**Assignment 1**

**N Kapil Chandra**

**MM14B022**

**Note: The x-axis in all my plots show the number of cost values I have stores, not the number of epochs actually, however, I have explicitly mentioned the number of epochs an algorithm took in the description.**

**Note:** I am also submitting a .py file called sub.py which contains my most general model and few other functions I used. I am not writing it in the word file because the code has many lines and indentation is spoiling if I copy paste.

**Question 1:**

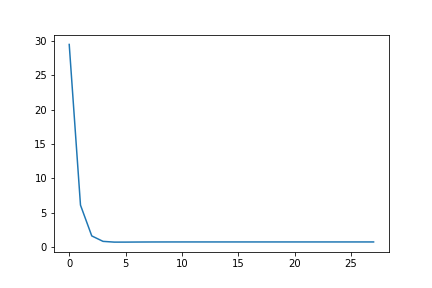
(i) Provided in sub.py

(ii) To see if my code works fine or not first I ran my code for the simplest case which is y=x line without any noise. To generate the data I have made a function of my own which takes the number of data points, slope and intercept of the line as input and returns the data. So I generated data for y = x line and see if my weight converges to 1 or not. After about 17 iterations the weight value was 0.999998727632, which is very close to the expected value 1. The test cases that can be used to check this can be any value of x. Using this weight we can calculate yhat, and finally can calculate error with respect to original y.

Another way I check if my algorithm is, by plotting the cost for each iteration. The curve should look something like this. This method of checking is especially useful when there is noise in data and you don’t know to what error value the model will converge to finally.

Using the function I made I generated data for a line with non-zero intercept as well. I did the same a above, ran the algorithm and checked the final values of weights (in this case, slope and intercept) if they are matching with the values I gave. And as expected the values converged pretty close to the original values.

To step it up, I even induced noise in datasets of both the cases by using random function in numpy library. To check the validity of the model in this case I plotted the cost function and saw the cost converged after a few number of iterations.

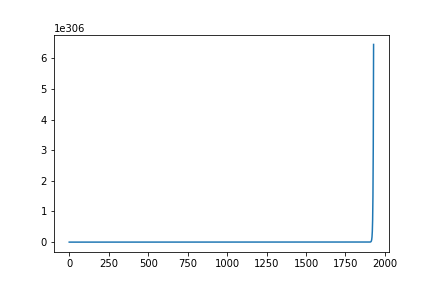


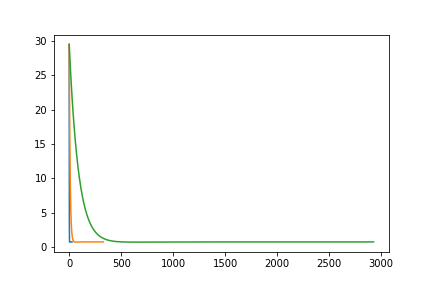
In the above graph, I took y=x data set with some noise(randomness) induced by my function and made the algorithm to converge, it converged after 28 iterations given the convergence criterion I set.

(iii) Coming to the convergence criterion. In the case where there is no noise in data I put a criterion of difference between previous cost and current cost less than 0.000000001. I also plotted the cost versus number of iterations graph to check if the model is converging.

Coming to the learning rate, I did experiment with it a lot. I have observed that there is a cutoff for learning greater than which the solution doesn’t converge. Using a learning rate greater than 0.3 in my case causes the solution to diverge and the plot of cost versus # of iterations looks something like this.

Here is when the learning rate is 0.4, the solution diverges to infinity





Green : learning rate = 0.001

Orange: Learning rate = 0.01

Blue : Learning rate = 0.1

If I use a learning rate of 0.1, the solution converges and it converges after 28 iterations, based on the convergence criterion mentioned above.

If I use a learning rate of 0.01, the solution obviously converges, but only after 329 iterations will the solution converge finally. For learning rate of 0.001 the solution will converge after 2930 iterations. So basically we need not use such a small learning rate because it takes more time to converge, a learning rate of 0.1 seems good enough for this problem.

It is better to experiment with different learning rates usually, 0.1,0.3,0.01…. And see which is giving good solution in less number of iterations.

