M2 IASD

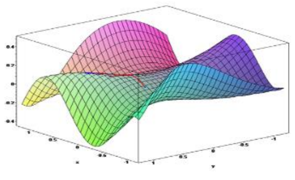
Fast SGD with Adagrad and Momentum, comparisons with basic implementations
With API apache spark

Realized by:

- Aziz Ben Ammar
- Mohamed Amine Gaidi
- Amenallah Salem

Professor:

- Mr Dario Colazzo



2019-20

Summary

1 Introduction :

There are many APIs in Spark written in Python and R such as PYspark and SparkR respectively that allow us to use them from Python or R. However, most Spark books and online examples are written in Scala. Arguably, we think that learning how to work with Spark using the same language on which the Spark code has been written will give us many advantages over Java, Python, or R as a data scientist:

Better performance and removes the data processing overhead

- Provides access to the latest and greatest features of Spark
- Helps to understand the Spark philosophy in a transparent way
- Better performance and removes the data processing overhead

The objective of this project is to compare Fast SGD Adagrad and Momentum with the other typical methods of gradient descent.

In the following we will compare optimization methods based on the SGD with those that are basics (Batch, SGD, MiniBatch), we will implement a code in Scala using Spark, this using three types of data structuring:

- **RDD**
- **Dataframe**
- **Dataset**

2 Challenge :

these methods used to optimise gradient descent, which tends to encounter multiple cha llenges such as problems convergence, which are mainly due to the poor choice of the a ppropriate learning step, sometimes there are problems where the latter is unable to adapt to the characteristics of a dataset, and using the same step risk of not getting us a good Convergence.

Any point in three-dimensional space can find the plane tangent to it, and hyperplane can also be found tangent to it in high-dimensional cases. So there are many directions at any point on the tangent plane, but only one direction can make the value of the function rise fastest. This direction is called gradient direction, and the reverse direction of the gradient direction is the direction of the function value decline fastest, which is the process of gradient descent.

Based on the above concepts, we further understand **Batch Gradient Updating BGD**A s the name implies, it calculates all samples at the same time to get the gradient value, and then updates the parameters. This method is very simple. It can converge to global optimum for convex function and local optimum for non-convex function. At the same time, its shortcomings are obvious: in the large amount of data, it takes a huge amount of memory, takes a long time to calculate, and can not be updated online. Face the bottleneck of BGD**SGD**As the times require, it updates only one sample at a time. Compared with BGD, it converges faster and can update online, and has the opportunity to jump out of local optimum. However, SGD can not use matrix operation to accelerate the calculation process. Considering the advantages and disadvantages of the two methods mentioned above, there is a small batch gradient descent algorithm **(MBGD),** which only selects fixed small batch data for gradient updating each time.
Gradient-based updating also implies some challenges:

- It is difficult to choose the appropriate initial learning rate, which will hinder the convergence and cause the loss function to oscillate or even deviate from the minimum value.
- There are many local optimal solutions or saddle points in the process of non-convex loss function optimization.
- The parameters are updated with the same learning rate.
- In response to the above challenges, we will list some optimization algorithms

for you next.

If we take the gradient descent method as a process from the hillside to the valley, then the ball rolling has a certain initial speed. In the process of falling, the kinetic energy a ccumulated by the ball increases, and the speed of the ball will increase, and it will run to the valley bottom faster, which inspires us.**Momentum Method**

The momentum method adds a product of the gradient value and the decay rate on the basis of the current gradient value, so that the last gradient value can be accumulated Continuously. The attenuation rate is generally less than or equal to 0.9. When the **SGD** algorithm with momentum term updates the model parameters, it will increase the updating intensity for the parameters with the same current gradient direction as the previous gradient direction, while for the parameters with different current gradient direction and the last gradient direction, it will reduce, that is, the updating speed in the current gradient direction is slowed down. Therefore, compared with **SGD**, momentum method can converge faster and reduce oscillation.

