Universit Paris Dauphine PSL
Mastere Intelligence Artificielle et Sciences de Données (IASD)
Systems paradigms and algorithms for Big Data

Fast SGD with Adagrad and Momentum, comparisons with basic implementations

en utlisant les API de Apache Spark

Réalisé par:

Riadh Balti & Wael Chabir

Professeur:
Dario Colazzo

Tunis, Mai 2020

Table des matières

Introduction :

L'objectif dans ce projet est de faire la comparions entre Fast SGD Adagrad et Momentum avec les autre méthodes typiques de descente de gradient, ces méthodes utilisées pour l'optimisation de la descente de gradient qui a tendance à rencontrer des multiples challenges tels que des problèmes de convergence qui sont dus principalement aux mauvais choix du pas d'apprentissage approprié, parfois on se trouve à des problèmes où ce dernier est incapable de s'adapter aux caractéristiques d'un ensemble de données, et utilisant le même pas risque de non pas nous procurer une bonne convergence.

Dans ce qui suit on va comparer méthodes d'optimisation basées sur le SGD avec celles qui sont basiques (Batch, SGD, MiniBatch), on va implémenter un code en Scala en utilisant Spark, ceci en se recourant à trois types de structuration de données :

- RDD
- Dataframe
- Dataset

I. Problématique :

L'objectif était de calculer les différents types de descentes de gradient sur un problème de régression linéaire afin de trouver les paramètres optimales (poids W) permettant d'obtenir le minimum (local ou global) de la fonction d'erreur

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

La fonction d'erreur que nous avons utilisée est :

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

Pour calculer le gradient nous allons faire le dérivé de la fonction d'erreur :

$$2(\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{prediction}-\text{target}})\phi(x)$$

Pour faire la mise à jour des paramètres W, il suffit de choisir un pas d'apprentissage (n) et de la multiplier par le gradient puis on soustrait cette valeur de la valeur de W d'une itération antérieure

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

Train Loss (W) : est la somme de erreurs de tous l'ensemble d'apprentissage

II. Implémentation

Pour implémenter les étapes citées précédemment nous allons construire tout d'abord un ensemble de données d'apprentissage, qui va servir à entrainer notre fonction de gradient.

Pour ce faire nous avons adoptés cette méthode :

- Générer des données avec des valeurs de 1 à 1000 qui correspond à x

- Chaque valeur de x va servir à créer chaque ligne de notre ensemble de données qui sera de la forme : ((x1,x2),y)
  Avec :
  x1 :x
  x2 :x+5
  y :5x+2

1. Génération et répartition des données

Spark possède un majeur atout, il nous permet de faire un calcul parallèle, nous allons exploiter cet avantage pour calculer les différents types de descente de gradients en utilisant la notion de répartition.

La création de notre ensemble de données et sa répartition diffère d'une structuration de données à une autre, c'est-à-dire le typage des variables n'est pas le même dans RDD, Dataframe et Dataset, ainsi que ses répartitions :

- **RDD**

```
// Générer des données
val donnes=(1 to 1000).toArray
val instances=donnes.map(x=>(ArrayBuffer(x.asInstanceOf[Double],(x+5).asInstanceOf[Double]),(5*x+2).asInstanceOf[Double]))
```

On a générer les données dans un vecteur puis on utilise **.map** pour créer chaque ligne de cette forme :

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

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

Pour répartir nos données en 10 partitions, nous avons utilisé **.parallelize,** puis la fonction **.glom** qui retourne un RDD créé en fusionnant tous les éléments de chaque partition dans une liste, ensuite on attribue un indexe à chaque liste représentant une partition à l'aide **.zipWithIndex,** et enfin on inverse les positions de la liste et son indexe par un dernière **.map**

- **Dataframe :**

```
//Define class row
case class row(x1 : Double, x2:Double , y:Double)

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

Tout d'abord pour définir une ligne dans Dataframe on devrait définir une classe contenant x1, x2 et y sous forme d'un tuple, puis on a généré 1000 exemples dans un vecteur, transformé après en en Dataframe en utilisant la fonction **.toDF()**

