HW2 Report

Kimia Tahayori 110124141

Kasra Mojallal 110124782

Siyam Sajnan Chowdhury 110124636

28 May, 2023

Question 1: Create some simulated data sets and then use the Adaline neuron and the Sigmoid to perform some prediction.

Our code is implementing the Adaline and Sigmoid neuron learning algorithms using both Batch Gradient Descent (BGD) and Stochastic Gradient Descent (SGD), and then testing and comparing their performances over a synthetically generated dataset. We use the Numpy library in the Python file to generate synthetic data. The code generates two sets of data: an input vector X(which draws random samples from a normal distribution) and an output Y(figure 1). Both vectors contain 5000 data points. Then we generate the error vector eps, which represents a normal distribution with less variance than that of the input X.

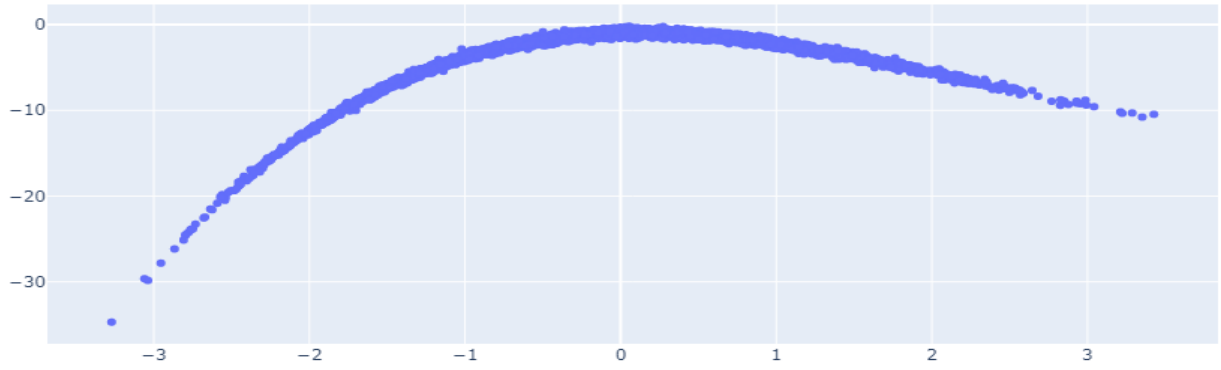

Figure 1 – The graph illustrates the relationship between Y and X

We have two classes: Adaline and Sigmoid Neuron. The Adaline class represents a single-layer NN. It uses a linear function for binary classification. The activation function that Adaline uses is a linear activation function which makes it more robust in dealing with noisy data. (Figure 2) The Sigmoid Neuron class represents a single neuron that uses a non-linear activation function(sigmoid), enabling the modeling of more complex and non-linear tasks rather than Adaline. These classes provide building blocks for constructing neural network models in PyTorch.

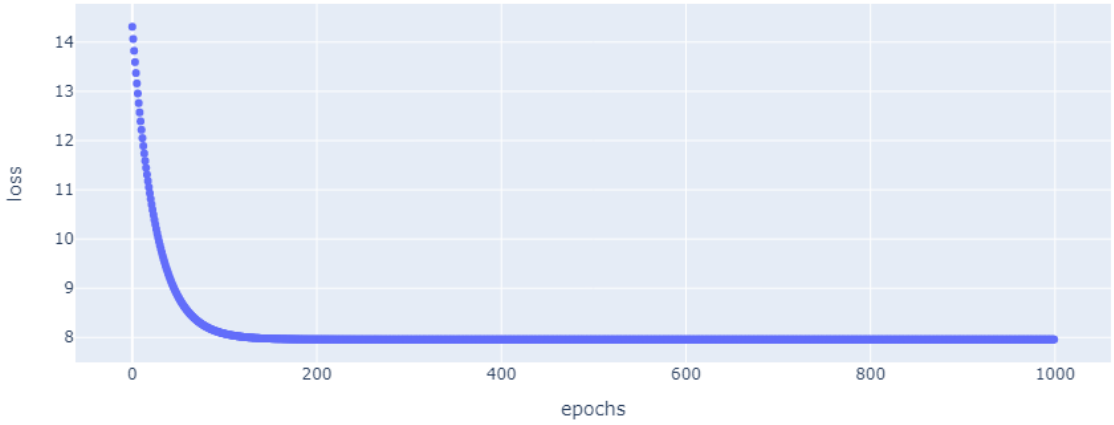

Figure 2 – the figure represents the loss of Adaline during 1000 epochs

Cross-validation Training of an Adaline Model:

The Python file performs a 10-fold cross-validation training of an Adaline model on the given dataset using PyTorch. In the Cross-Validation loop, iteration is going through the dataset using a 10-fold cross-validation procedure. The training and test datasets are reshaped and converted into PyTorch tensors. The training data is used to implement **Stochastic Gradient Descent (SGD)** by setting the batch size to 1 and enabling shuffling. An Adaline model is initialized for each fold. The model's parameters are optimized using the SGD algorithm with a learning rate of 0.01(figure 3). We repeat the same procedure for Gradient Descent (GD) instead of SGD, which will be a minor change in the data loading step. Instead of using a batch size of 1, as in SGD, we would load the entire dataset in each iteration. (figure 4)

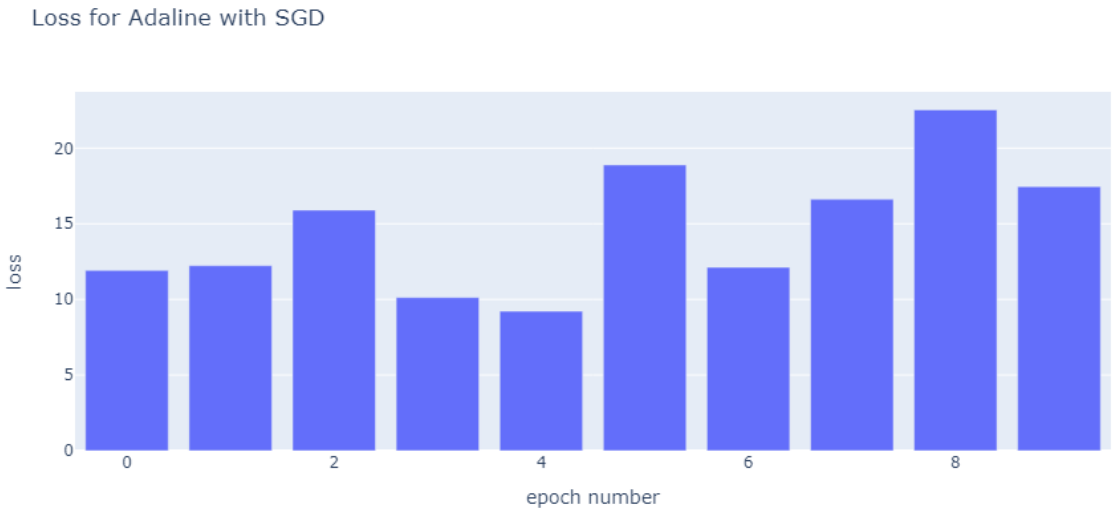

Figure 3 – the figure represents the loss of Adaline during epochs using SGD

With this change, the updates are made based on the aggregate error over all the training examples instead of updating the model's parameters based on each training example. This method can lead to more stable convergence but might be slower and more computationally expensive, especially for large datasets.

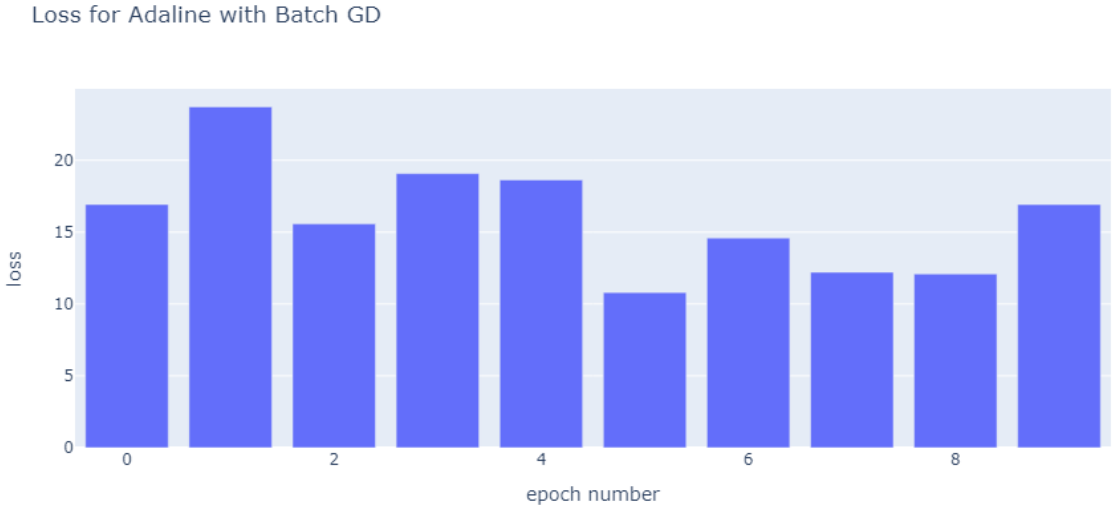

Figure 4 – the figure represents the loss of Adaline during epochs using BGD

Cross-validation Training of a Sigmoid Neuron learning algorithm:

In this part of the provided Python code, we perform a 10-fold cross-validation training of a Sigmoid Neuron model on the given dataset using PyTorch. The main process takes place inside a loop that iterates over each fold of the 10-fold cross-validation. We create training and test datasets for each fold by splitting the data. Then, we set up the training data for **Stochastic Gradient Descent (SGD)**, where the model is updated based on each individual training example. The Sigmoid Neuron model's parameters are set to be optimized using the SGD algorithm with a learning rate of 0.01.(figure 5)

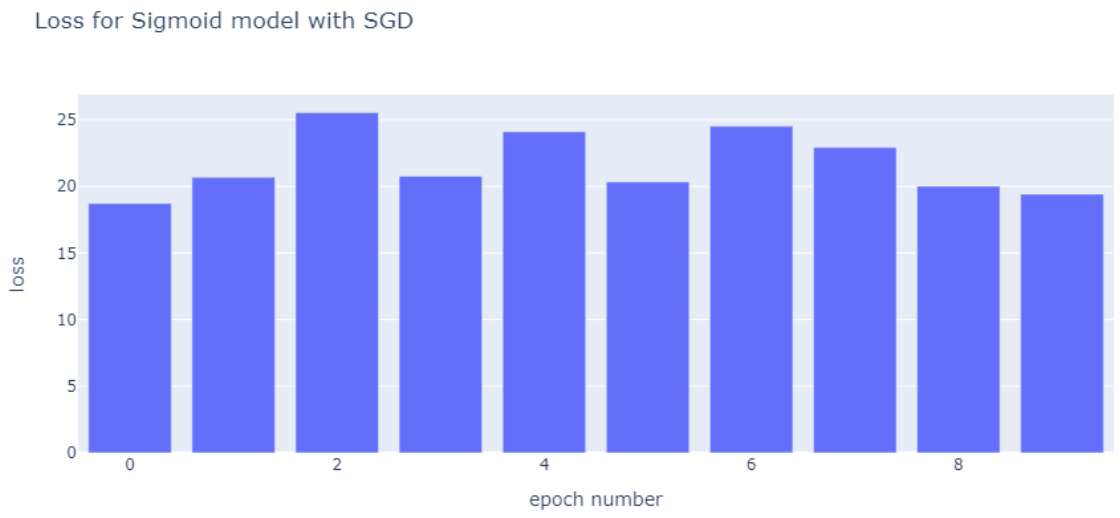

Figure 5 – the figure represents the loss of sigmoid function during epochs using SGD

In the modified scenario of the provided Python code, we perform a 10-fold cross-validation training of a Sigmoid Neuron model on the dataset using PyTorch, but instead of Stochastic Gradient Descent (SGD), we use **Batch Gradient Descent (BGD)**. Instead of processing each data point individually as we do in SGD, in BGD we adjust the batch size to the size of the training set, processing all the training examples for each iteration of gradient computation and update step. (figure 6)

After running this code, we will have four average test losses - one each for the Adaline and Sigmoid neuron models trained with SGD and BGD. Comparing these can help you determine which model and which learning algorithm performed better on your synthetic dataset.

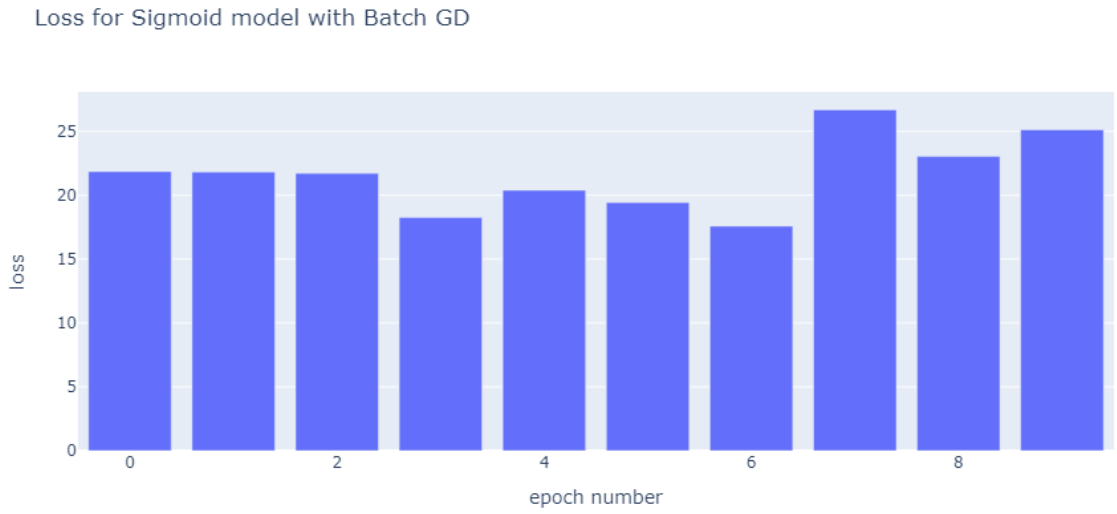

Figure 6 – the figure represents the loss of sigmoid function during epochs using BGD

