CS771 Assignment 1

D-Bot

Aarish Khan, Emaad Ahmed, Krishnansh Agarwal, Rishi Agarwal, Vansh Raj Sachan, Zoya Manjer

1 Mathematical Derivation showing how a Sparse CDU PUF can be Broken by a Single Sparse Linear Model

Let the challenge for an S-sparse CDU be represented by $c_i \in \{0, 1\}$. For a CDU PUF, the challenge can be represented as a vector $X_i \in \mathbb{R}^D$, with each value $c_i \in \{0, 1\}$, for a D-dimensional PUF. In our case, D = 2048. Furthermore, let all PUFs have a delay of $p_i \in \mathbb{R}/\mathbb{R}^-$

Since we are given an S-sparse PUF problem, let $s_i \in \{0, 1\}$ represent whether a CDU is active; s_i = 1 represents the CDU is active and vice-versa. Then the total delay, which is the desired correct response, can be given as $t = \sum s_i \cdot p_i \cdot c_i$

Since the prediction is linear with c_i, $\phi(c)$ is simply the vector $X_i \in \mathbb{R}^D$. Therefore the mapping $\phi : \{0, 1\}^D \mapsto \mathbb{R}^D$ is found by simply assigning the i^{th} dimensional value of the feature map, the value of c_i.

Expressing the correct response as

$$\mathbf{w}^T \phi(c) = \sum s_i \cdot p_i \cdot c_i$$
$$\Rightarrow \mathbf{w}^T = [s_1 \cdot p_1 \quad s_2 \cdot p_2 \quad s_3 \cdot p_3 \quad \cdots \quad s_n \cdot p_n]$$

Therefore, **w** = $\begin{bmatrix} s_1 \cdot p_1 \\ s_2 \cdot p_2 \\ s_3 \cdot p_3 \\ \cdots \\ s_n \cdot p_n \end{bmatrix}$ is the model.

Note that since the only S = 512 CDUs are non-sparse, number of indices $i \ \forall \ s_i \neq 0$ is equal to S = 512.

Now, $\mathbf{w}_i = s_i \cdot p_i \Rightarrow i \ \forall \ \mathbf{w}_i \neq 0 \leq S \Rightarrow ||\mathbf{w}||_0 \leq S$

Here **w**, as demonstrated above, is an S-sparse linear model that illustrates correct responses.

2 Code Used to Solve the Problem

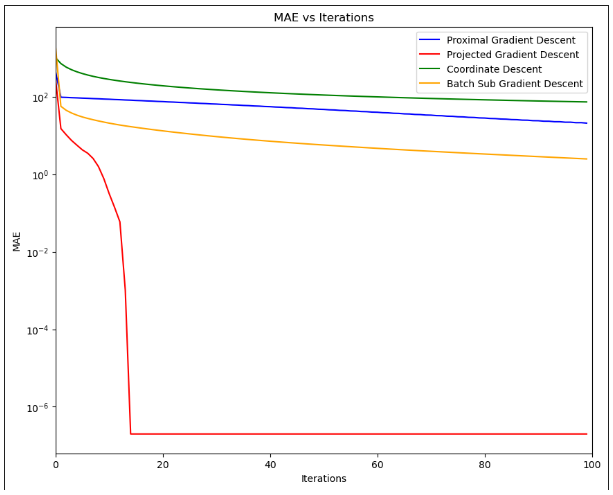
```
import numpy as np

def proj_func(w, sparse_dim):
    t = np.zeros_like(w)
    if sparse_dim < 1:
        return t
    else:
        ind = np.argsort(abs(w))[-sparse_dim:]
        t[ind] = w[ind]
        return t

def grad_f(w, x, y):
    return (np.dot(w,x) - y)*x/len(x)

def projective_gradient_descent(w0, learning_rate, num_iterations, x, y):
    w = np.copy(w0)
    for i in range(num_iterations):
        for j in range(1600):
            gradient = grad_f(w, x[j], y[j])
            w = w - learning_rate*gradient
        w = proj_func(w, 512)
        ind = np.nonzero(w)[0]
        x_proj = x[:, ind]
        sqr_w = np.array(np.linalg.lstsq(x_proj, y, rcond = None)[0])
        w[ind] = sqr_w
    return w

def my_fit( X_trn, y_trn ):
    w0 = np.array(np.linalg.lstsq(X_trn, y_trn, rcond = None)[0])
    model = projective_gradient_descent(w0, 0.70324444, 15, X_trn, y_trn)
    return model
```

