Predicting the Output of Unpredictable Microelectronic Circuits

Xiaoyan Feng
xyan399@umd.edu
MSML 604 - PCS1
University of Maryland College Park
Master of Professional Studies
Machine Learning

Sushant Karki
skarki@umd.edu
MSML 604 - PCS1
University of Maryland College Park
Master of Professional Studies
Machine Learning

Justin Letwinsky
jletwin@umd.edu
MSML 604 - PCS1
University of Maryland College Park
Master of Professional Studies
Machine Learning

Chandramani
cm05@umd.edu
MSML 604 - PCS1
University of Maryland College Park
Master of Professional Studies
Machine Learning

***Abstract* - The microelectronic circuits explored in this project report are known as delay-based Physical Unclonable Functions (PUFs), which have been proposed as a security solution used in cryptography, key generation and identification, field-programmable gate arrays (FPGA), privacy protection, and internet of things (IoT) applications. Taking advantage of slight variations during the manufacturing process, these PUFs produce an unpredictable output response to a distinct input value referred to as a "challenge vector". Although PUFs have been known for their unclonability and light-weight design, more recently, vulnerabilities have been exploited by machine learning attacks. Inside the microelectronic circuits, there are two delay paths, as well as a comparator that determines which circuitry path is faster (resulting in an output of either 0 or 1). Using optimization theory, the output of the microelectronic circuit can be predicted to a reasonable degree of accuracy. This paper explores various machine learning binary classification algorithms, formulated as optimization problems, in order to accurately predict the output of the comparator.**

I. Introduction

In the ever-evolving environment of today's digital world, securing data and devices has become a top priority. Cryptography has been the traditional approach to achieve this goal, however recent advancements made in computing technology have created new challenges to previous techniques used for digital security. In 2001, the concept of the Physical Unclonable Function was introduced and has since emerged as a promising solution.

PUFs rely on distinct features of their physical microstructure, which are a result of random physical factors introduced during production. These factors are both unpredictable and uncontrollable during the manufacturing process, which to their advantage, makes it nearly impossible to replicate the microelectronic devices (e.g. transistors) within them. Instead of using a single cryptographic key, PUFs use these manufacturing variations to produce an unpredictable output, or response, to each distinct input value, which is a Boolean vector referred to as the "challenge vector" $\boldsymbol{c}$. A specific challenge vector $\boldsymbol{c}$ and its corresponding response is called the challenge-response pair (CRP).

The structure of a delay-based PUFs circuit is represented in Figure 1, which shows the starting point of both delay paths on the left. It is at time 0, when the signal value initially switches from logic 0 to logic 1. From here, a multiplexer chooses its output from these logic inputs. An arbiter at the end of the circuit serves as a delay comparator, and r represents the response value where $r = 1$ if the upper signal has the least delay, and r = 0 if the lower signal results in the least delay.

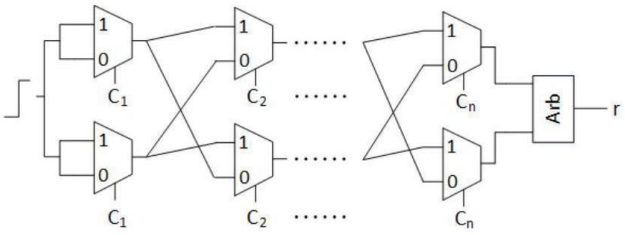


Figure 1: Delay-based PUF multiplexer structure

Each multiplexer "stage" consists of two multiplexers in parallel, as well as a select bit c_n that is shared between the two multiplexers. Also, each stage has two input bits (determined by the shared select bit c_n) and two output bits. The challenge vector $\boldsymbol{c}$ determines the value of the select bit (either 0 or 1), which in turn determines the input bits for that specific multiplexer stage. Figure 2 depicts the crossing/non-crossing scenarios that could occur in each stage, with each stage acting like a switch box. For example, if the select bit of a stage (represented by n) is 1, then both parallel multiplexers in that stage choose input 1. If input 1 is chosen, the upper output of the previous stage (n-1) goes into the upper input of the current stage n, and the lower output of the previous stage (n-1) goes into the lower input of the current stage n. Conversely, if the select bit is 0 (both multiplexers choose input 0), the upper output of the previous stage (n-1) goes into the lower input of the current stage n, and the lower output of the previous stage (n-1) goes into the upper input of the current stage n. In other words, the inputs of the current stage cross.