**Question 2:**

For the whole of the assignment I have written a single function for training the data, it has different modes which can be specified before training the data. The type of gradient descent algorithm used can be specified while calling the function itself. Keyword to be given for a particular algorithm is listed below:

Normal gradient descent → ‘vanilla’

Stochastic gradient descent → ‘stochastic’

Stochastic with momentum → ‘stochastic’; ‘beta’ is the momentum parameter default value is set to 0, to put momentum mention the value to be used while calling the function

Using adagrad → ‘adagrad’

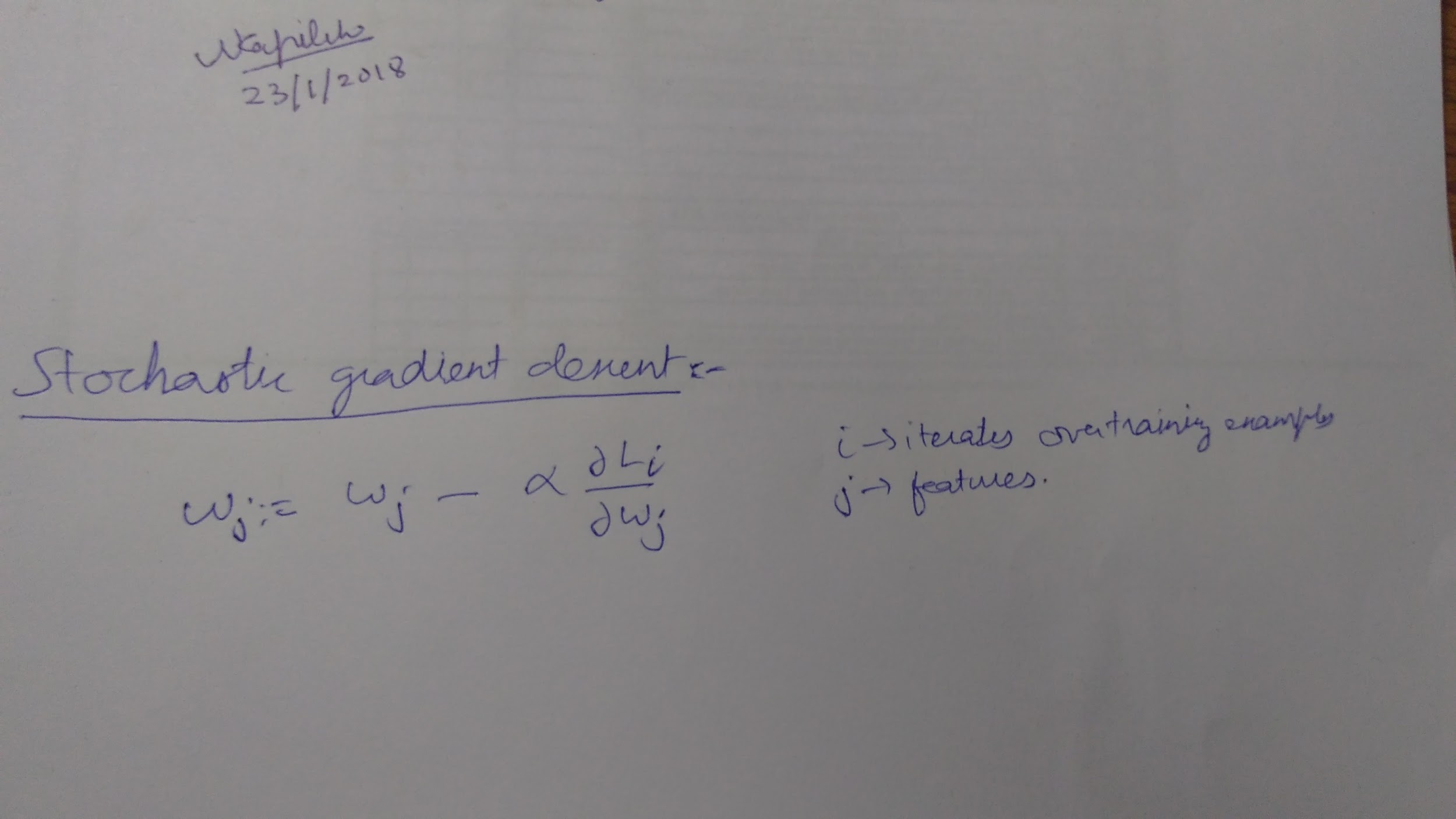
Using RMSProp → ‘rms’ parameter used for this is ‘gamma’ which is set to a default value of 0.9. For changing mention it while calling the function.

To test my code for the variants of stochastic gradient descent I have used a pre-defined function called ‘make\_regression’ from sklearn.datasets module. This is just to make the data set richer and the noise and number of features can be changed with great ease. In this exercise I have focused more on the algorithm, since I have already used the dataset generation in the previous question.

1. **Stochastic Gradient Descent:**

The difference is only in the part of the code where we are updating the parameters. While subtraction of the gradient, the gradient we calculate is one training example at a time in stochastic gradient descent, as opposed to batch gradient descent where we sum up all the gradients due to all training examples and then update parameters.

