PROJECT ELECTIVE FINAL REPORT

Samaksh Dhingra (IMT2019075)

December 9, 2022

Contents

1 Introduction

The main task for the final leg of our project was to make a simulator for the parallelized Stochastic Gradient Descent Implementation (as described in TensAir paper) so that we can visually interpret how the values of the loss function with respect to each data batch is changing to understand things better. This would make base for providing a final concrete mathematical proof much stronger. For the ease of understanding the problem broadly, we will look at task of linear regression using gradient descent with the described parallel setting.

2 Phase 1: Choosing an appropriate data set

The dataset chosen for this task is an open source dataset provided on Kaggle. The choice for the dataset is made due to its easy to understand nature. Also the data has been preprocessed and cleaned priorly so that we can focus and spend our time on our main problem.

2.1 Dataset Description

The dataset contains information about the various attributes of an automobile vehicle like height, weight, length, make, etc. and its price. The task is to use the given data as training dataset for predicting price of an automobile given its all feature description. Dataset chosen is such that it is not too large that it takes time to process it and not too small that the the batches formed are of very small size

2.2 Data Preprocessing

The data is normalized using MinMax Scaler.

3 Phase 2: Linear Regression on whole data using Gradient Descent

For first understanding how the linear regression can be implemented in python, the mathematical model of linear regression using gradient descent is understood and implemented. Initially we will be running it for whole data set to get an overall understanding about convergence rate, etc.

As we know the formula of a linear regression model is

$$y_{pred} = bias + \theta X$$

where bias is the intercept and Θ is the slope/coefficient. A multi-regression model will be written as:

$$y_{pred} = bias + \theta_0 X_0 + \theta_1 X_1 + ...$$

The value of y are dependent on the θ values. In order to achieve optimum θ values, we use Gradient Descent (SGD). The aim of the it is to minimize the error function which can also be written as:

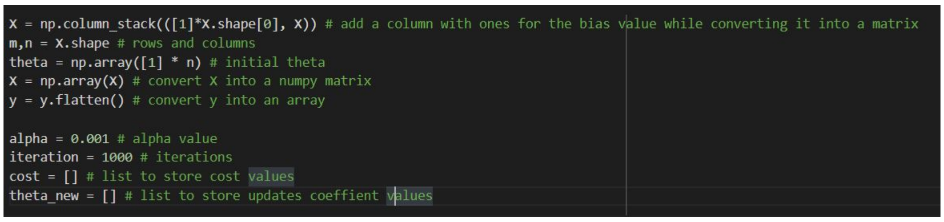
$$J = \frac{1}{2m} \cdot \Sigma(y_{pred} - y)^2$$

In order to minimize the function above, the θ coefficients can be calculated as the derivative of the the cost function. Using a small α value, calculate the coefficient (θ) values as:

$$\theta_{new} = \theta_{old} - \frac{\alpha}{m} \cdot \Sigma(y_{pred} - y) \cdot X$$

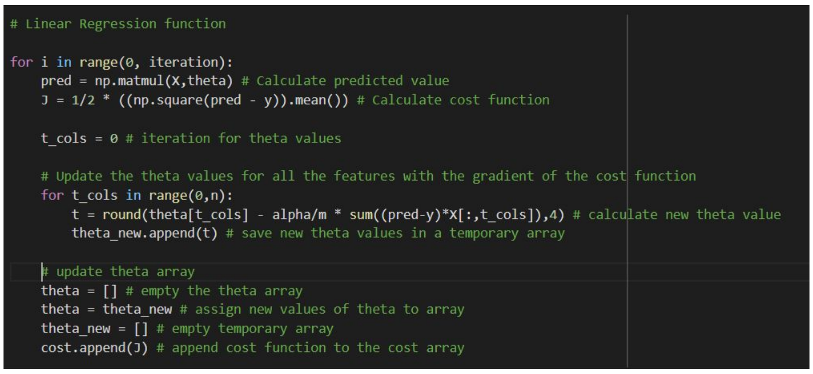
This can be repeated for a considerably large number of iterations until the local minima of the cost function is met. Various values of alpha can be tested before selecting the one that is large enough to converge at a local minima but small enough to reduce the cost function with every iteration.

First we initialize the values needed to produce the model. To accommodate the bias term, we add a column of ones to the dependent data set. This column is added to both the train and test set.