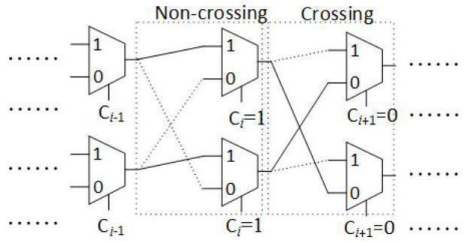


Figure 2: Crossing/Non-Crossing Scenarios of Each Stage

As a result of unavoidable variations in the manufacturing process, the slight differences among the multiplexers result in carrying a delay (from input to output) that is much more significant than any other component within the microelectronic circuit. Since the resulting delay is unique to each multiplexer, the microstructure of the PUF establishes its own specific identity.

II. Background on Mathematical Model

The PUF is able to generate two paths in the end; we denote the difference, the accumulative delay of upper path subtract this of the lower one, between these two paths at the i^{th} stage by Δ_i. There are two methods to calculate Δ_i . First, directly compute the difference when you have the accumulative delays of the upper path and the lower path respectively, for example, you have already known that the upper path's additive delay is x_1 while the lower's is x_2 , then Δ_i equals to x_1- x_2. Secondly, utilize the inductive equation shown below:

$$\Delta_i = \begin{cases} \Delta_{i-1} + \delta_i^1 & \text{if} \quad c_i = 1 \\ -\Delta_{i-1} + \delta_i^0 & \text{if} \quad c_i = 0 \end{cases} \text{for } i = 1, 2, \ldots, n$$

where the delay of each stage is denoted by δ_{i0} and δ_{i1}, which represents crossing and non-crossing case at the i^{th} stage respectively, and n shows the total stages in the PUF, $\Delta_0 = 0$, and $c = (c_1, c_2, \ldots, c_n)$ is the challenge vector.

By deriving this inductive equation, the cumulative delay after the n^{th} stage is :

$$\Delta_n = \Phi \cdot \omega \qquad (1)$$

where $\Phi = (\phi_1, \phi_2, \ldots, \phi_{n+1})$ is the feature vector. It switches the challenge vector into -1 and 1 by the equation below:

$$\phi_i = \phi_i(\mathbf{c}) = \prod_{j=i}^{n} (2c_j - 1) \quad \text{for } i = 1, 2, \ldots, n, \ \phi_{n+1} = 1$$

In equation (1), $\omega = (\omega_1, \omega_2, \ldots, \omega_{n+1})$ is the manufacturing variation vector for the input vector c and is named by weight vector. To briefly clarify, we could roughly treat predicting ω as the main task for this project. The equations of ω listed as below:

$$\omega_i = \frac{\delta_{i-1}^0 + \delta_{i-1}^1 + \delta_i^0 - \delta_i^1}{2} \text{ for } i = 2, 3, \ldots, n,$$
$$\omega_1 = \frac{\delta_1^0 - \delta_1^1}{2}, \ \omega_{n+1} = \frac{\delta_n^0 + \delta_n^1}{2}$$

R, the response bit, has two results and determines the different categories, one is the upper path is faster than the lower one, another is the opposite outcome:

$$r = \begin{cases} 1 \text{ if } \Delta_n > 0 \\ 0 \text{ otherwise} \end{cases}$$

Overall, the calculation process could be presented as the Figure 3 below.