The update equation looks something like this:



The above equation should be evaluated for each training example at once.



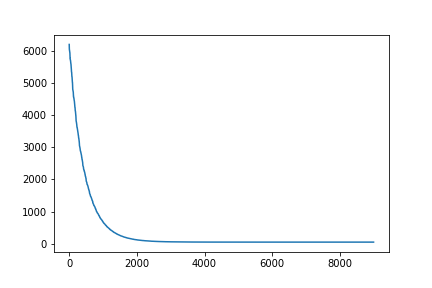
**Note: Please note that even though in the above code it looks like momentum is there, for running a plain stochastic gradient descent the default value of beta which is zero is being used.**

To check if my algorithm is working, first I have run the algo using normal batch gradient descent and saw the final loss in the dataset. Then I have run the stochastic gradient code on the same dataset and plotted the cost versus number of iterations, the solution converged very close to the value as in the case of batch gradient descent.

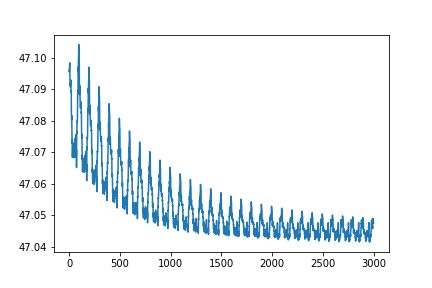
The plot of cost versus # of iterations looks like this:

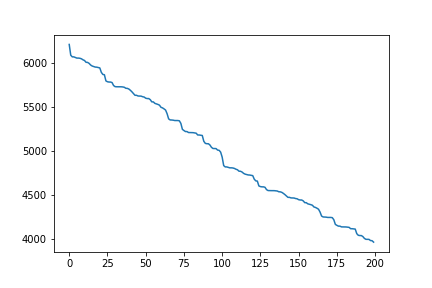
To check the validity, I ran batch gradient descent with a learning rate of 0.001 and it took 6900 epochs for the algorithm to converge. Whereas with the same learning rate, SGD converged in only 90 epochs. This is true because in stochastic gradient descent our model makes progress with each iteration whereas in batch gradient descent, our model has to look over all the examples before making progress.

To see the characteristics of SGB I plotted costs versus # of iterations, which looked like this:



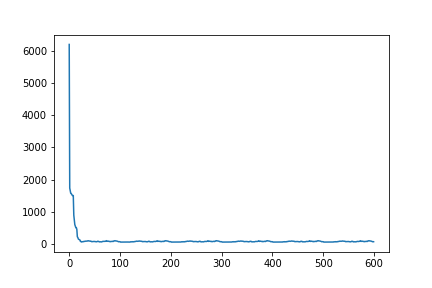
If we look closely at the end of the graph we can see that the solutions keeps oscillating about a point rather than smoothly converging to some value, which is a characteristic of SGD. Below are different regions in the above plot.



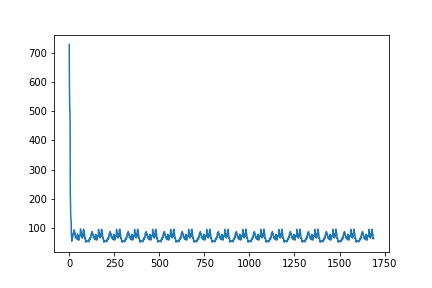


**Note: DONOT look at the number on x-axis for the above figures, they are just a small part of the original plot, I zoomed in to show zig zag nature of the plot.**

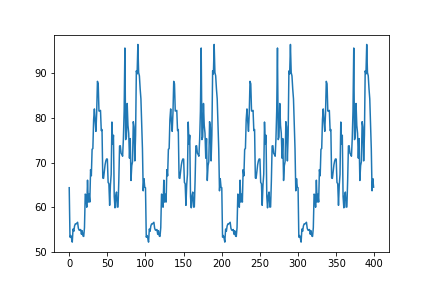
Another observation is that if I use a relatively high learning rate like 0.1 in SGD the solution will not come to a very good solution but will keep oscillating with approximately the same loss.



