Machine Learning on Big Data
Paris-Dauphine University

Vincent Gouteux, Louis Monier
Master IASD students

Project Report : Machine Learning on Big Data

Introduction :

In this project report, we deal with an article which was written by Sebastian Ruder and published in 2017 named An overview of gradient descent optimization algorithms.

The work presented in this article consists of reviewing different already existing optimization techniques. All the 11 described approaches are based on first order gradient descent techniques. They can be applied in the context of the search for global minimum of classic machine learning loss functions (such as in linear regression and logistic regression to give examples of the most basic and well-known ones).

First of all, we will go through each one of the aforementioned techniques by giving an express intuitive idea and/or basic maths formulations . Then, we show the results at the core of our project report : we combined the Map Reduce framework with 11 standard machine learning optimization techniques. This combination takes advantage of distributed computing to process enormous data volumes with high scalable parallelized algorithms.
More concretely, given a "toy" and a "real" datasets, we will compare the behaviors of classic basic methods like Stochastic Gradient Descent (SGD) before study more complex gradient descent algorithms.

The main objective of this project is to review, compare all these methods on criterion such as convergence speed, variance, scalability... All methods are implemented in PySpark. All the code is available in 2 Jupyter-Notebook (one per dataset) provided with this present report.

Keywords : *optimization, gradient descent, Map Reduce*

Course instructor : **Dario Colazzo**
dario.colazzo@lamsade.dauphine.fr

Contents

1 A Few Words About the Datasets

1.1 The "Toy" Dataset

Following on from the practical sessions of the *Machine Learning on Big Data* course, we first created and tested our implementations on an extremely simple dataset (which we call the "toy" dataset). It is based on 1000 randomly generated data points with Gaussian noise of mean 0 and standard deviation of 1. Then, we apply a linear transformation such as :

$$y = 5x + 2 + \epsilon$$

$$\epsilon \sim \mathcal{N}(0, 1).$$

$$x \sim \mathcal{N}(0, 1).$$

to create the targets (or "ground truths") of each data point.

The main goal is to find a model able to connect the data points to their targets in a supervised fashion. To do so, we choose a simple linear model $h_\omega := X\omega$. The idea is then to perform a linear regression with various gradient descent techniques to find the optimal weight w^* minimizing $\underset{w}{argmin}||Xw - y||^2$.

This simple dataset allows us to know the optimal $w^* = [2, 5]$ and use it as a reference to compute the Mean Square Error (MSE) for each gradient update. This value can also be analytically found thanks to this well-known formula : $w^* = (X^T X)^{-1} X^T y$

1.2 The Boston Housing Dataset

Once we demonstrated the various gradient descent techniques work on the "toy" dataset, we deployed on a more realistic one : The Boston Housing Dataset available at : https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html.
This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. There are a total of 506 examples and 13 features. This is a classic educational regression type dataset.

2 Gradient Descent Methods

The idea is to minimize the objective cost function :

$$\mathcal{J}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(h_\omega(x^{(i)}) - y^{(i)})^2 = ||Xw - y||^2 \tag{1}$$

where :

- $\mathcal{D} = \{x^{(i)}; y^{(i)}\}_{i=1}^{n}$ is the training dataset
- ω are the weights of the model $\in \mathbb{R}^{1xd}$
- h_ω is a linear approximator : $h_\omega(x^{(i)}) = \sum_{i=1}^{d} \omega_i x_j^{(i)} = \omega^T x_j := Xw$

Straightforwardly, we find that : $\nabla J(w) = 2X(Xw - y)$ Computing gradient descent on enormous volume of data can hamper performance. The combination of gradient descent techniques and the Map Reduce framework is becomes essential to exploit the largest existing datasets. For our implementations, we provide here the code we used to perform gradient descent together with a Map Reduce setting :

```
rdd = rdd.map(lambda x : 2 * ( np.dot(w, x[:-1]) - x[-1] ) * x[:-1])
rdd = rdd.reduce(lambda x,y : (x+y)) / n
```

NB) the last column of the x matrix x[-1] contains the target (named y above)

2.1 Vanilla Gradient Descent (aka Batch Gradient Descent)

The most famous and simple gradient descent method is the classic Batch Gradient Descent also known as Vanilla Gradient Descent. The update rule is as simple as follows :

$$w = w - \eta \nabla J(w)$$

2.2 Mini-Batch Gradient Descent

The Mini-batch Gradient Descent is almost exactly the same approach as the Vanilla GD except it only selects a subset of data points instead of the whole dataset to perform a gradient update. This results in faster computation time but higher variance due to the less accurate gradient estimator (because there are less data points to evaluate it).

To implement it concretely, we had to separate our data in different partitions then index those partitions and select them randomly to form our batches of data. In order to stay coherent, we made sure that we kept the same partitions for every method.