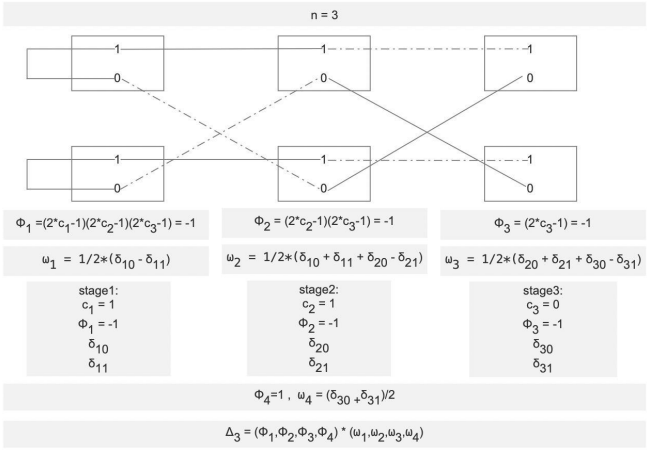


Figure 3: Calculation Process

III. Formulation

To best estimate the weights ω, multiple algorithms were explored in order to formulate optimization problems that could efficiently and accurately address the binary classification of r to a targeted success rate of 99% or better. The primary optimization problems used were: Logistic Regression (from scratch), Neural Networks (using PyTorch), Neural Networks (from scratch), and Support Vector Machines (using Scikit-Learn's Support Vector Classification).

A. Logistic Regression (from Scratch) Formulation

The reason for selecting Logistic Regression as one of the primary algorithms is due to the fact that it is useful in predicting a binary outcome, in this case $r = 1$ or $r = 0$. Since logistic regression deals with binary classification, it relies on the sigmoid function (Figure 4) to confine the outcome to be within the range [0, 1].

$$h(z) = \frac{1}{1+e^{-z}}$$

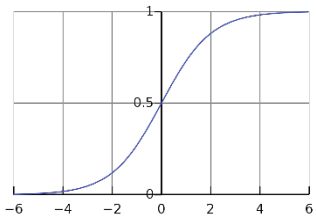


Figure 4: Sigmoid Function

The sigmoid function crosses the y-axis at the value of 0.5, which will be deemed the classification threshold for $h(z)$, where $z = \Phi^T \omega$. The predicted end result was determined from the following classifier:

$$r_i = 1 \; if \; \Phi^T \omega > 0 \; and \; 0 \; if \; \Phi^T \omega \leq 0$$

The prediction classifier implies that the posterior probabilities of the classes can be represented as follows:

$$log \; \frac{Pr(r_i = 1 \mid X = \omega)}{Pr(r_i = 0 \mid X = \omega)} = \Phi^T \omega$$

In exploring optimization setups, we initially optimized the problem through maximum log-likelihood (Figure 5), since l is a concave function.

$$l(\omega) = \sum_{i=1}^{m} r_i \; log \; (h(\Phi^T \omega_i)) + (1 - r_i) \log (1 - h(\Phi^T \omega_i))$$

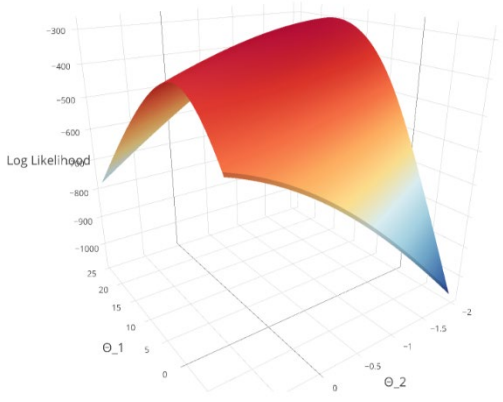


Figure 5: Maximum Log-Likelihood

Ultimately, in order to quantify the error in our weight predictions, the log-loss cost function was used in the actual implementation to measure the difference between the prediction probabilities and the actual binary outcomes.

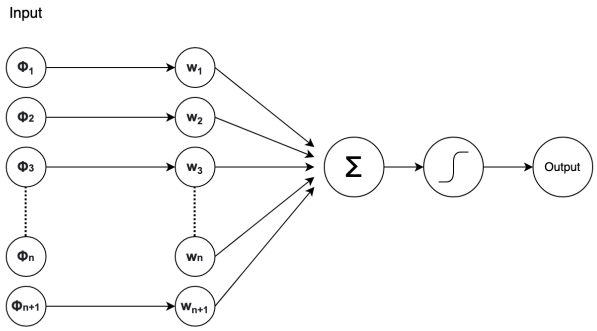
$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[r^{(i)} log \left(h_\omega(\Delta_n{}^{(i)}) \right) + (1 - r^{(i)}) log \left(1 - h_\omega(\Delta_n{}^{(i)}) \right) \right]$$

Since the log-loss cost function is a convex optimization problem, a single global minimum can be found.