If we look closely we can see the oscillations,

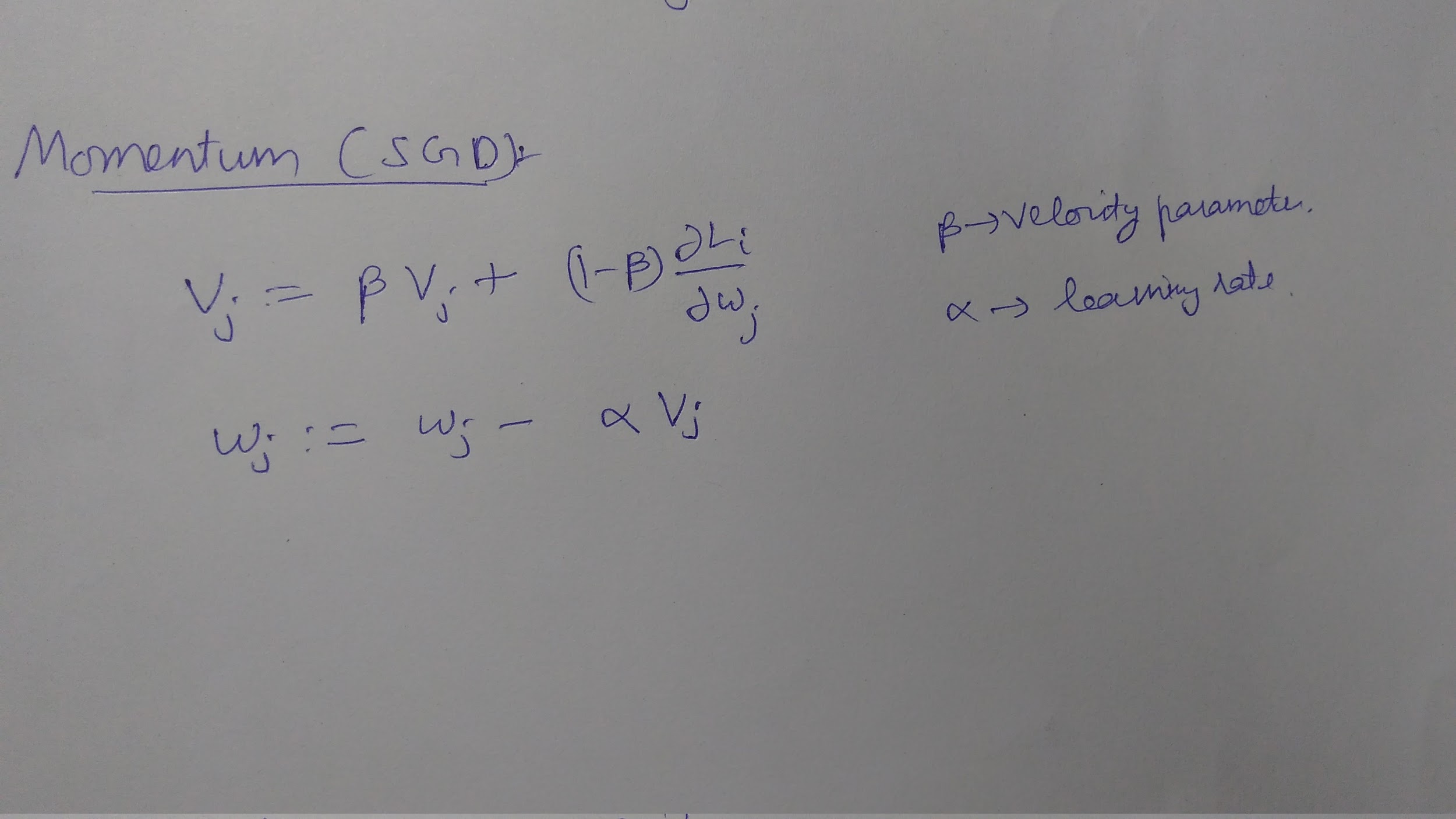


Looking at it closer we get this way:



Merits of this variant are that we can get solution faster than normal batch gradient descent. Demerit is that the solutions keeps oscillation around the optimum rather converging to optimum because in this type at each iteration local optimization takes place while updating parameters, because we consider only one training example at a time.

**(b) With momentum:**



**Note: The code for this part is same as in the case of stochastic gradient descent, only the value of beta should be mentioned while calling the function.**

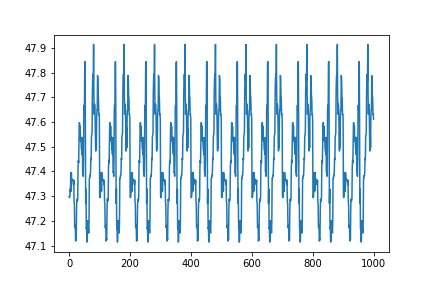
Code is similar to stochastic gradient descent, only that we have to explicitly give the value of parameter ‘beta’ to use while calling the function. In the case of simple SGD we shouldn’t mention the value of ‘beta’.