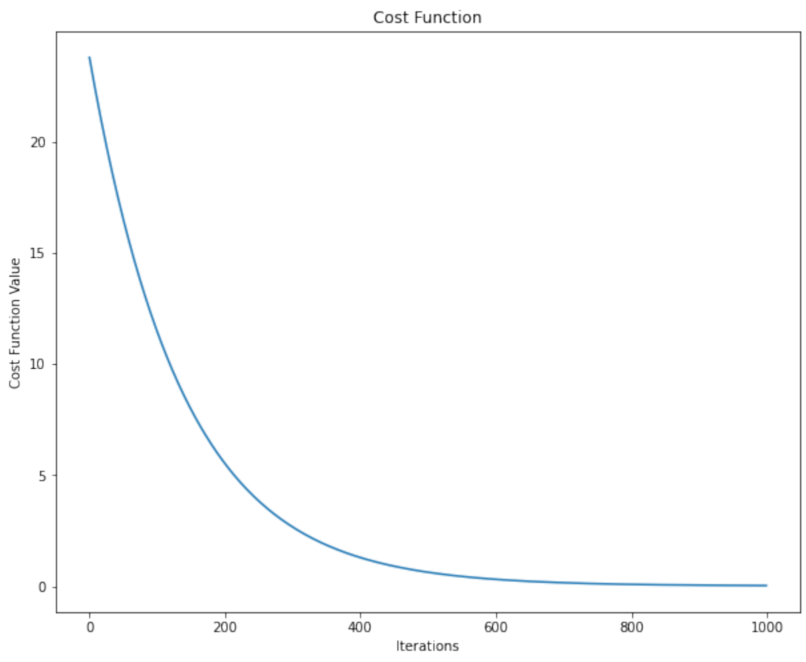
```
X = np.column_stack(([1]*X.shape[0], X)) # add a column with ones for the bias value while converting it into a matrix
m,n = X.shape # rows and columns
theta = np.array([1] * n) # initial theta
X = np.array(X) # convert X into a numpy matrix
y = y.flatten() # convert y into an array

alpha = 0.001 # alpha value
iteration = 1000 # iterations
cost = [] # list to store cost values
theta_new = [] # list to store updates coeffient values
```

Code for Linear Regression:



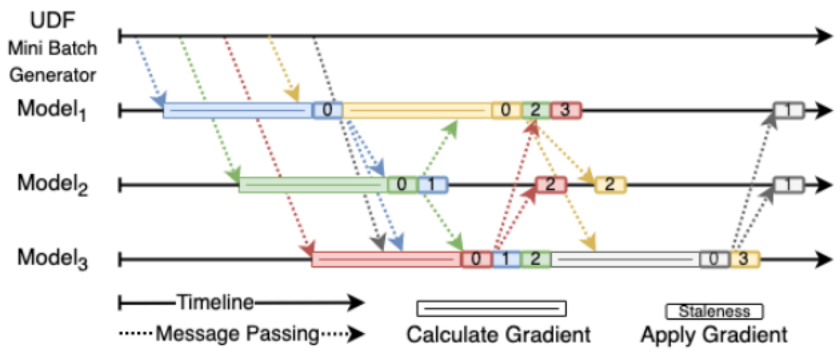
```
# Linear Regression function

for i in range(0, iteration):
    pred = np.matmul(X,theta) # Calculate predicted value
    J = 1/2 * ((np.square(pred - y)).mean()) # Calculate cost function

    t_cols = 0 # iteration for theta values

    # Update the theta values for all the features with the gradient of the cost function
    for t_cols in range(0,n):
        t = round(theta[t_cols] - alpha/m * sum((pred-y)*X[:,t_cols]),4) # calculate new theta value
        theta_new.append(t) # save new theta values in a temporary array

    # update theta array
    theta = [] # empty the theta array
    theta = theta_new # assign new values of theta to array
    theta_new = [] # empty temporary array
    cost.append(J) # append cost function to the cost array
```

Loss functions v.s. Iterations graph obtained after running Linear regression on the chosen dataset is then plotted to visualize how loss function values behave after each step.



4 Phase 3: The Simulator

During this phase, I implemented the simulator with the help of understanding made in previous phases. For illustration purposes, we will use the example from *tensAIR* paper and understand how the simulation is implemented for it.