2.3 Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent is the most largely used method in the machine learning community. The method is almost exactly the same as the mini-batch GD but the batch size is fixed at 1 data point. Thus, data point are randomly-at-uniform selected in the dataset to perform successively gradient updates. For our implementation with Map Reduce, we enumerate every data point in the batch and update the weights parameter w at each step.

In the following, we compare the 3 methods detailed so far. The graph represents the Mean Square Error (or the loss) against the number of gradient updates performed.

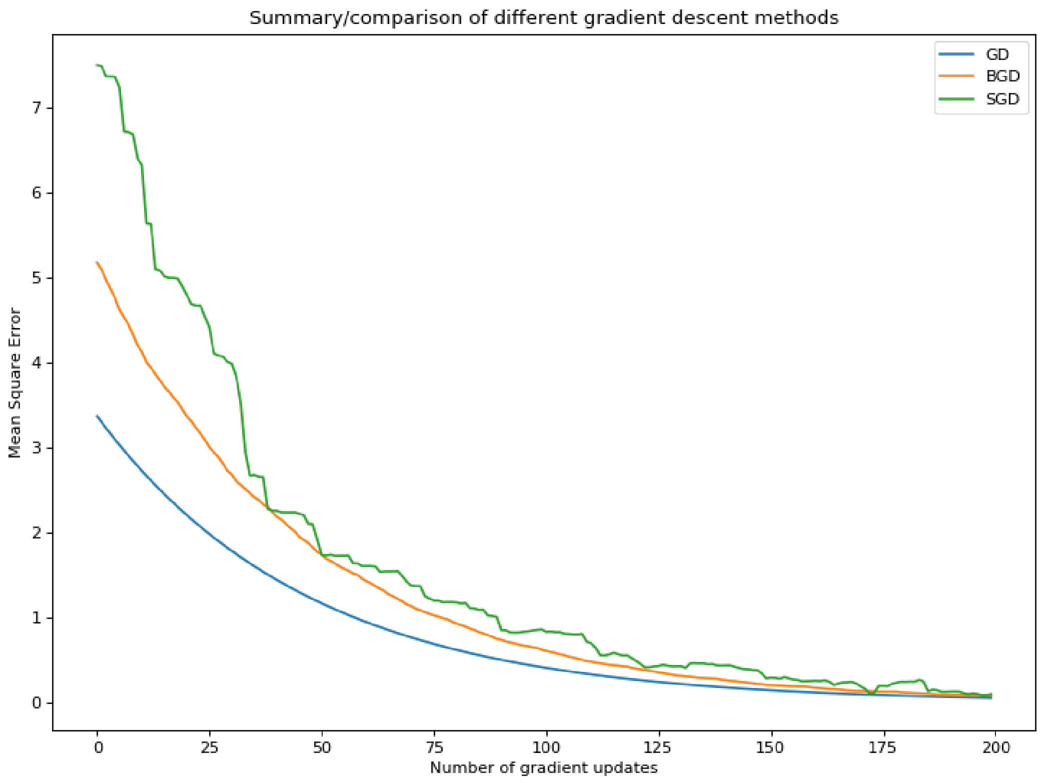

Figure 1: Comparison on the 3 most classical gradient descent

It is important to mention our datasets are not sufficiently small to be addressed with basic Vanilla GD and without the Map Reduce framework. Indeed, the graph shows fast convergence behavior. However, with enorme volume of data, Vanilla GD often fails due to the size of X matrix that can not be loaded into memory to compute the gradient. That is why, SGD (or Mini-batch GD) is often preferred over Vanilla GD.

2.4 SGD with Momentum

SGD with Momentum is a method that helps to accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector :

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

$$w = w - v_t$$

2.5 Nesterov Accelerated Gradient (NAG)

The Nesterov accelerated gradient (NAG) works exactly as SGD with Momentum except it takes the gradient of $w - \gamma v_{t-1}$ instead of w :

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

$$w = w - v_t$$

We performed hyperparameter tuning on the γ to find a range of optimal values. The following graph shows the MSE of the SGD Momentum for different values of γ :