B. Neural Network (using PyTorch) Formulation

The Physical Unclonable Function (PUF) can be viewed as a black-box, whereby the input select bits are processed through one end, and an output of either 0 or 1 is generated from the other end. The inner workings of the PUF are unknown, much like the architecture of a perceptron, a single-layer neural network, that utilizes a single neuron to map inputs to outputs.

To accurately model the PUF as a perceptron, we used the following architecture:



The input to the perceptron is the vector ϕ where each element of ϕ is given by:

$$\phi_i = \phi_i(c) = \prod_{j=i}^{n} (2c_j - 1) \; for \; i = 1,2, \ldots, n$$

$$\phi_{n+1} = 1$$

The perceptron has n+1 weights, one for each input dimension. We set the bias in the layer to be zero because since we explicitly have $\phi_{n+1} = 1$ in our input, our w_{n+1} weight will act as our bias. With this formulation, the output from our forward pass will look like the following, which is equivalent to the mathematical model of the PUF :

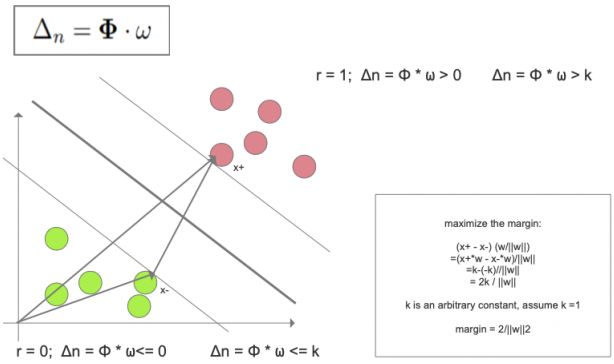
$$Logits = \Phi \cdot \omega$$

Although the PUF uses a step function for classification, we use sigmoid activation for our network as it is easier to differentiate. For our loss function, we use the BCEWithLogitsLoss function that is available in pytorch. One thing to note is that since BCEWithLogitsLoss in PyTorch combines the sigmoid layer with the binary cross entropy loss, we do not explicitly activate our output with a sigmoid layer and just return the logits from the network.

C. Support Vector Machines (using Scikit-Learn) Formulation

Support Vector Machine is feasible for coping with binary and multi-class classification problems. As the response bit of

the PUF problem is binary, it can be treated as a binary classification problem. The PUF has linearity regarding to the function of getting Δ_n as well as the linearly separable dataset. Therefore, this unpredictable problem could be solved by SVC. The figure below is the mathematical process of deriving the loss function through SVC method in the PUF.



By using Scikit-Learn's Support Vector Classification, weights are the output, which is the best weight vector for predicting r.

D. *Neural Network (from Scratch) Formulation*

A PUF is a device that can be used to generate a unique identifier based on its physical properties. It can be thought of as a black box: you put in some input bits, and it outputs a 0 or 1. The inner workings of the PUF are unknown, which makes it very secure.

The architecture of a PUF is very similar to that of a perceptron, a single-layer neural network with a single neuron. The input bits are fed into the neuron, which produces an output of 0 or 1. The weights of the connections between the input bits and the neuron are adjusted during training to produce the desired output.

We took this knowledge to our advantage and developed a neural network model of a perceptron with 65 weights, one for each of the 64 stages of the PUF and one for the last arbiter stage. The inputs based on the challenge vector are forwarded to the perceptron, which produces an output based on its inner weights.