```
val DF = Dataframe.repartition(10)
var DF_f =DF.withColumn("partition_ID", spark_partition_id)
```

Pour répartir nos données en 10 partitions, nous avons utlisé **.repartition,** puis on a créé une colonne contenant le numéro de la partition à laquelle appartient chaque exemple de données, ceci en utilisant la fonction **spark_partition_id** de spark SQL

- **Dataset:**

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

Pour créer une dataset nous avons suivre les mêmes étapes que celles de Dataframe sauf qu'on a remplacé la fonction **.toDF()** par une autre **.toDS().**

2. Calcul des gradients :

Pour pouvoir calculer la descente de gradient de plusieurs manières (Batch, MiniBatch ,SGD), il nous a fallu créer des nouvelles fonction intermédiaires qui sont :

- **prod_by_scal**(x:ArrayBuffer[Double],n:Double) : pour multiplier un vecteur par un scalaire
- **prod_scal** (x:ArrayBuffer[Double], w:ArrayBuffer[Double]) : Pour calculer le produit scalaire de deux vecteurs
- **somme**(x:ArrayBuffer[Double],y:ArrayBuffer[Double]) : pour faire la somme des vecteurs
- **difference**(x:ArrayBuffer[Double],y:ArrayBuffer[Double]) : Soustraction de deux vecteurs
- **grad**(x:(ArrayBuffer[Double],Double),w:ArrayBuffer[Double]) : cette fonction va servir de calculer le gradient d'un seul exemple(Instance)

Méthode Batch

Cette méthode consiste à parcourir tout l'ensemble des données pour calculer le gradient de chaque instance avant de faire une seule mise à jour des paramètres

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

```
//calcul du gradient d'une partition
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
}
```

Ci-dessus on a défini une fonction qui calcule le gradient d'une seule partition, on applique cette dernière sur l'ensembles des partitions et puis on fait la mise à jour de nos paramètres en choisissant un pas d'apprentissage fixe (0.000001), on peut répéter cette action plusieurs fois

Méthode SGD

La descente de gradient stochastique (SGD), en revanche, effectue une mise à jour des paramètres pour chaque exemple d'apprentissage x (i) et étiquette y (i), cette technique capable de supprimer la redondance en effectuant une mise à jour à la fois.

```
def SGD( p:Array[(ArrayBuffer[Double] , Double)] , alpha:Double , W:ArrayBuffer[Double] )={
  var nov_W  = W
  for (i <- p)
  {
    var grad_inst = grad(i,nov_W)
    nov_W =  difference(nov_W ,prod_by_scal(grad_inst,alpha ))
  }
  nov_W
}
```

Ci-dessus on a défini une fonction qui calcule le SGD d'une seule partition (chaque partition reçoit le vecteur des paramètres W de l'ancienne partition), on applique cette dernière sur l'ensembles des partitions en choisissant un pas d'apprentissage fixe (0.000001), on peut répéter cette action plusieurs fois

Méthode SGD_MiniBatch

Cette présente méthode consiste à choisir un échantillon de notre ensemble de données selon une fraction bien déterminée. Après cela, on calcule le gradient pour chaque échantillon, pour pourvoir enfin faire la mise à jour des paramètres.

```
//Compute SGD en utilisant des MiniBatch
def SGD_miniBatch( p:Array[(ArrayBuffer[Double] , Double)] , alpha:Double , W:ArrayBuffer[Double] )={
  var nov_W  = W
  var gd_Batch = ArrayBuffer.fill(W.length)(0.0)
  for (i <- p)
  {
    gd_Batch = somme(gd_Batch  , grad(i,nov_W) )
  }
  nov_W =  difference(nov_W ,prod_by_scal(prod_by_scal(gd_Batch,1/(p.length.toFloat)),alpha ))
  nov_W
}
```

Ci-dessus on a défini une fonction qui calcule le SGD_Mini Batch d'une seule partition, on applique cette dernière sur l'ensembles des partitions, (chaque partition reçoit le vecteur des paramètres W de l'ancienne partition) en choisissant aussi un pas d'apprentissage fixe (0.000001) et une fraction d'échantillonnage (30% d'une partition), on peut répéter cette action plusieurs fois.

3. Méthodes d'Optimisation de la descente de gradient

Momentum :

Momentum est une méthode qui aide à accélérer la SGD dans la direction appropriée et amortit les oscillations. Il le fait en ajoutant une fraction du vecteur de mise à jour du pas du temps passé vers le vecteur de mise à jour actuel

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