Taking the case where learning rate was 0.001 in SGD and I ran the algorithm for 80 epochs, without momentum the solution converged to a cost of 47.0510383347

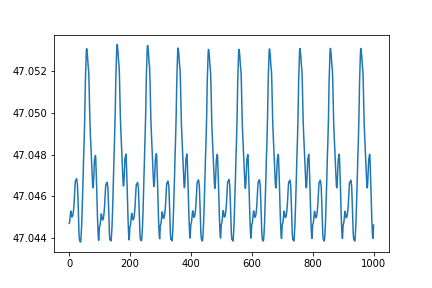
And with beta = 0.9 the model got a loss of 47.0471548101 and when beta = 0.99 the model got a loss of 47.0471548101 which is even lesser. This implies that using momentum in SGD helps in reaching the solution faster.

Similar results are obtained when a learning rate of 0.01 is used, the solution comes closer in a particular number of iterations when momentum is used.

Without momentum the oscillations look like this:



With momentum the oscillations look like this:



We can see that the magnitude of oscillations is smaller in the case where we have used momentum.

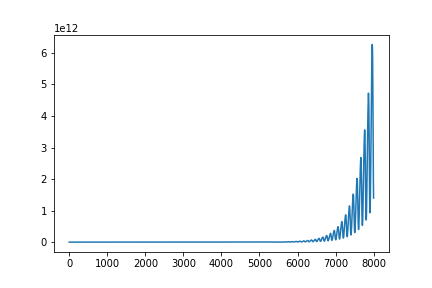
**Note: Don’t look at the number on the x-axis, these plots are zoomed in the plots of the original plots, so the numbers don’t actually correspond to the number of oscillations.**

If we see the magnitude of oscillation in the solution, the oscillations are less when momentum is used as compared when it is not used.

So the advantage of using momentum is that the oscillations in the solutions are reduced and it is more stable and also arrives close to the solution faster.

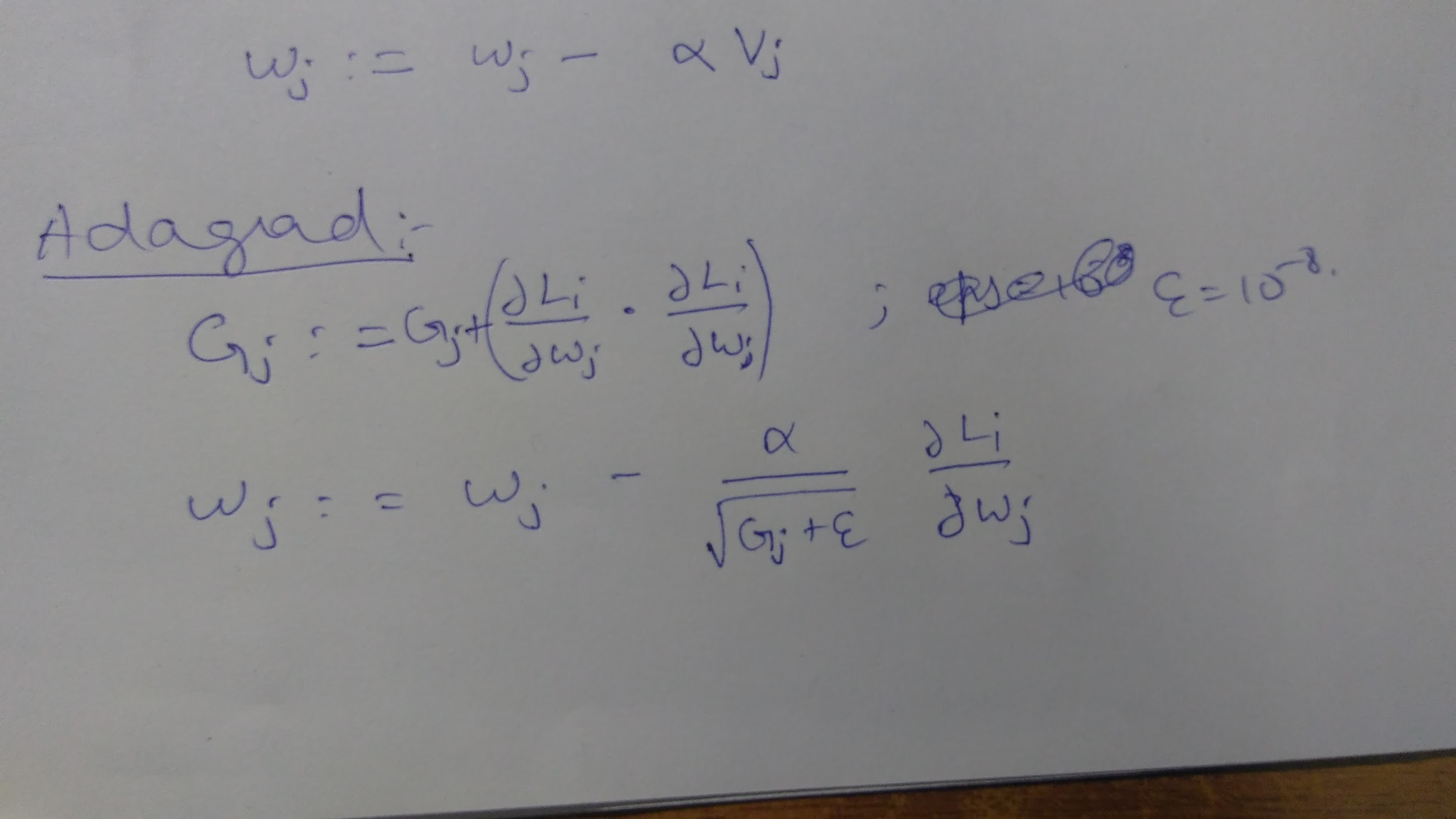
Sometimes it so happens that if we use more momentum the solution instead of going to minima, because of the momentum acquired will just jump over it and start diverging. This happened in the case when learning rate is 0.1 and beta = 0.99. This can be a potential disadvantage with momentum.

