify the sequences and predict the Lineage it belongs to.

4.1.2. Host Dataset

The National Institute of Allergy and Infectious Disease (NIAID) Virus Pathogen Database, Investigation Resource (ViPR) [27], and GISAID was used to retrieve the Spike protein sequences from a collection of spike sequences from several clades of the Coronaviridae family, along with details about the hosts that each spike sequence has infected. The hostname is used as the class label in our classification tasks for this dataset. It displays the distribution of the dataset across the various host types (grouped by family).

4.2. Evaluation Metrics

For performance evaluation of SVM without and with Anderson acceleration, we use cross-entropy loss. The cross-entropy loss, also known as the negative log-likelihood loss, is commonly used in supervised learning problems with categorical targets. The cross-entropy loss for a single sample can be expressed mathematically as follows: $L = -\log\left(\frac{e^{f_{j_1}}}{\sum_{j=1}^K e^{f_j}}\right), \text{ where } f_{y_i} \text{ is the predicted score for the correct class and } K \text{ is the number of classes. The cross-entropy loss is averaged over the entire training set to obtain the final objective function optimized during training.}$

The cross-entropy loss penalizes the predicted scores for the incorrect classes and rewards the predicted score for the correct class. During training, the goal is to minimize the cross-entropy loss so that the predicted scores for the correct class are as high as possible compared to those for incorrect classes.

5. Results And Discussion

In this section, we report results comparison without and with Anderson acceleration using cross-entropy loss for different biological sequence datasets.

5.1. Results For Genome Data

The results for genome data using all embedding methods are reported in Figure 2 for the best value of Anderson Acceleration (AA) factor α . We use cross-validation to get the best value for α ranging from (0, 0.1, 0.2, ..., 1.0) for respective embeddings, where 0 implies no AA and 1.0 shows maximum AA. For Spike2Vec embedding, we can observe that although cross-entropy loss without Anderson acceleration is smaller with fewer iterations, as we increase the iterations, the loss increases too. On the other hand, the loss does not increase significantly while using Anderson acceleration in SVM. Moreover, with AA, the loss started to converge after 300 iterations, which is almost half compared to the loss convergence without AA (i.e., \approx 600 iterations). For Minimizer-based embedding, although we can observe more fluctuation in loss compared to Spike2Vec, the loss (and convergence) is less when SVM is used along with AA. Similarly, the behavior of spaced *k*-mers-based embedding differs from both Spike2Vec and Minimizer-based embedding. Although we can see an overall increasing trend in loss with an increasing number of iterations, the SVM with AA loss is lower than without AA when the number of iterations increases. Overall, it is evident from all three embedding results that the loss with AA is less than the loss without AA for different embedding methods as we increase the number of iterations, showing the significance of using AA for the training of SVM.

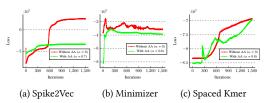


Figure 2: Loss plots with AA (green line) and without AA (red line) for **Genome data**. The x-axis shows the increasing number of iterations, while the y-axis shows the cross entropy loss. The figure is best seen in color.

5.2. Results For Host Data

The results for host data using all embedding methods are reported in Figure 3 for the best value of Anderson Acceleration (AA) factor α . We use cross-validation to get the best value for α ranging from (0, 0.1, 0.2, ..., 1.0) for respective embeddings, where 0 implies no AA and 1.0 shows maximum AA. For Spike2Vec-based embedding, the behavior is not different from the same embedding in the case of Genome data. Although SVM without and with Anderson acceleration converges very fast (i.e., in < 100 iterations), the cross entropy loss with AA is smaller than SVM without AA. We observed some improvement in the SVM without AA in the Minimizer and Spaced k-mers-based embedding methods. However, when the number of iterations is smaller, we can observe some fluctuation in the cross-entropy loss for SVM without AA, compared to the smooth loss curve for SVM with AA, showing its significance in efficient training of the SVM classifier.

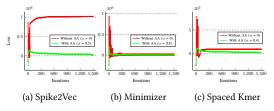


Figure 3: Loss plots with AA (green line) and without AA (red line) for **Host data**. The x-axis shows the increasing number of iterations, while the y-axis shows the cross entropy loss. The figure is best seen in color.

Overall, we can observe from the results of all two datasets and all three embeddings that SVM with Anderson acceleration converges faster and shows less crossentropy loss than SVM without Anderson acceleration. This behavior shows the effectiveness of SVM when used with Anderson acceleration in terms of training time and performance when applied in a real-world setting (as we show its performance on a 3 real-world set of biological sequences). We believe this study can open more opportunities for researchers to explore further the applications of Anderson acceleration for other datasets (e.g., from other domains such as finance and images) when applied to other machine learning models such as perceptron. These future studies can help us further understand and improve the predictive performance of classical ML models when applied in a real-world (biological) setting.

6. Conclusion

In summary, this study provides a novel use of Anderson acceleration in the bioinformatics area of support vector

machine (SVM) classifier. Our experiments on several sequence-based bioinformatics datasets show that Anderson acceleration results in a considerable decrease in training loss and improved convergence compared to the standard SVM. In the future, we will investigate more traditional linear classifier models, such as the Perceptron, and bigger biological data to assess their scalability and resilience. Moreover, evaluating the robustness and generalizability of the proposed Anderson acceleration method is also an interesting future extension.