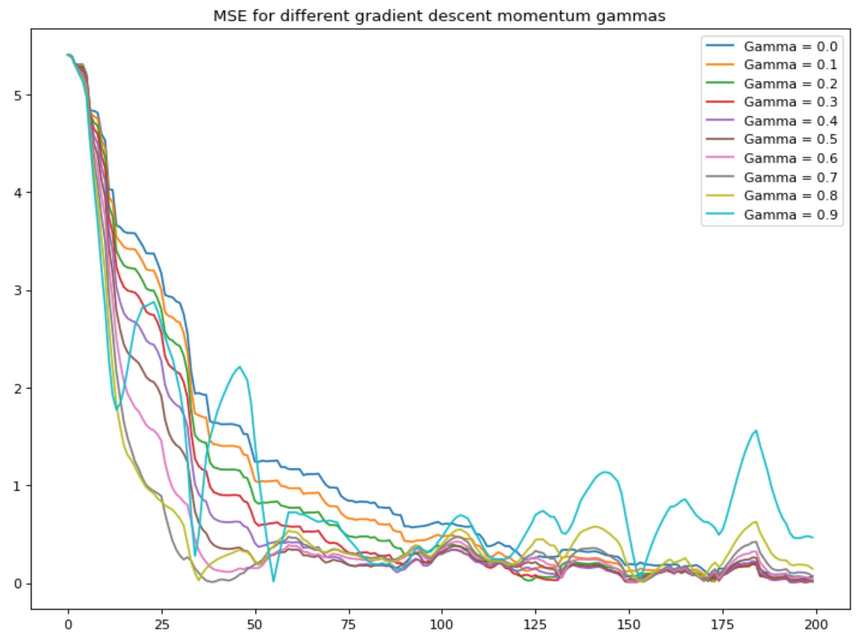

Figure 2: SGS Momentum gradient descent for different values of γ

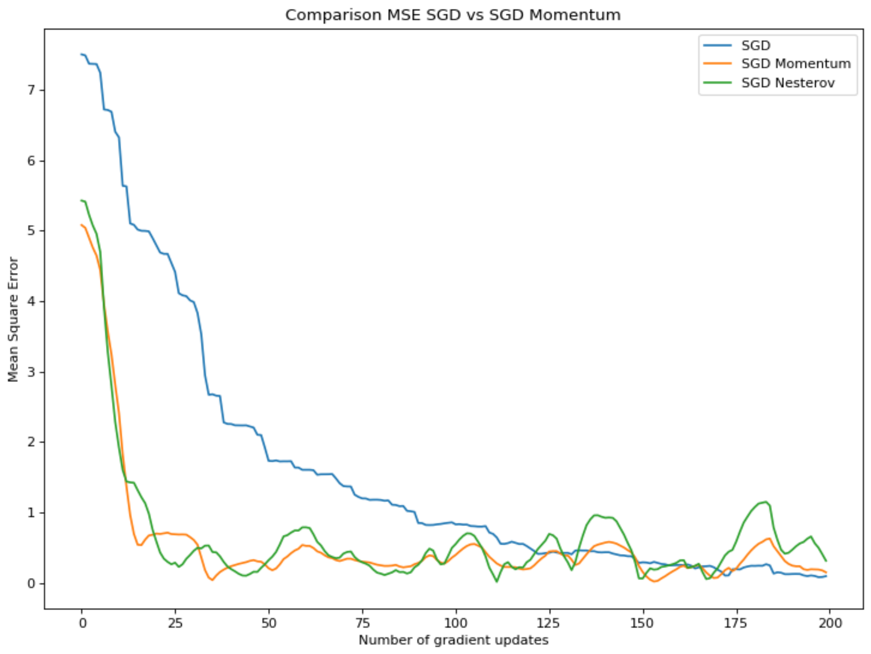

Figure 3: Comparison of SGD, Momentum SGD and NAG

2.6 Adagrad

The specifitcy of Adagrad is that it adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. We compute it with the formula below :

$$g_{t,i} = \nabla_{w_t} J(w_{t,i})$$

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

2.7 Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size :

$$\Delta w_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$$RMS[\Delta w]_t = \sqrt{E[\Delta w^2]_t + \epsilon}$$

$$\Delta w_t = -\frac{RMS[\Delta w]_{t-1}}{RMS[g]_t} g_t$$

2.8 RMSprop

RMS is a really close method of Adadelta. RMS as well divides the learning rate by an exponentially decaying average of squared gradients :

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_{t,i}$$

2.9 Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

2.10 Adamax

Adamax is very similar to Adam but we have a max in the update rule that gives better results and stability :

$$u_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) \mid g_t \mid^\infty = max(\beta_2 . v_{t-1}, \mid g_t \mid)$$
$$\omega_{t+1} = \omega_t - \frac{\eta}{u_t} \hat{m}_t$$

2.11 Nadam

As we have seen before, Adam can be viewed as a combination of RMSprop and momentum: RMSprop contributes the exponentially decaying average of past squared gradients, while momentum accounts for the exponentially decaying average of past gradients. We have also seen that Nesterov accelerated gradient (NAG) is superior to vanilla momentum. Nadam (Nesterov-accelerated Adaptive Moment Estimation) thus combines Adam and NAG. In order to incorporate NAG into Adam, we need to modify its momentum term m_t.

$$\Delta\omega_t = -\frac{RMS[\Delta\omega]_{t-1}}{RMS[g]_t} g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