The input to the perceptron is the vector ϕ of dimension n+1 where each element of ϕ is given by

$$\phi_i = \phi_i(c) = \prod_{j=i}^{n} (2c_j - 1) \; for \; i = 1,2, \dots, n$$
$$\phi_{n+1} = 1$$

The weights of the neural network are initialized randomly. During training, each weight is updated at each epoch using the backward propagation formula. The backward propagation formula computes the gradient of the loss function with respect to each of the weights using the chain rule, and the weights are updated in the opposite direction of the gradient. We use batch Gradient Descent for our optimization algorithm where we average the gradients of each weight at the end of each iteration (for the entire training dataset) and use the average gradient to then update the individual weights.

We used the sigmoid activation function as our output layer activation function which has a range of [0, 1], and it is differentiable at all points in its range. This makes it easier to compute the gradient of the loss function with respect to the weights. We used binary cross entropy as our loss function.

IV. NUMERICAL METHODS

A. *Logistic Regression (from Scratch) Numerical Method*

In regards to numerical methods, Python was used as the programming language of choice, however for Logistic Regression, we derived the gradient descent of the log-loss cost function by self-writing the coded equations from scratch (vice importing the "Logistic Regression" model from Scikit-Learn). Gradient descent served as the optimization algorithm to learn the best-fitting weight parameters. By taking the gradient of the log-loss cost function, using the expression in Figure X, we were able to produce accurate predictions of the arbiter response *r*.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{2m} \times \sum_{i=1}^{m} \left(h_\omega\left(\Delta_n{}^{(i)}\right) - y\right) \cdot \Delta_n{}_j^{(i)}$$

First the training set was generated using 10,000 as the number of training samples from the "puf_query" function. By differentiating the cost function, the weights were updated by using a learning rate step size of $\alpha = 0.1$, $\alpha = 0.01$, and $\alpha = 0.001$ for 1000 iterations (followed by 500 iterations) until convergence (Figure X). The experiment was then repeated with the same hyper-parameters, but with 5,000 as the number of training samples.

$$\omega_j^{(new)} = \omega_j^{(old)} - \alpha \frac{1}{2m} \times \sum_{i=1}^{m} \left(h_\omega\left(\Delta_n{}^{(i)}\right) - y\right) \cdot \Delta_n{}_j^{(i)}$$

$$\text{for } j = 0, 1, \dots, k$$

We used the newly learned parameters to predict the test data. After each trial was conducted, varying the number of training samples, learning rate, and number of iterations, the

results were recorded as summarized in Table X of Section V.

B. *Neural Network (using PyTorch) Numerical Method*

We modeled the perceptron by creating a single layer PyTorch module with 65 as our input dimension and 1 as the output dimension. We used Binary Cross Entropy (BCE) for our loss function which is given by:

$$Loss = -[(P_{true} \times log(P_{pred}) + (1 - P_{true}) \times log(1 - P_{pred})]$$

We tried both the Adam optimizer and SGD for optimization. We ended up using SGD as it converged significantly faster during our trials.

As in any neural network, the weights were updated by backpropagation (more information in the Neural Network from scratch section).

C. *Support Vector Machines (using Scikit Learn) Numerical Method*

For our SVM implementation, Python was the language of choice and we used the SVC class from the *scikit-learn package* to model our PUF.

Firstly, we used the provided *puf_query* function to generate our training dataset. Secondly, to find the best parameters for the SVM model, we used GridSearchCV to search for the best combination of hyperparameters (C and gamma). We found a training size of 5000 samples, with C=115 and gamma=0.01 to perform the best and thus, these parameters were used in our final model which reached an accuracy higher than our target of 99%.

D. *Neural Network (from Scratch) Numerical Method*

Our model works by passing a matrix of the input data ϕ through the perceptron. The perceptron does a matrix multiplication of the input matrix to its weights. A pointwise sigmoid activation is applied to the resulting matrix and the Binary Cross Entropy loss is calculated using the sigmoid values. The dimensions of the matrix at each step of the forward pass are as follows:

$$Dimension\ of\ \Phi\ =\ B\ X\ 65$$
$$Dimension\ of\ \omega\ =\ 65\ X\ 1$$
$$Dimension\ of\ \Phi \cdot \omega\ =\ B\ X\ 1$$
$$Dimension\ of\ Sigmoid(\Phi \cdot \omega)\ =\ B\ X\ 1$$
$$Dimension\ of\ BCELoss(Sigmoid)\ =\ B\ X\ 1$$
$$Dimension\ of \frac{\partial L}{\partial \omega} =\ B\ X\ 65$$
$$Dimension\ after\ taking\ Mean(\frac{\partial L}{\partial \omega})\ =\ 1\ X\ 65$$

where B = Batch size = Number of training samples in our case

At the end of the epoch, the mean of the gradients is used to update the weights. The update rule for the weights is as follows :