AdagradThe idea is to automatically adjust the learning rate in the process of learning. For the parameters with low frequency, a higher learning rate is used, while the parameters with high frequency use a smaller learning rate.

3 Problem:

The objective was to calculate the different types of gradient descents on a regression problem to find the optimal parameters (W weight) to achieve the minimum (local AN Dglobal) of the   error function

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

The loss feature we used is:

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\underbrace{f_{\mathbf{w}}(x) - y}_{\text{residual}})^2$$

To update the W settings, simply choose a learning step (n) and multiply it by gradient and then subtract this value from the W value of a previous iteration.

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \mathsf{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

Loss Train (W): is the sum of errors of all the learning set

4 Implementation :

To implement the steps mentioned above we will first build a set of learning data,

which will be used to train our gradient function.

To do this we have adopted this method:

Each x value will be used to create each line of our data set that will be of the form:

(v1,v2),w)

With:

v1:v / v2:v+5 / w:5v+2 (v between 1 and 1000)

5 Data allocation:

Spark has a major asset, it allows us to do a parallel calculation, we will exploit this

advantage to calculate the different types of gradient descent using the notion of

distribution.

The creation of our data set and its distribution differs from one data structuring to ano ther, i.e.the typing of variables is not the same in RDD, Dataframe and Dataset, as well as its distributions:

RDD

Import our package :

```
import scala.collection.mutable.ArrayBuffer
import scala.math.sqrt
import breeze.linalg.{norm, DenseVector => BDV}
```

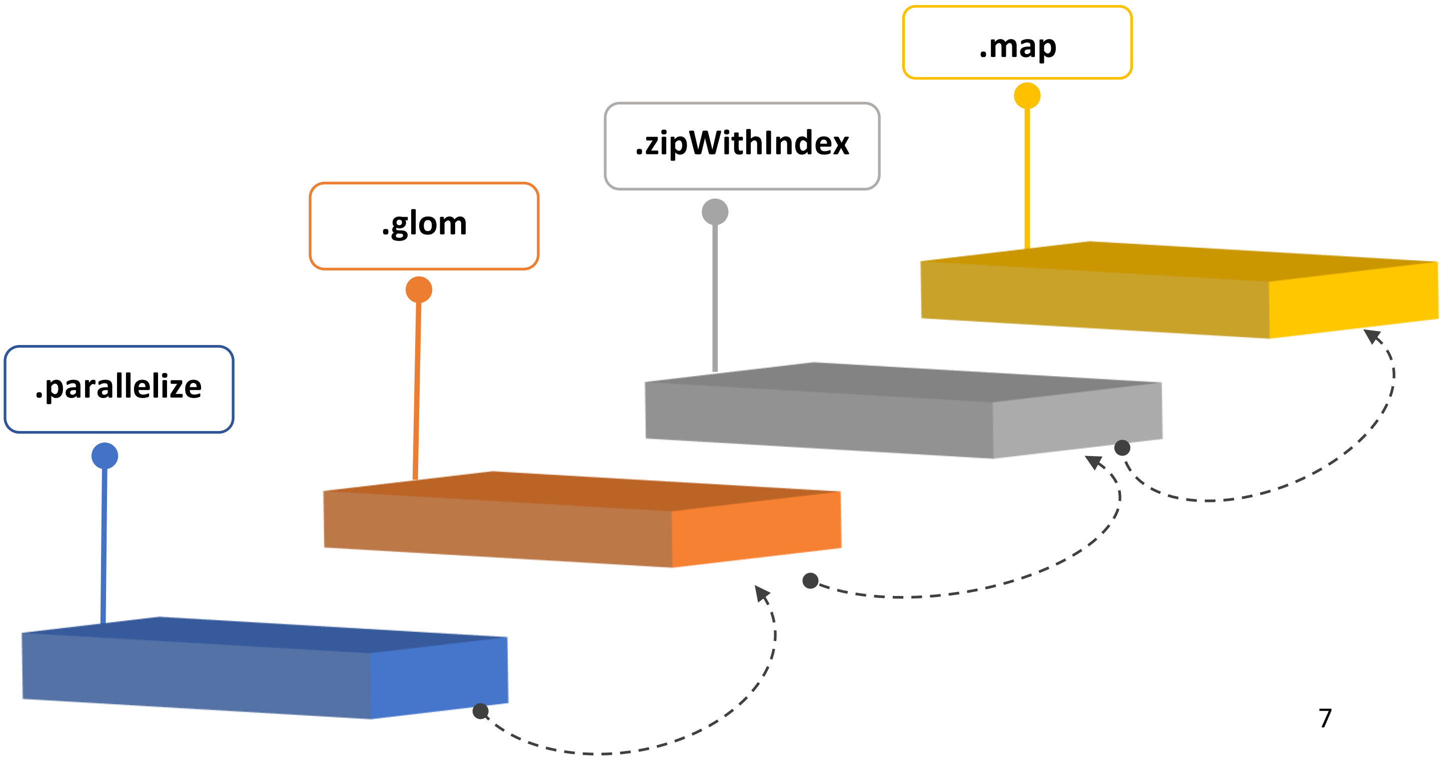
we use **.map** to create each line of this shape :