The data is batched in 5 parts namely - *blue, green, red, yellow, white* similar to example given above. Each batch is normalized in the way we discussed above for whole dataset.

As we saw from the analysis from running linear regression with gradient descent on whole data at once, that the convergence took around 600 iterations. Hence, here we run each batch for 100 iterations only so that it does not completely gets trained. The purpose of choosing less iterations is that in a streaming data setup, we don't have enough time to train whole data for large iterations, hence we just partially train it and use the gradients obtained to train the other batch of data to reduce latency.

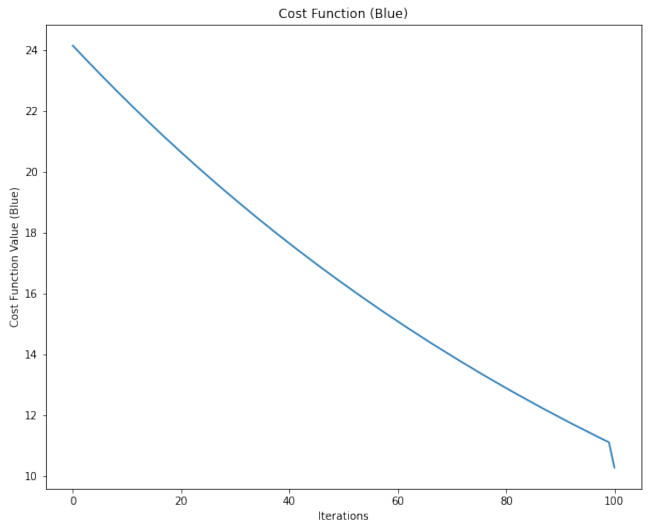
Simulation Procedure followed:

1. Using default initialization setting, i.e. θ_0 = [1,1,...] and *epochs* = 100, linear regression is performed on the *blue* data batch. This gives us final weights, gradients and a list of loss function values (at each iteration) for *blue* data batch.
2. Similarly, we perform linear regression on *green* data batch. We further apply *blue* gradients on the *green*'s weights.
3. Then, we perform linear regression on *red* data batch. We further apply *blue* and *green* gradients on the *red*'s weight.
4. Using *blue* weights as initial weights, we apply linear regression on *yellow* data batch. For *yellow* weights obtained, we further apply *green* and *red* gradients.
5. We use *red* weights as initial weights to run linear regression on *white* data batch. *white* weights obtained are then applied with the *yellow* weights.
6. Now, all the remaining gradients which came to models after their respective process ended, were applied to its final weights.
7. While we were performing all these operations, we kept track of loss functions with respect to each dataset on the go in their respective loss functions list.
8. We then used this list to finally visualize how the changes in loss function w.r.t to all the data batches took place after the whole process.

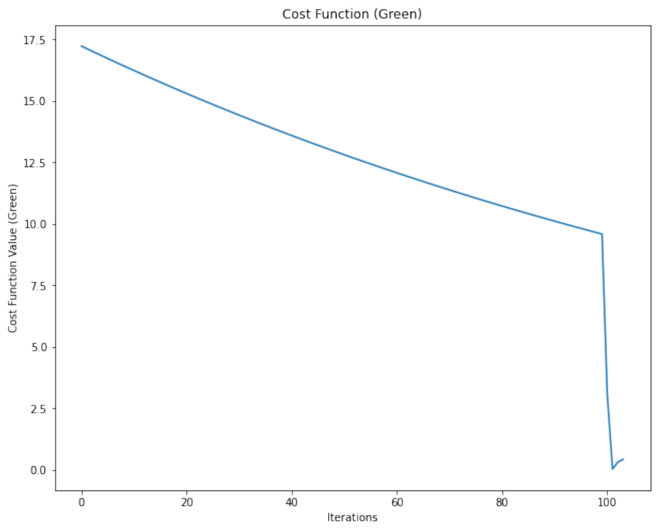
4.1 Results and Inferences

The loss function vs process iterations graph obtained for all the data batches are as follows:-