The following plot shows failing of the momentum:

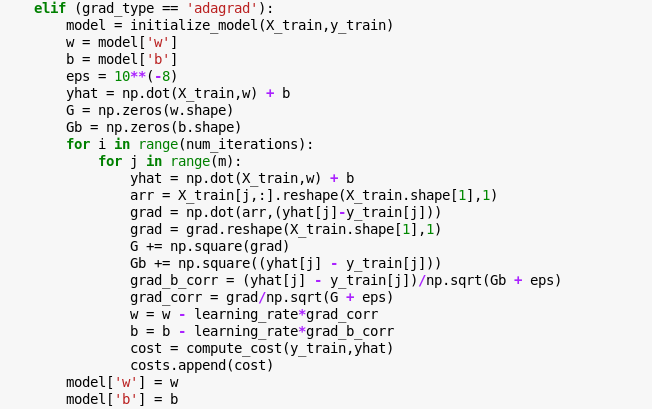


The cost kept on increasing and didn’t converge.

**(C) ADAGRAD:**

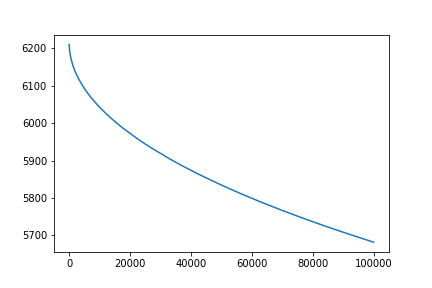


The code written for adagrad is given below:



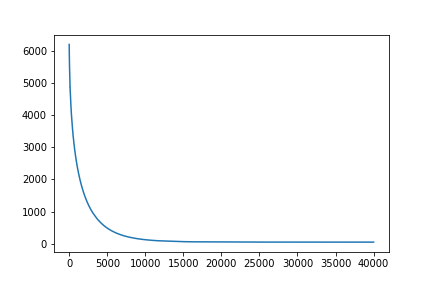
Adagrad mean where the learning rate dynamically changes after each iteration. It is adaptive gradient. The learning rate keeps decreasing constantly as the training occurs. To check the model first I used a learning rate of 0.01 and the ran for 1000 epochs, the result was very bad, the loss seemed very high. So i increased the number of epochs to 2000, the loss changed from 5681 to 5470 even after increasing the epochs by 1000. Looking at this I understood that the rate at which the model is progressing is very slow after some number of iterations, so this time I tried increasing the learning rate. I increased to 0.1, even though the loss was less this time it was not satisfactory, so I tried increasing learning rate more.

When learning rate = 0.01 and the algorithm progression as slow as a snail:



So i increased it to 1, the results were amazing. The algorithm worked like a charm. The algorithm which was having huge loss even after running for 2000 iterations when learning rate was 0.01 now has a loss of only 47.04 after 500 iterations when learning rate is 1. 47.04 is the loss we get when we run batch gradient descent and is infact the correct value. So from this I learnt that we have to start with a high learning rate in the case of adagrad.

When learning rate = 1 and the algo works well:



I even tried with a learning rate of 2, this also gave very good results.