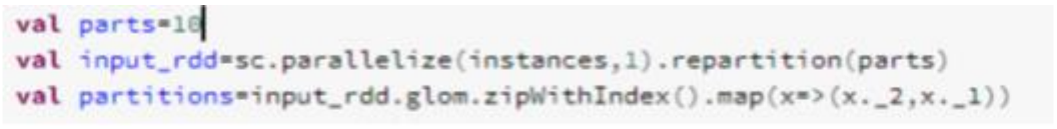
```
Array[(scala.collection.mutable.ArrayBuffer[Double], Double)]
```

```
val dataset=(1 to 1000).toArray
val brace=dataset.map(x=>(ArrayBuffer(x.asInstanceOf[Double],(x+1).asInstanceOf[Double]),(5*x+2).asInstanceOf[Double]))
```

Our function:

.parallelize	Divide our data into 10 partitions
.glom	Returns a created RDD by merging all the elements of each score into a list
.zipWithIndex	Assigns an index to each list representing a score
.map	Reversed the positions of the list and its index

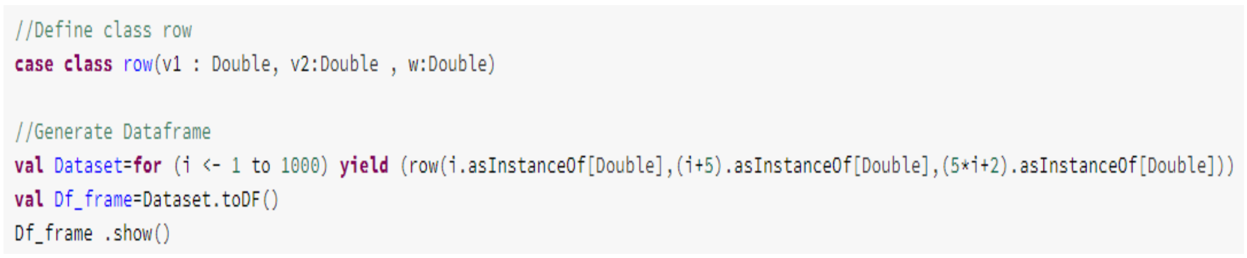




```
val parts=10
val input_rdd=sc.parallelize(instances,1).repartition(parts)
val partitions=input_rdd.glom.zipWithIndex().map(x=>(x._2,x._1))
```

Dataframe

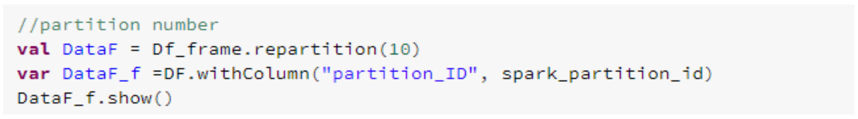
- First of all to define a line in Dataframe we should define a class containing v1, v2 and w in the form of a tuple