References

- D. Goldfarb, Y. Ren, A. Bahamou, Practical quasinewton methods for training deep neural networks, Advances in Neural Information Processing Systems 33 (2020) 2386–2396.
- [2] Q. Tong, G. Liang, X. Cai, C. Zhu, J. Bi, Asynchronous parallel stochastic quasi-newton methods, Parallel computing 101 (2021) 102721.
- [3] D. F. Shanno, Conditioning of quasi-newton methods for function minimization, Mathematics of computation 24 (1970) 647–656.
- [4] H. Naganuma, R. Yokota, A performance improvement approach for second-order optimization in large mini-batch training, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, 2019, pp. 696–703.
- [5] C. Brezinski, S. Cipolla, M. Redivo-Zaglia, Y. Saad, Shanks and anderson-type acceleration techniques for systems of nonlinear equations, IMA Journal of Numerical Analysis 42 (2022) 3058–3093.
- [6] N. Qian, On the momentum term in gradient descent learning algorithms, Neural networks 12 (1999) 145–151.
- [7] H. F. Walker, P. Ni, Anderson acceleration for fixed-point iterations, SIAM Journal on Numerical Analysis 49 (2011) 1715–1735.
- [8] W. Shi, S. Song, H. Wu, Y.-C. Hsu, C. Wu, G. Huang, Regularized anderson acceleration for off-policy deep reinforcement learning, Advances in Neural Information Processing Systems 32 (2019).
- [9] N. Keskar, et al., On large-batch training for deep learning: Generalization gap and sharp minima, arXiv preprint arXiv:1609.04836 (2016).
- [10] D. G. Anderson, Iterative procedures for nonlinear integral equations, Journal of the ACM (JACM) 12 (1965) 547–560.
- [11] H. An, X. Jia, H. F. Walker, Anderson acceleration and application to the three-temperature energy equations, Journal of Computational Physics 347 (2017) 1–19.
- [12] V. Mai, M. Johansson, Anderson acceleration of

- proximal gradient methods, in: International Conference on Machine Learning, 2020, pp. 6620–6629.
- [13] R. R. Barton, J. S. Ivey Jr, Modifications of the Nelder-Mead simplex method for stochastic simulation response optimization, Technical Report, 1991.
- [14] M. L. Pasini, J. Yin, V. Reshniak, M. K. Stoyanov, Anderson acceleration for distributed training of deep learning models, in: SoutheastCon 2022, 2022, pp. 289–295.
- [15] G. Zuo, S. Huang, J. Li, D. Gong, Offline reinforcement learning with anderson acceleration for robotic tasks, Applied Intelligence (2022) 1–14.
- [16] M. L. Pasini, J. Yin, V. Reshniak, M. Stoyanov, Stable anderson acceleration for deep learning, arXiv preprint arXiv:2110.14813 (2021).
- [17] Q. Bertrand, Q. Klopfenstein, P.-A. Bannier, G. Gidel, M. Massias, Beyond 11: Faster and better sparse models with skglm, arXiv preprint arXiv:2204.07826 (2022).
- [18] Q. Bertrand, M. Massias, Anderson acceleration of coordinate descent, in: International Conference on Artificial Intelligence and Statistics, 2021, pp. 1288–1296.
- [19] P. Rodriguez, Computational assessment of the anderson and nesterov acceleration methods for large scale proximal gradient problems, in: Symposium on Image, Signal Processing and Artificial Vision (STSIVA), 2021, pp. 1–5.
- [20] H. He, S. Zhao, Y. Xi, C. J. Ho, Y. Saad, Solve minimax optimization by anderson acceleration, in: International Conference on Learning Representations, 2022.
- [21] M. Garstka, M. Cannon, P. Goulart, Safeguarded anderson acceleration for parametric nonexpansive operators, in: European Control Conference (ECC), 2022, pp. 435–440.
- [22] S. Ali, M. Patterson, Spike2vec: An efficient and scalable embedding approach for covid-19 spike sequences, in: IEEE International Conference on Big Data (Big Data), 2021, pp. 1533–1540.
- [23] M. Roberts, W. Hayes, B. Hunt, S. Mount, J. Yorke, Reducing storage req for biological sequence comparison, Bioinformatics 20 (2004) 3363–3369.
- [24] R. Singh, A. Sekhon, et al., Gakco: a fast gapped k-mer string kernel using counting, in: Joint ECML and Knowledge Discovery in Databases, 2017, pp. 356–373.
- [25] S. Wold, K. Esbensen, P. Geladi, Principal component analysis, Chemometrics and intelligent laboratory systems 2 (1987) 37–52.
- [26] GISAID Website, https://www.gisaid.org/, 2021. [Online; accessed 17-October-2022].
- [27] B. E. Pickett, E. L. Sadat, Y. Zhang, J. M. Noronha, R. B. Squires, V. Hunt, M. Liu, S. Kumar, S. Zaremba, Z. Gu, et al., Vipr: an open bioinformatics database

and analysis resource for virology research, Nucleic acids research 40 (2012) D593–D598.