3 Exploration of Methods: Comparative Analysis and Preferred Approach

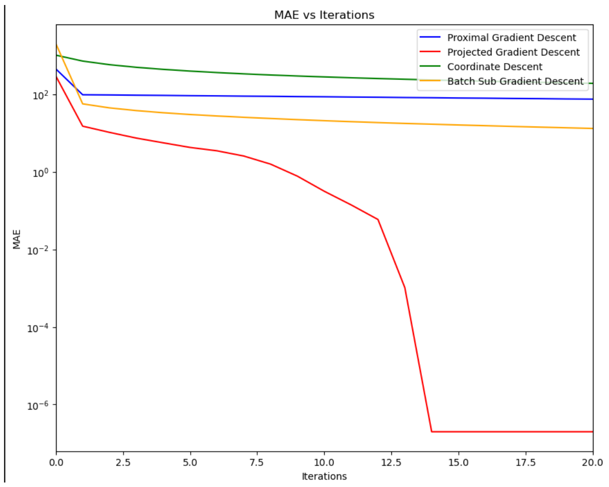
We trained models for the following methods:

- Batch Sub-gradient descent with L1 regularization:
  Train Time = 25s
  Best MAE = 0.0023
- Coordinate gradient descent with L1 regularization:
  Train Time = 7min
  Best MAE = 76
- Proximal gradient descent with L1 regularization:
  Train Time = 3min
  Best MAE = 0.11
- Projected gradient descent:
  Train Time = 7.6s
  Best MAE = 10^{-7}

The comparison between these methods is also visible in the following plots:



(a) MAE error for all models for 100 iterations



(b) MAE error for all models for 20 iterations

Figure 1: MAE vs Iterations

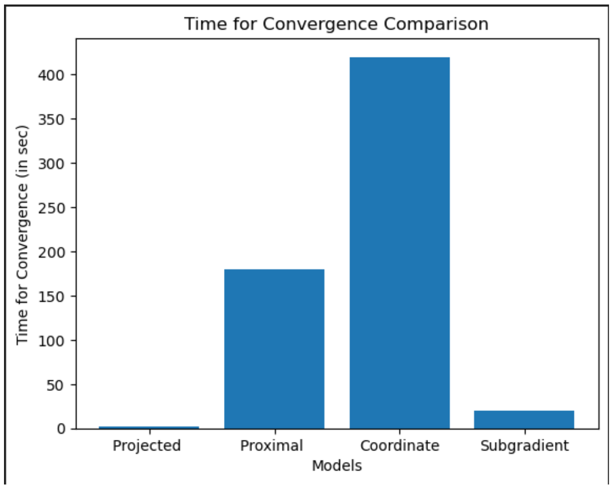


Figure 2: Time of convergence for different models

As is clear from the above comparisons, projected gradient descent outperformed other models in our case. So we selected projected gradient descent for our submission.
The sparse model converges faster with projected gradient descent because in each iteration we take the top 512 candidates after gradient descent step from the previous solution and then solve for minimum least square on these weights. It finds the optimum value for the 512 most promising weights in each iteration to converge quickly.

4 Optimisation of Hyperparameters

The only hyperparameter with the projected gradient descent approach was the learning rate(α). The metric for comparison was chosen to be mean absolute error. First we prepared a set of different $\alpha's$ that were first spaced out with large values that had differences of orders of magnitude, for example [0.01, 0.1 , 1, 10]. The lowest error were found at 0.1 and 1, which indicated a minima in this range. So again we splitted this range and searched for a smaller range for a better approximation of ideal learning rate. After repeating this process we finally achieved best performance for $\alpha = 0.7032$