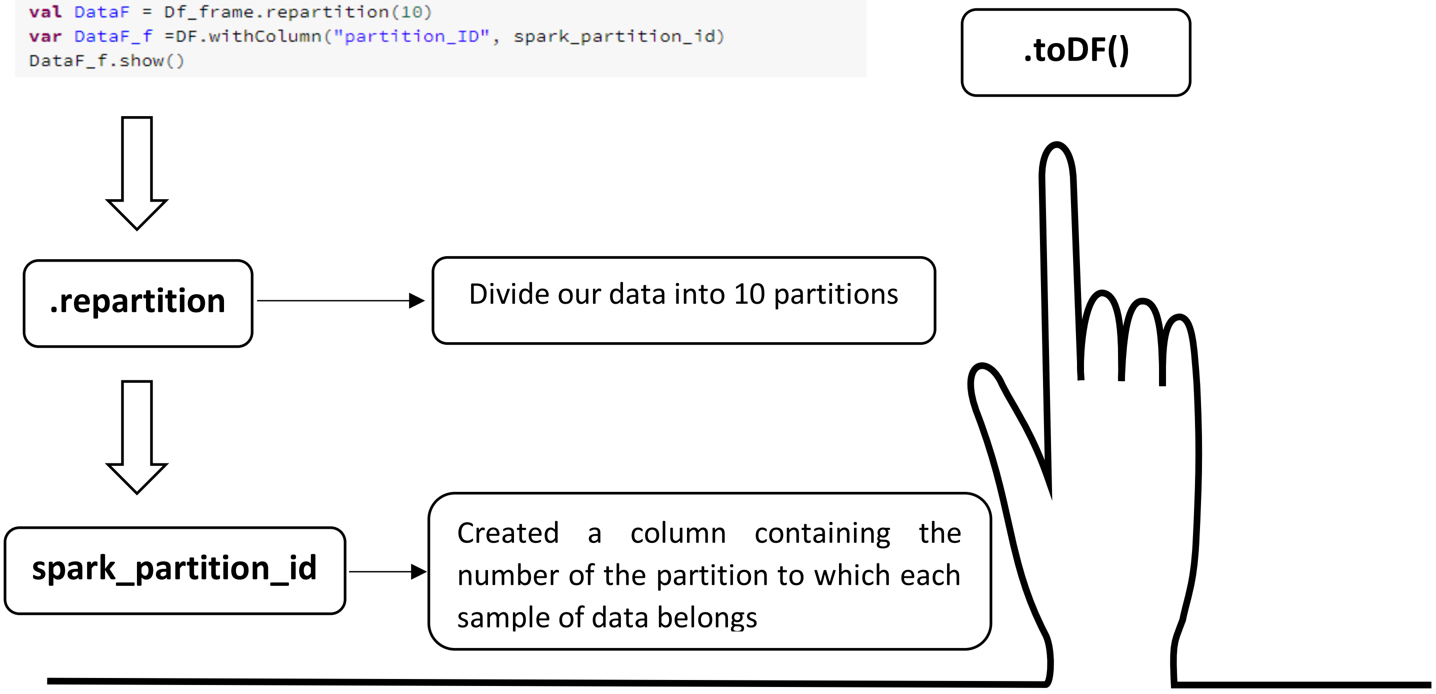
```
//Define class row
case class row(v1 : Double, v2:Double , w:Double)

//Generate Dataframe
val Dataset=for (i <- 1 to 1000) yield (row(i.asInstanceOf[Double],(i+5).asInstanceOf[Double],(5*i+2).asInstanceOf[Double]))
val Df_frame=Dataset.toDF()
Df_frame .show()
```

⇨ Then we generated 1000 examples in a vector

Later turned into Dataframe using the **.toDF()** function



```
//partition number
val DataF = Df_frame.repartition(10)
var DataF_f =DF.withColumn("partition_ID", spark_partition_id)
DataF_f.show()
```

