Assignment - 1

Adrija Bera 210071 | **Amay Raj** 210116 | **Amandeep Upadhyay** 210114 | **Aniket Sen** 210134 | **Jefrin Jo W** 210471

Tanuja Kaleli 211099

Abstract

To learn a linear model that is D-dimensional but S-sparse that can accurately predict the responses on the test set.

1 Mathematical Interpretation

Given:

- D: Number of CDUs in the PUF ($D = 2048$)
- S: Number of active CDUs in the PUF ($S = 512$)
- $\phi(c)[i] = p_i$: Mapping function $\phi(c)$ maps a D-bit $0/1$-valued challenge vector c to a D-dimensional feature vector $\phi(c)$, where $\phi(c)[i]$ represents the delay p_i of the CDU at index i.

We want to derive a sparse linear model with a weight vector $w \in \mathbb{R}^D$ that satisfies the following conditions:

1. The weight vector w has at most S non-zero coordinates (sparse weight vector).
2. For any challenge $c \in \{0,1\}^D$, the expression $w^T\phi(c)$ gives the correct response.

To achieve this, let's proceed with the derivation:

Step 1: Mapping Function ϕ

The mapping function ϕ maps a D-bit $0/1$-valued challenge vector c to a D-dimensional feature vector $\phi(c)$. In this case, $\phi(c)[i] = p_i$ represents the delay of the CDU at index i.

Step 2: Sparse Linear Model

We want to find a weight vector $w \in \mathbb{R}^D$ that satisfies the given conditions:

- $\|w\|_0 \leq S$ (w has at most S non-zero coordinates)
- $w^T\phi(c)$ gives the correct response for all challenges $c \in \{0,1\}^D$.

Step 3: Deriving the Sparse Linear Model

Let A be the set of indices of the active CDUs, where $|A| = S$. We can construct the weight vector w such that $w[i] = 1$ for $i \in A$, and $w[i] = 0$ for $i \notin A$. In other words, the weight vector w has non-zero entries only at the indices corresponding to the active CDUs.

Preprint. Under review.

For any challenge vector $c \in \{0,1\}^D$, we have:

$$w^T \phi(c) = \sum_{i=1}^{D} w[i] \cdot \phi(c)[i] = \sum_{i=1}^{D} w[i] \cdot p_i$$

Since $w[i] = 1$ for $i \in A$ (active CDUs) and $w[i] = 0$ for $i \notin A$, we can simplify the expression:

$$w^T \phi(c) = \sum_{i \in A} p_i$$

The summation $\sum_{i \in A} p_i$ represents the total delay incurred by the active CDUs, which is the correct response for the given challenge c.

Therefore, $w^T \phi(c)$ will give the correct response for any challenge $c \in \{0,1\}^D$, as it computes the total delay of the active CDUs.

Step 4: Sparsity of the Linear Model

To show that the weight vector w has at most S non-zero coordinates, we need to prove that $\|w\|_0 \leq S$.

Since $w[i] = 1$ for $i \in A$ (active CDUs) and $w[i] = 0$ for $i \notin A$, the number of non-zero coordinates in w is equal to the cardinality of A, which is S.

Hence, $\|w\|_0 \leq S$, and the weight vector w is sparse with at most S non-zero coordinates.

In conclusion, we have mathematically derived a sparse linear model for the Sparse CDU PUF, satisfying the conditions of having at most S non-zero coordinates.

2 Techniques used to solve the 512-Sparse PUF Problem

2.1 Naive Method

The Naive method involved obtaining a solution from the Least Square technique and applying a simple hard thresholding operation to generate the desired 512-sparse vector. However, this approach resulted in a significant increase in error when sparsifying the vector. Therefore, we decided not to utilize the Naive method.

2.2 Sub-Gradient Descent (LASSO Technique)

The Sub-gradient descend technique, which consists of three variants—Batch, Stochastic, and Mini-batch—was considered for solving the problem. The choice among these variants was based on the time required to reach the final solution. Through experimentation, we determined that a mini-batch size of 400, combined with a regularization constant of 0.35, produced a better 512-sparse vector compared to the Naive method. The Sub-gradient descend technique incorporated regularization to aid in feature selection. However, the resulting error reduction was not significant compared to the Naive method.

2.3 Projected Gradient Descent

Ultimately, we selected the projected gradient descent method as the preferred approach for solving the problem. This method employed L1 regularization to maintain ongoing feature selection. Instead of using the traditional gradient descent technique, we implemented a momentum method to expedite convergence to the solution. Additionally, we chose a mini-batch size of 1400 instead of 1600 to improve efficiency. The number of iterations served as the convergence criterion, and after careful analysis, we determined that 35 iterations yielded optimal results.