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w}$$

where:

- L is the loss function (Binary Cross Entropy Loss)
- y is the output of the neural network [0 and 1]
- h is the result of $\Phi \cdot \omega$
- w is the weight

The weights are updated using the following formula:

$$\omega \leftarrow \omega - \eta\ Mean(\frac{\partial L}{\partial \omega})$$

where η is the learning rate.

V. Experiment Results

A. *Logistic Regression (from Scratch) Results*

The results for the Logistic Regression optimization problem exceeded the target success rate, with a best accuracy result of 0.9981 and an effective training time of just 0.3281 seconds (with a training size of 10,000). The best training time result was 0.0469 and with an accuracy that still exceeded the target, resulting in 0.9927 (with a training size of 5,000). After each iteration in updating the weights, it was clear that the log-loss cost function was successful in descending to a global minimum (Figure 6). A summary of the overall results is shown in Table 1:

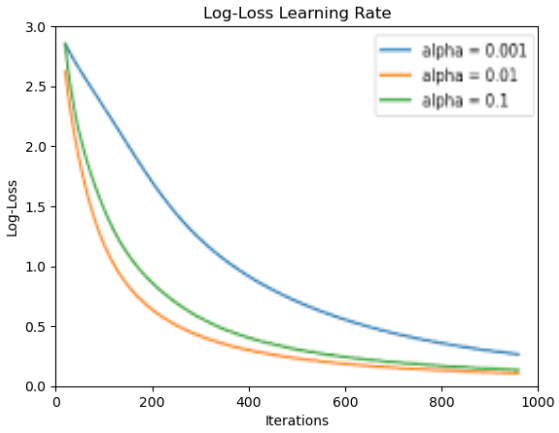


Figure 6: Log-Loss Learning Rate

Table 1: Logistic Regression Results

Training Size	Learning Rate (α)	No. of Iterations	Success Rate	Training Time
10,000	0.1	1,000	0.9981	0.3281 sec
10,000	0.1	500	0.9961	0.2510 sec
10,000	0.01	1,000	0.9945	0.4218 sec
10,000	0.01	500	0.9976	0.2656 sec
10,000	0.001	1,000	0.9958	0.5156 sec
10,000	0.001	500	0.9949	0.2511 sec
5,000	0.1	1,000	0.9948	0.2500 sec
5,000	0.1	500	0.9926	0.1875 sec
5,000	0.01	1,000	0.9933	0.3281 sec
5,000	0.01	500	0.9927	0.0469 sec
5,000	0.001	1,000	0.9923	0.4531 sec
5,000	0.001	500	0.9913	0.1094 sec

B. Neural Network (using PyTorch) Results

We experimented with various hyperparameters such as training size, learning rate, number of epochs, weight initialization of the neurons and choice of optimizers. Table 2 provides a summary of the combinations that gave the best results:

Table 2: Neural Network (using PyTorch Results):

Training Size	Learning Rate	Epochs	Weight Initialization	Success Rate	Training Time
15000	300	10	Uniform (-30, 30)	0.9983	5.3822
15000	200	10	Uniform (-20,20)	0.9975	4.96920
15000	250	10	Normal (0, 1)	0.9966	4.79360
10000	300	10	Uniform (-30, 30)	0.9965	4.58163
10000	250	10	Normal (0, 1)	0.9964	3.79617
5000	250	10	Normal (0, 1)	0.9936	2.75570

With the Adam optimizer, the network seemed to converge very slowly, and switching to SGD improved the training speed by an order of about 5. We found that we could increase the learning rate to a large number and still converge. Overall, a learning rate between 200~300 along with a weight initialization to a standard normal, or a uniform distribution along the interval [-30, 30] gave us a combination of the best speed and success rates. As expected, the success rates went up with more training samples, but the training time increased correspondingly as well. We found that with all the other parameters tuned as mentioned above, a training size of somewhere between 10000 to 15000 samples consistently gave us a success rate of over 99.5%. A training size of 5000 gave us a considerable reduction in training time, but also reduced our success rate to about 99.1% which is close to the minimum accuracy that we are aiming for.