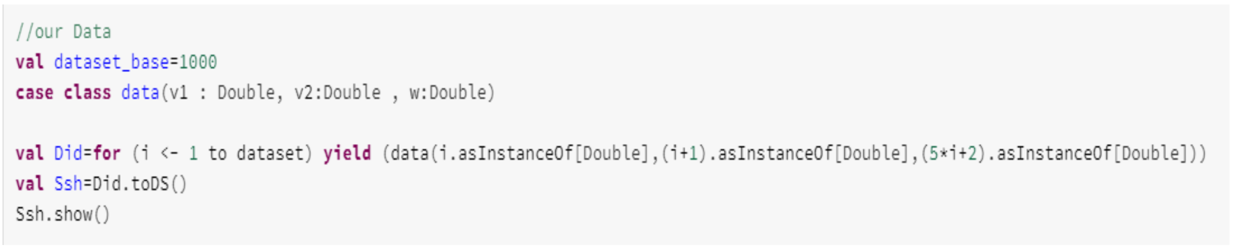


.toDF()

.repartition → Divide our data into 10 partitions

spark_partition_id → Created a column containing the number of the partition to which each sample of data belongs

Dataset:



```
//our Data
val dataset_base=1000
case class data(v1 : Double, v2:Double , w:Double)

val Did=for (i <- 1 to dataset) yield (data(i.asInstanceOf[Double],(i+1).asInstanceOf[Double],(5*i+2).asInstanceOf[Double]))
val Ssh=Did.toDS()
Ssh.show()
```

To create a dataset we follow the same steps as Dataframe except that we have replaced the **.toDF()** function with another **.toDS().**

6 Calculating gradients:

To be able to calculate gradient descent in several ways (Batch, MiniBatch ,SGD), we had to create new intermediate functions that are:

➢ **block_by_scal**(v:ArrayBuffer[Double],n:Double) : to multiply a vector by a scalar

➢ **block_scal** (v:ArrayBuffer[Double], u:ArrayBuffer[Double]) : To calculate the scalar product of two vectors

➢ **sommation**(v:ArrayBuffer[Double],w:ArrayBuffer[Double]) : to sum up the vectors

➢ **diff**(v:ArrayBuffer[Double],w:ArrayBuffer[Double]) : Subtraction of two vectors

➢ **grad**(v:(ArrayBuffer[Double],Double),u:ArrayBuffer[Double]) : this function will be used to calculate the gradient of a single example (Instance)

Batch Method:

This method consists of going through all the data to calculate the gradient of each instance before making a single update of the settings.

```
def compute_grad( p:Array[(ArrayBuffer[Double] , Double)],W:ArrayBuffer[Double] )={
  var grad_p = ArrayBuffer.fill(W.length)(0.0)
  for (i <- p){
    grad_p= somme(grad_p , grad(i,W) )
  }
  grad_p
}
```

Above we defined a function that calculates the gradient of a single partition, we apply the latter on the set of partitions and then we update our parameters by choosing a fixed learning step (0.000001), we can repeat this action several times.

<u>SGD method</u>

The stochastic gradient descent (DMS), on the other hand, updates the parameters for each sample of learning v (i) and label w (i), the technique capable of removing redundancy by performing one update at a time.