Initially, when running projected gradient descent without correction, the resulting error was smaller than that obtained using the LASSO technique. However, the improvement was not significant. Consequently, we introduced a correction method that yielded a remarkable mean absolute error of less than 0.05, representing a noteworthy achievement.

Increasing the number of iterations to 100 would result in an error in the range of 10^{-7}. However, the corresponding computation time was approximately 24 seconds. After implementing a mini-batch size of 1400 and setting number of iterations to be 35, we got MAE in the range of $2 \cdot 10^{-7}$. This was achieved in around 10 seconds.

3 Hyperparamater Tuning

"Selecting an optimal value for a hyperparameter is often considered an art rather than a science." So, there is no fixed algorithm to find out the best hyperparameters . Instead, it depends from one Machine learning project to another Machine learning project .

The hyperparameters used in our attempt were as follows:

1. Regularization constant (for Sub-gradient descent)
2. Initialization of the Response Vector
3. Step length
4. Mini-Batch size
5. Beta (Momentum parameter)
6. Number of Iterations

Our algorithm involved dividing the training data into 1000 subsets, with the remaining 600 samples reserved for testing. We trained the model using each hyperparameter value from the array and evaluated its performance based on the resulting best fit response vector (1 x 2048). The mean absolute error was calculated by testing the model on the test dataset for each hyperparameter value. The corresponding mean absolute errors were stored in an array called 'Error.' Finally, using a function from the numpy library, we determined the hyperparameter value that yielded the lowest mean absolute error.

3.1 Regularization Constant (for Sub-gradient descent)

To tune the regularization constant, one of the most effective methods is Bayesian optimization. However, since we had a predetermined range of values for the regularization constant, we opted to perform a simple grid search. We constructed an array consisting of 50 elements ranging from 0.1 to 5.0, with a step length of 0.1. For a mini-batch size of 400, the best regularization constant was 0.35.

3.2 Initialization of the Response Vector

For initializing the response vector, we based it on the solution obtained from the least square method. This approach ensured that the initial vector was closer to the actual solution compared to initializing it with a zero vector. Initializing the response vector with a zero vector resulted in a mean absolute error of 6.22 and took approximately 6 seconds to execute. However, by initializing the response vector with the least square solution, we achieved a mean absolute error of 0.03 within 3 seconds.

3.3 Step Length

For the step length hyperparameter, we employed a manual selection approach. While a step length of 0.01 often works well in many machine learning problems, it varies depending on the specific problem at hand. Selecting a step length that is too large may lead to non-convergence and Runtime overflow errors, which we encountered during our experiments. Therefore, we aimed to identify a step length that allowed convergence without causing overflow errors, while also considering the total execution time. We found that a step length of 0.5 was suitable for the LASSO technique implemented using the Sub-gradient descent technique.

In our final submission, we did not explicitly utilize the step length parameter, as it is typically associated with gradient descent methods. Instead, we employed the momentum method, which is an algorithm known for achieving faster convergence and incorporates a hyperparameter referred to as Beta.

3.4 Mini-Batch Size

We have implemented mini-batch to decrease the execution time. To determine the optimal mini-batch size, we employed a grid search technique. We constructed an array starting from 100 and incrementing by 100 up to 1500. We trained the model using a mini-batch size of 1400, with 200 samples reserved for testing. The methodology employed was similar to that used for tuning the regularization constant. It is worth noting that the optimal batch size was found to be 1400, which reduced noise in the training process. Although the time required for training with a batch size of 1400 was slightly longer compared to smaller batch sizes, the difference was not significant enough to outweigh the benefits.