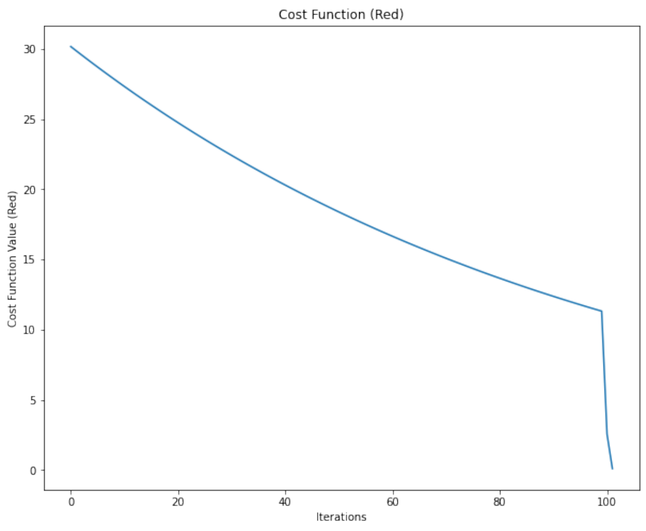
- Blue Data Batch



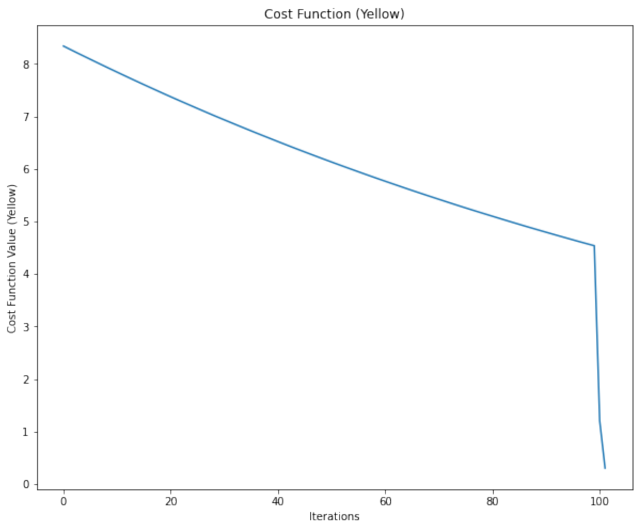
- Green Data Batch



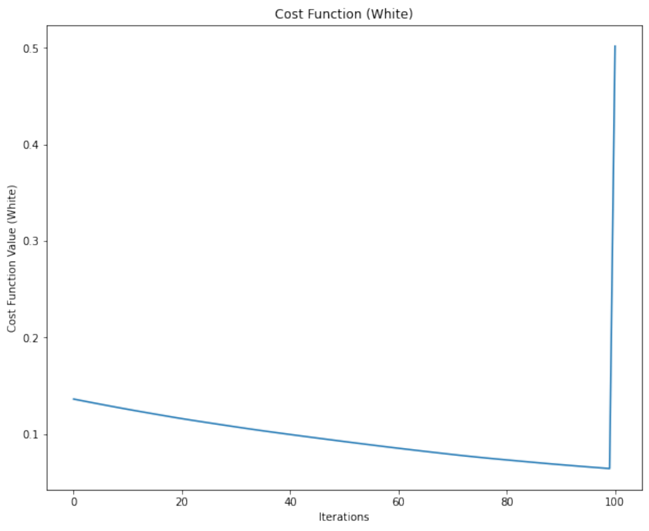
- Red Data Batch



- Yellow Data Batch



- White Data Batch



4.1.1 Inferences

We see that the results obtained are exactly how we expected our model to work. Some of the noted observations are:-

1. We see the overall trend of loss function getting reduced after each iteration in the process.
2. The sudden reduce of loss function after external gradient is applied can be seen, which is exactly what our aim was, i.e. reducing loss function as fast as we can.
3. We can see a sudden increase of loss function value after applying last *yellow* gradient to it. This helps us supporting the prediction we thought of that gradients which are very stale can lead to unexpected behavior when applied to already sufficiently trained model.

5 Potential Future Tasks

The current implementation of simulator is a basic one and hard coded for the chosen example as we wanted to get of how things are working while the process takes place. The execution of models can be made parallel in threads or any other methods of parallel processing. Also synchronization can be built for exchange and apply of gradients on the go itself. This setup however is majorly built up in tensAIR. The live visualization process for simulations can be integrated in the same for simulating any general setting of streaming data. Due to short of time, we saw the final visualizations after the process ends. However, with help of a web application we can see the visualizations live as the process runs. The web applications can also provide user to set their own data batches and setting for desired streaming of it.

6 Relevant Links

- Link to the Github Repo:- `https://github.com/samakshd/TensAIR_Simulator`