```
def Fast_SGD_Momentum( p:Array[(ArrayBuffer[Double] , Double)] , alpha:Double, Beta:Double, V:ArrayBuffer[Double], W:ArrayBuffer[Double] )={
  var nov_V=V
  var nov_W  = W
  var gd_Mom = ArrayBuffer.fill(2)(0.0)
  for (i <- p){
    var grad_inst= grad(i,nov_W)
    nov_V=somme(prod_by_scal(nov_V,Beta),prod_by_scal(grad_inst,alpha))
    nov_W =  difference(nov_W ,nov_V)
  }
  (nov_W,nov_V)
}
```

Ci-dessus on a défini une fonction qui calcule le Fast SGD_Momentum d'une seule partition, on applique cette dernière sur l'ensembles des partitions (chaque partition reçoit le vecteur des paramètres W de l'ancienne partition) en choisissant un pas d'apprentissage fixe (0.000001) et une fraction gamma (0.9), on peut répéter cette action plusieurs fois.

Adagradv:

Cette méthode d'optimisation adapte le pas l'apprentissage aux paramètres, de manière à effectuer des mises à jour plus importantes pour les mises à jour peu fréquentes et plus petites pour les mises à jour moins fréquentes.

Dans sa règle de mise à jour, Adagrad modifie le pas d'apprentissage général à chaque pas de temps t pour chaque paramètre i basé sur les gradients passés qui ont été calculés pour i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

```
def Fast_SGD_Adagrad( p:Array[(ArrayBuffer[Double] , Double)] , alpha:Double,  W:ArrayBuffer[Double] )={
  var nov_W  = W
  for (i <- p){
    var grad_inst= grad(i,nov_W)
    var G=prod_scal(grad_inst,grad_inst)+ 1e-8
    var nov_alpha = alpha * (1/sqrt(G))
    nov_W=difference(nov_W,prod_by_scal(grad_inst,nov_alpha))
  }
  nov_W
}
```

Ci-dessus on a défini une fonction qui calcule le Fast SGD_Adagrad d'une seule partition, on applique cette dernière sur l'ensembles des partitions (chaque partition reçoit le vecteur des paramètres W de l'ancienne partition) en choisissant un pas d'apprentissage (0.0025) qui va être modifié au fur et à mesures des itérations, on peut répéter cette action plusieurs fois.

III. Résultats et comparisions

- Batch : Nous avons remarqué qu'en utilisant cette méthode nous devons calculer les gradients de l'ensemble de données pour effectuer une seule mise à jour, la descente peut être très lente et est intraitable pour les ensembles de données qui ne tiennent pas en mémoire. La descente de gradient de la méthode Batch effectue des calculs redondants pour les grands ensembles de données, car elle recalcule les gradients pour des exemples similaires avant chaque mise à jour des paramètres.
- SGD: effectue des mises à jour fréquentes avec une variance élevée qui font que la fonction d'erreur fluctue fortement
- Il est bien clair que la méthode SGD_MiniBatch et Momentum atteignent une convergence des paramètres plus rapidement que les autres méthodes, les paramètres deviennent stables dès les premières itérations.
- Les méthodes d'accéleration de gradtion SGD (Momentum et Adagrad) nous ont permis de gagner une convergence plus rapide et réduisent les oscillations.
  En termes de **temps d'exécution** (en secondes):

	Batch	**SGD**	**MiniBatch**	**Momentum**	**Adagrad**
RDD	35.73	8.33	12.47	7.90	8.01
Dataframe	31.40	7.28	14.26	6.04	5.62
Dataset	21.63	5.55	14.12	5.75	5.42

En analysant les temps d'exécution des différents types de descente de gradient, il est bien clair d'une part que les méthodes d'optimisation (Adagrad et Momentum) nécessite beaucoup moins de temps pour être exécuté, d'autre part on remarque toutes méthodes sont plus rapide en utilisant une structuration des données de type Dataset.