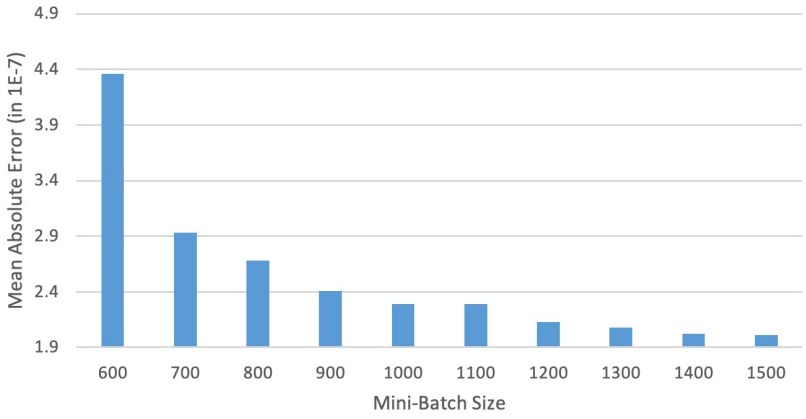

Figure 1: MAE vs Mini-Batch Size

3.5 Beta (Momentum parameter)

The momentum method helped accelerate the convergence compared to traditional gradient descent methods. We manually tuned the Beta parameter to a value of 0.85 to achieve the desired solution. We avoided selecting beta values such as 0.3 or 0.4 because they would reduce the contribution of the previous gradient during the current iteration, which is contrary to the intended purpose of the momentum method. Typically, higher values of beta are preferred, and some references consider $(1 - beta)$ as the step length in this algorithm. Through manual experimentation within the range of 0.70 to 1.00, we determined that 0.80 was the optimal value for our specific problem.

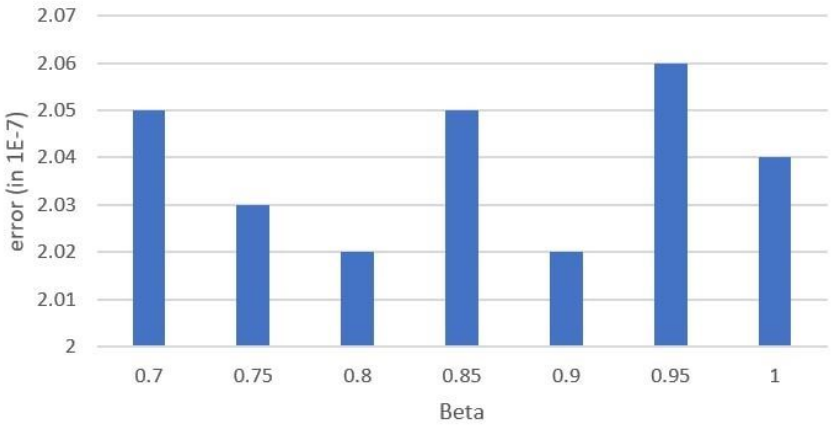

Figure 2: MAE vs Beta

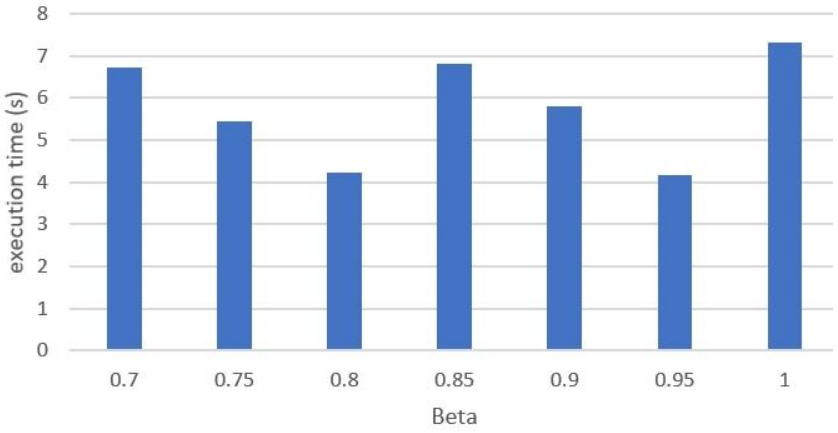

Figure 3: Execution Time vs Beta

3.6 Number of Iterations

Increasing the number of iterations would result in more time required to reach a solution. To determine an appropriate number of iterations, we initially experimented with 100 iterations and observed minimal difference in the error compared to 50 iterations, indicating that the solution had nearly converged by that point. We further reduced the number of iterations to 35, as we found this value to be sufficient for achieving high accuracy within a shorter timeframe.

4 Conclusion

Based on the histograms and fine-tuning of various hyperparameters, we have reached the following conclusions:

- Initializing the solution with the least square problem's result resulted in the least deviation.
- Among the tested batch sizes, 1400 yielded the least error while still maintaining reasonable execution time.
- Implementing the momentum method with a beta value of 0.8 provided a good balance between error reduction and execution time. Other values in the range of 0.70 to 1.00 were also considered, but 0.80 demonstrated the best trade-off.
- Iterations above 100 took too much time, while fewer than 10 iterations did not yield satisfactory results. By reducing the number of iterations, we determined that approximately 35 iterations were optimal.

Through the optimization process outlined above, our model achieved a mean absolute error of $2 \cdot 10^{-7}$ within an execution time of 10 seconds (subject to change between 9 and 12 seconds because mini-batch is used).

References

[1] https://www.kaggle.com/general/301545

[2] https://web.stanford.edu/class/ee392o/subgrad_method.pdf

[3] https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f