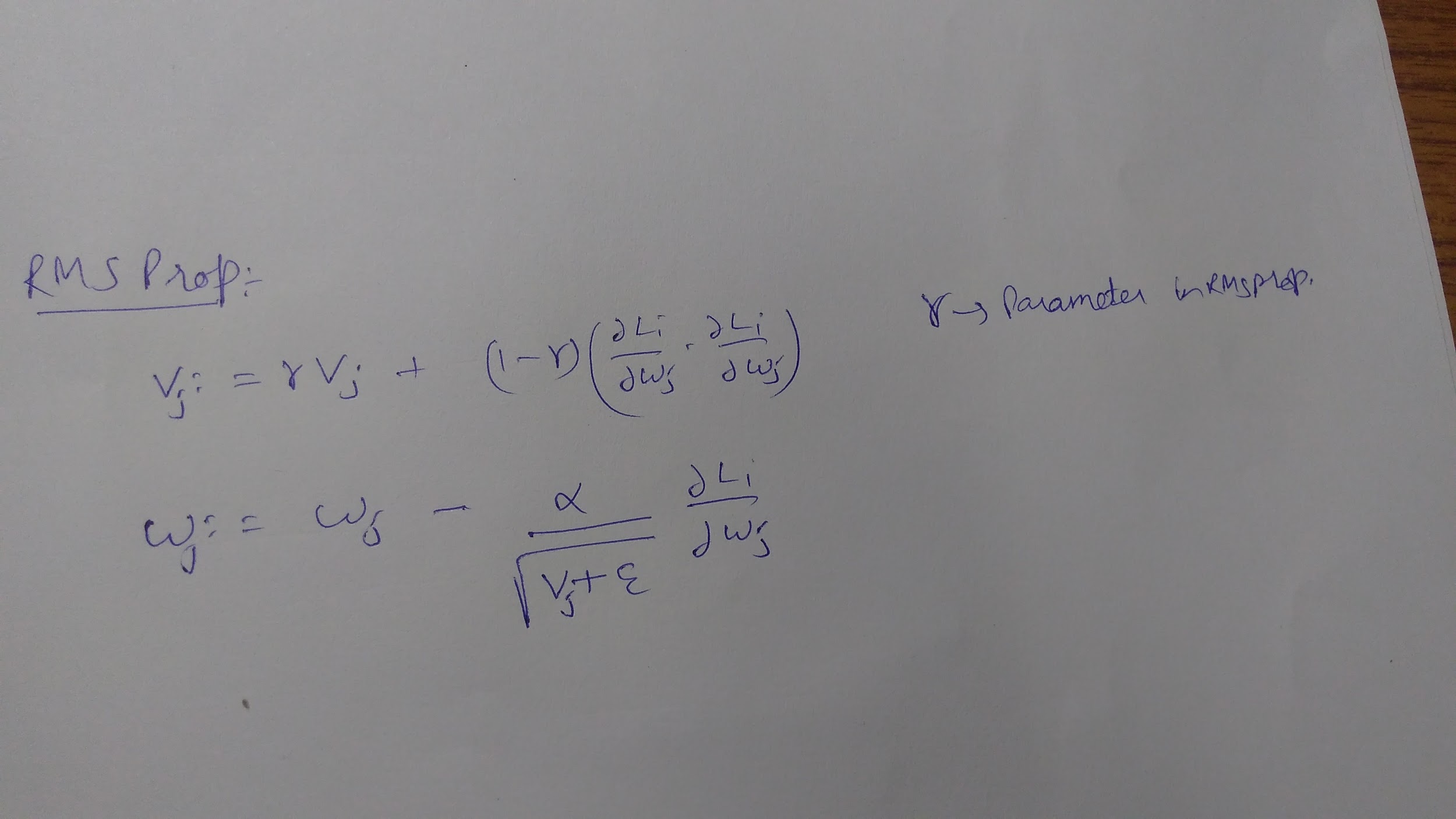
I tried learning rate of 6, after 400 iterations it came to 47.07 and after 1500 iterations it gave a loss of 47.0510199014. It is taking a lot of time to reach the correct value because of very high learning rate, but it will surely come to correct value after many iterations. So too high a learning rate is also not good in this case.