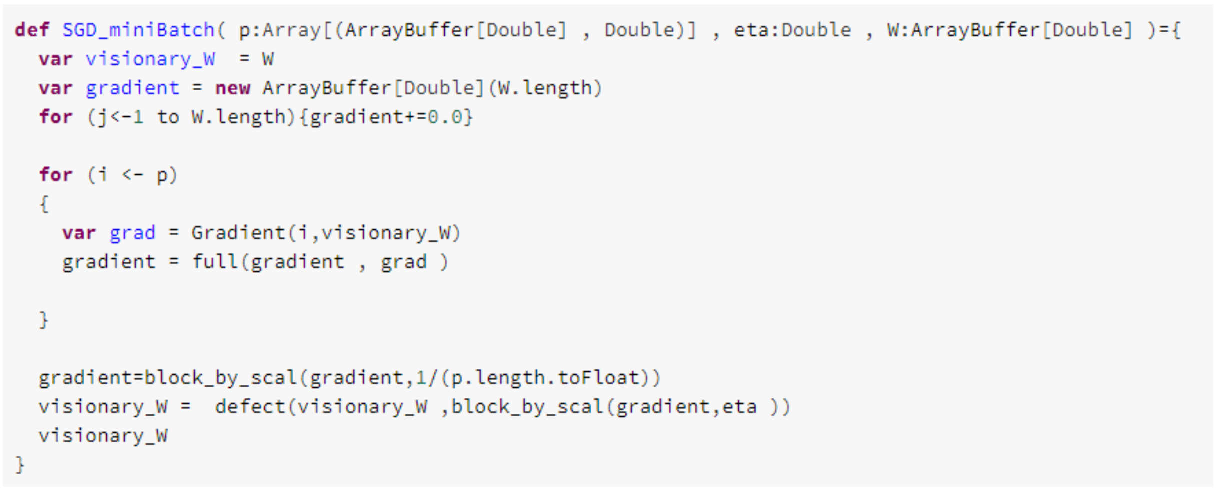
This is a function that calculates the SGD of a single partition (each partition receives the vector of the W parameters of the old partition), the latter is applied to the set of partitions by choosing a fixed learning step (0.00001), we can repeat this action several times.

```
//SGD require a higher computation effort to process each observation
def SGD( p:Array[(ArrayBuffer[Double] , Double)] , eta:Double , W:ArrayBuffer[Double] )={
  var visionary_W  = W
  var gradient = new ArrayBuffer[Double](W.length)
  for (i <- p)
  {
    var grad = Gradient(i,visionary_W)
    visionary_W =  defect(visionary_W ,block_by_scal(grad,eta ))
  }

  visionary_W
}
```

<u>SGD_MiniBatch method :</u>

This method consists of selecting a sample of our data set based on a specific fraction. After that, the gradient for each sample is calculated, in order to finally update the parameters.

Above we have defined a function that calculates the SGD_Mini Batch of a single partition, one applies the latter on the set of partitions, (each partition receives the vector of the pip parameters of the old partition) by also choosing a fixed learning step (0.000001) and a fraction (30% of a score), this action can be repeated several times.



```
def SGD_miniBatch( p:Array[(ArrayBuffer[Double] , Double)] , eta:Double , W:ArrayBuffer[Double] )={
  var visionary_W  = W
  var gradient = new ArrayBuffer[Double](W.length)
  for (j<-1 to W.length){gradient+=0.0}

  for (i <- p)
  {
    var grad = Gradient(i,visionary_W)
    gradient = full(gradient , grad )

  }

  gradient=block_by_scal(gradient,1/(p.length.toFloat))
  visionary_W =  defect(visionary_W ,block_by_scal(gradient,eta ))
  visionary_W
}
```