Result:

Adaline:

In the Adaline model, SGD resulted in a lower test loss (14.7) than BGD (16.05). This indicates that SGD, which updates more frequently, may lead to better performance and prevent overfitting in the Adaline model.

SGD:

Conversely, for the Sigmoid neuron model, the average test loss was almost the same for SGD (21.69) and BGD (21.59), suggesting that both learning methods performed comparably. This could be due to the smoother, differentiable nature of the Sigmoid function, which may make it more compatible with both types of gradient descent.

These results, however, are specific to the dataset and parameters used. Different datasets, parameters, or additional optimization strategies could yield different results.

Question 2: In this question, you are to create some simulated data sets and then use the Perceptron neuron to perform some classification.

For the first part of question 2, we aimed to generate 5000 random data points, such that these points would be classified into two distinct categories (-1 and +1) based on their position relative to a chosen linear function. In this scenario, the function used for separation was the line y = x. Initially, 2500 data points were randomly created in the range of -100 to 100 for both the x and y coordinates. For classifying the generated data points, we compare the y coordinate of each data point with its x coordinate. If the point was identified as being below the line y = x and thus assigned to class -1. Conversely, if the y coordinate was greater than or equal to the x coordinate, the point was understood to lie on or above the line y = x, thus belonging to class +1.

Perceptron Learning Algorithm Implementation

For the second part of question 2, the aim of this task was to implement the Perceptron learning algorithm on a synthetically generated dataset and evaluate its performance using a cross-validation method. The dataset was split into 70% of data for training sets and 30% for testing sets. The Perceptron model was initialized with a tolerance of 1e-3, which controls the stopping criteria. The cross-validation process was then conducted on the training data, with the number of folds (cv) set to 5.

The cross-validation scores shed light on how well the Perceptron model performed during training across different subsets of the data. The mean cross-validation score was quite high at approximately 0.9994, indicating the model's consistency in performance across the folds. The final test set score of approximately 0.9987 further attests to the model's ability to generalize and make accurate predictions on unseen data.

In conclusion, the Perceptron learning algorithm was successfully implemented and evaluated on a synthetic dataset. The use of cross-validation provided a robust measure of model performance and helped avoid overfitting. The high test set score highlights the model's capability to generalize effectively and make precise predictions on new, unseen data.

how does the performance vary when changing the learning rate?

The learning rate is a hyperparameter that determines how much the model adjusts in response to the estimated error each time the model weights are updated. Choosing the right learning rate is crucial as:

- If the learning rate is too high, the model might overshoot the optimal point and the algorithm might not converge or even diverge, resulting in poor accuracy.

- If the learning rate is too low, the model will need more epochs (iterations over the entire dataset) to converge towards the best values, which can slow down the training process.

Therefore, the learning rate has a direct impact on both speed (training time) and accuracy. It is often beneficial to experiment with different learning rates to find a good balance.

how does the performance vary when changing the size of the training and test sets?

The size of the training set influences how well the model can learn. If the training set is too small, the model might not be able to learn the underlying patterns in the data, leading to poor performance when making predictions (underfitting). On the other hand, if the training set is large, the model can learn more effectively, resulting in better accuracy. However, larger training sets will also require more computational resources and time to train the model, affecting the speed.

For the third part of question 2, we replaced the perceptron algorithm with the pocket algorithm. The pocket algorithm includes an additional feature: it keeps the best solution obtained so far during its iterations in its "pocket". This makes it more reliable when dealing with datasets that are not linearly separable.

The data is split into training and testing sets, with a test set size of 20%. The pocket algorithm is initialized with a weight vector of zeros, which is updated each time a data point is misclassified. The algorithm iterates over the data points for a specified number of epochs (T = 1000). If a data point is misclassified, the weight vector is updated. The algorithm also maintains a record of the best weight vector encountered so far – the one that resulted in the fewest misclassifications (errors).

After fitting the model using the training data, the model's performance is evaluated on both the training and test datasets. The sign function is used to determine the

predicted class labels. These predicted class labels are then compared to the actual labels to compute the accuracy of the model. Then, we print gives the final training set score and test set score (accuracy), which are measures of how well the pocket algorithm was able to classify the data points in the training and test sets, respectively.