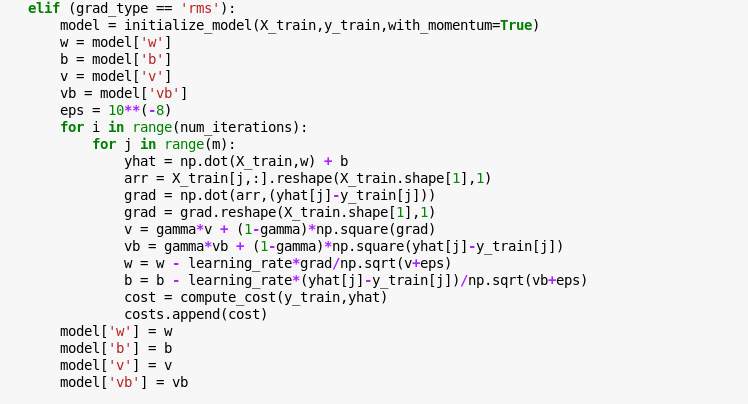
Learning rate of 1 seems to be a very good value.

Advantages of adagrad are that it works for high learning rates without diverging.

Disadvantages can be that sometimes the solution might take forever to converge because after a point the grads can become so small that they don’t make any progress at all.

**(d) RMSProp**

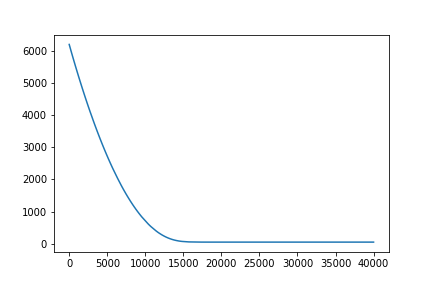




The default value of gamma value I used was gamma = 0.9.

I have used a learning rate of 0.01 and gamma = 0.9 and ran for 400 epochs and the solution converged to a loss of 47.127758801 which is slightly away from the value obtained from the value obtained from normal batch gradient descent.

The convergence plot for rms looks like this:



In the case of RMSProp a low learning rate can be used as opposed to the case of ADAGRAD. RMSProp is just another way to reduce the wastage of time because of oscillations in stochastic gradient descent.

RMSProp is taking more number of iterations to converge, this might be a disadvantage of this method.

**Question 3:**

**(a)**

(i) I deleted all the data points which have incomplete data.

Initially I tried running plain stochastic gradient descent, but i had to use very low learning rates. As low as 10^(-7) for it to not diverge. So instead I thought to use adagrad and used a learning rate of 0.01. To split the data set into training and test I used a function called train\_test\_split() in sklearn.model\_selection library which does a very good job of randomly picking part of the dataset.

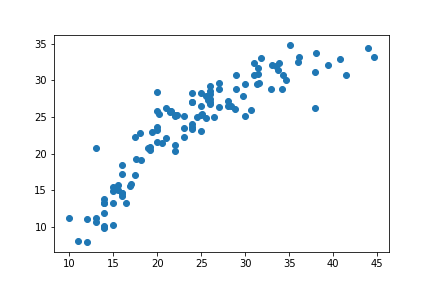
Model used : Stochastic with ADAGRAD

Learning rate : 0.01

# of iterations : 1000

The error which showed on the 30% of testing dataset is **6.37231013612**. Where as the training error was 5.91631581796. Both the errors are not so far apart which means that our model is performing well on the testing data set.

(ii)



We can see that the data points are close to or at least resemble y=x line, which means that the algorithms worked well.

**(b)**

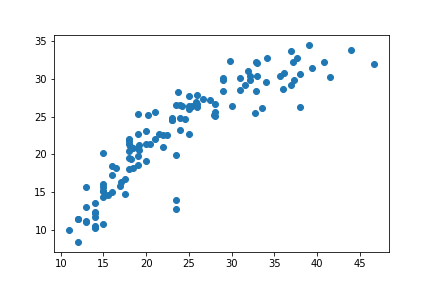
(i) The best way I can think of is first make a model without those data points. Somehow find a way to predict the values which are missing, and finally run the model again on this data set. This ensures that we have filled better estimates of missing values. But this is very tedious to code, so I won’t be able to implement this. However, I have discussed about other easy methods below.

There are some more ways in which we can replace NaN values differ depending on the variable we are dealing with. One method is where we can replace Nan values with the average of that feature taking the remaining existing data points. This works fine when we are dealing with features like height of a person, weight of a person, size of the house etc. Basically in the case of continuous variables we can do this. But this cannot be done in the case of categorical variables like number of bedrooms in house, gender of a person etc. Because 2.5 bedrooms in a house is absurd. So I think to replace Nan values in this case we can see which is the most occuring category and replace Nan with that category. So basically the mode of the data can be used in this case.

I have implemented the mean and mode replacements for continuous and categorical variables respectively. Here are the results I got:

I got a test error of **7.95518800357.** And the training error I got was 6.16346390743.

(ii)



We can see that the data points are close to or at least resemble y=x line, which means that the algorithms worked well.