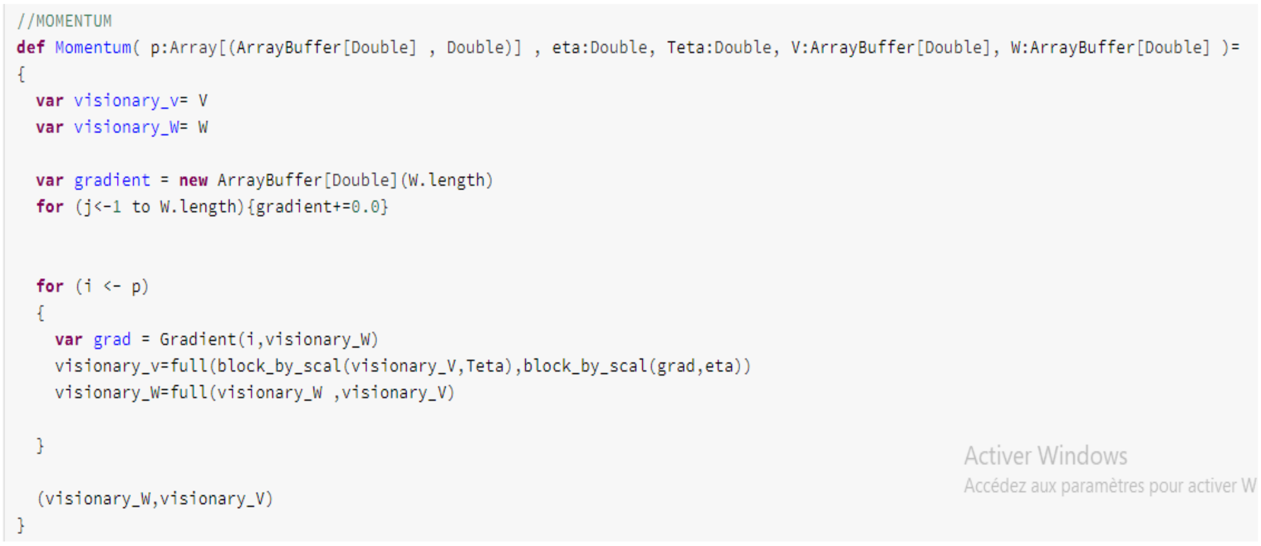
7 Methods for optimizing gradient descent

Momentum :

Momentum is a method that helps speed up the SGD in the proper direction and cushions Oscillations. It does this by adding a fraction of the update vector of the step spent to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

This is a function has been defined that calculates the Fast SGD_Momentum of a single score, applies the latter to the set of partitions (each partition receives the vector of the W parameters of the old partition) by choosing a fixed learning step (0.000001) and a Teta fraction (0.9), this action can be repeated several times.



```
//MOMENTUM
def Momentum( p:Array[(ArrayBuffer[Double] , Double)] , eta:Double, Teta:Double, V:ArrayBuffer[Double], W:ArrayBuffer[Double] )=
{
  var visionary_v= V
  var visionary_W= W

  var gradient = new ArrayBuffer[Double](W.length)
  for (j<-1 to W.length){gradient+=0.0}


  for (i <- p)
  {
    var grad = Gradient(i,visionary_W)
    visionary_v=full(block_by_scal(visionary_V,Teta),block_by_scal(grad,eta))
    visionary_W=full(visionary_W ,visionary_V)

  }

  (visionary_W,visionary_V)
}
```

Adagradv:

This optimization method adapts learning to the parameters, so as to perform larger updates for infrequent and smaller updates for updates less frequent day.
In its update rule, **Adagrad** changes the general learning step with each step of time t for each i setting based on past gradients that have been calculated for i:
we have defined a function that calculates the Fast SGD_Adagrad of a single partition, one applies the latter on the set of partitions (each partition receives the vector of the W parameters of the old partition) by choosing a learning step (0.0025) that will be modified as measures of iterations, this action can be repeated several times.

8 Results and comparisons

Batch: We noticed that using this method we have to calculate the gradients of dataset to perform a single update, the descent can be very slow and is untreatable for datasets that don't fit in memory. The gradient descent Batch method performs redundant calculations for large datasets, as it recalculates gradients for similar examples before each settings update.

- It is quite clear that the SGD_MiniBatch and Momentum method are achieving a convergence of faster than other methods, the parameters become stable as soon as they are first iterations.
- SGD (Momentum and Adagrad) gradation methods have enabled us to win convergence and reduce oscillations.In terms of execution time (in seconds):

	Batch	SGD	MiniBatch	Momentum	Adagrad
RDD	35.68	8.33	12.47	7.92	8.03
Dataframe	31.20	7.38	12.26	6.09	5.62
Datbase	21.66	5.65	14.12	5.75	5.42

9 Conclusion :

In analysing the execution times of the different types of gradient descent, it is quite clear that an optimization methods (Adagrad and Momentum) require much less time than optimization methods to be run, on the other hand we notice all methods are faster using a structuring Dataset-type data.