C. Support Vector Machines (using Scikit-Learn) Results

SVC, a subset of SVM, has many hyperparameters, and we won't use kernels in this experiment since we are going to finish a binary classification. By adjusting hyperparameters like training size, C, and gamma, we acquire different success rates and training time shown in the Table 3.

Table 3: Support Vector Machines (using Scikit-Learn) Results

Training Size	C	Gamma	Success Rate	Training Time
5000	115	0.01	0.9953	1.6631
5000	120	0.02	0.9934	5.4302
5000	130	0.02	0.9950	1.9974
5000	115	0.03	0.9925	3.1773
5000	115	0.05	0.9935	3.612
8000	120	0.02	0.9947	7.4821
8000	115	0.01	0.9942	5.65
10000	115	0.01	0.9957	19.1571

At first, SVC showed a low accuracy so we tried Bagging and Adaboost, a pretty difficult attempt in SVC, to improve, however, those methods failed for different reasons. Then, we tried GridSearchCV with a large number of parameter combinations of C and gamma. However, SVC is not able to use GPU in colab so we waited for a long time for the outcomes. Overall, training time is proportional to training size and the number of C. Unlike parameter C, gamma could slightly influence the training time. Our best model of SVC is the one in bold above.

D. *Neural Network (from Scratch) Results*

We explored different hyperparameters, such as the size of the training set, the learning rate, the number of epochs, the weight initialization method, and the optimizer.
The results are presented in Table 4.

Table 4: Neural Network (from Scratch) Results

Training Size	Learning Rate	Epochs	Weight Initialization	Training Time	Success Rate
15000	400	15	Uniform(-5, 5)	6.089	0.9984
10000	100	15	Uniform(0, 1)	4.06	0.9962
10000	200	15	Uniform(-1, 1)	4.2010	0.9958
10000	400	15	Uniform(-5, 5)	3.9909	0.9956
10000	100	15	Uniform(0, 1)	3.9055	0.9955
5000	450	10	Uniform(-3, -3)	1.4167	0.9931

The success rate was highest when the network was trained with a learning rate of 400, 15 epochs, and a uniform weight initialization. The training time was also shortest in this case.

The results suggest that the network is able to learn the underlying relationships in the data quickly and efficiently. The network is also able to generalize well to unseen data.

Here are some additional observations from the table:

The success rate increased as the training size increased. This is to be expected, as the network was able to learn more from the larger dataset.

The success rate decreased as the number of epochs increased. This is because the network was more likely to overfit the training data with a larger number of epochs.

The training time of the Neural Network from scratch implementation was shorter than that of the PyTorch one. This may be because for the scratch implementation we explicitly calculated the derivatives beforehand and that the model is simpler.

VI. Conclusions

Since this was a group project, models were evaluated on four different laptops with four different specifications. In order to determine the best model overall, the final results were calculated on the same laptop using an i7 4.90 GHz CPU. Overall, the best performing model was the “Neural Network (from scratch)” with a sample size of 10,000, a learning rate of 400, iterations using 15 epochs, and with a uniform distribution for weight initialization. The final results were an accuracy of 0.9935 with a training time of 0.0156 seconds.

VII. Team evaluation

All team members contributed greatly to the overall project. We met multiple times, critiqued each other's mathematical models, and proposed various algorithms to optimize. Although everyone was an equal contributing member to each piece of the project, Xiaoyan focused more on the SVM algorithm, Sushant focused more effort on the Neural Network from Scratch algorithm, Justin focused on the Logistic Regression algorithm, and Chandramani focused on the Neural Network with PyTorch algorithm. All team members worked extremely hard, and no one participated any less than another.