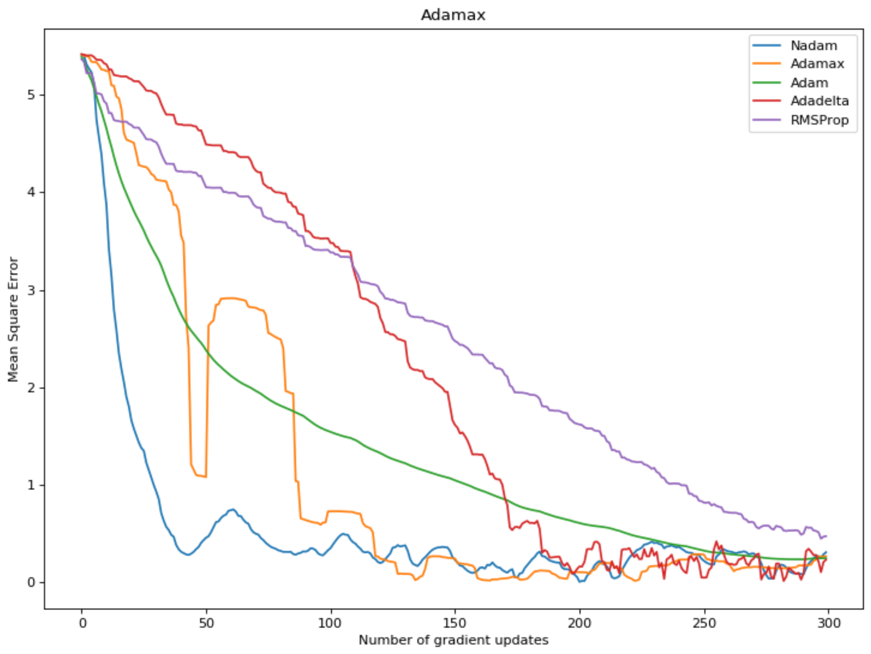

Figure 4: Comparison of Adagrad, Adadelta, RMSprop, Adam, Adamax and Nadam

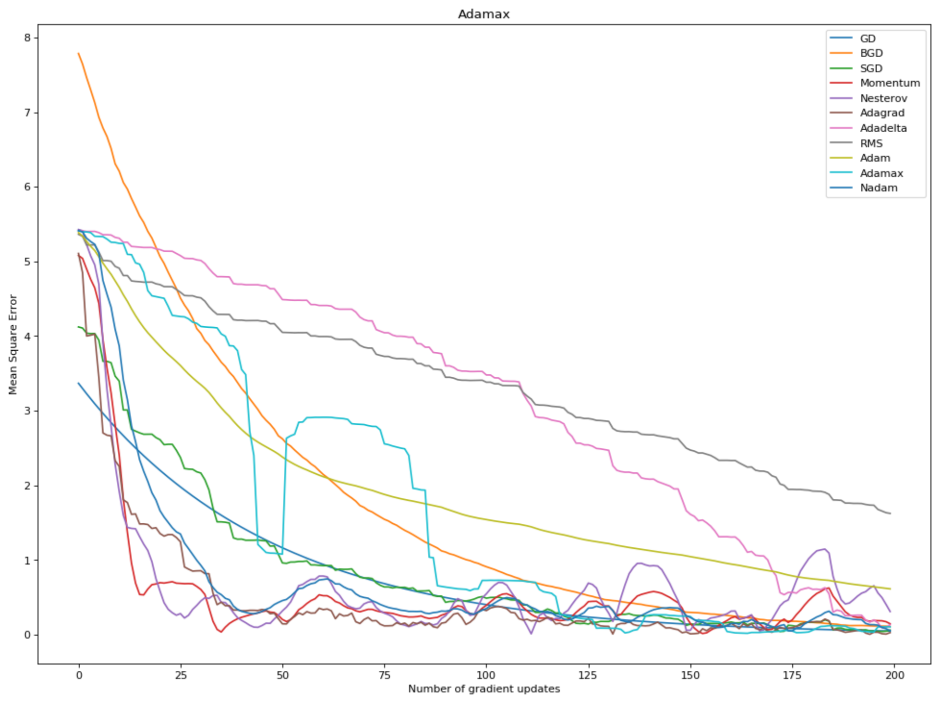

Figure 5: Comparison of all methods

3 Conclusion

All the graph showed in this notebook are the results of our gradient descent implementations on the ”toy” dataset. This demonstrates every approach described in the paper we implemented works well on this simple dataset. Once we made sure of the efficiency of all methods, we deployed them on a more realistic dataset : the Boston Housing Dataset. All the results and graphs are available in the corresponding notebook.

Important comment : To fairly compare all the methods on the Boston Housing dataset, we show the evolution of the loss against **the number of epochs** for all the plots of the notebook. This is different from the other ”toy” dataset notebook where we plot the evolution of the loss against the number of gradient updates.

To summarize, we provide in the report the global comparison between all methods on the Boston Housing dataset :

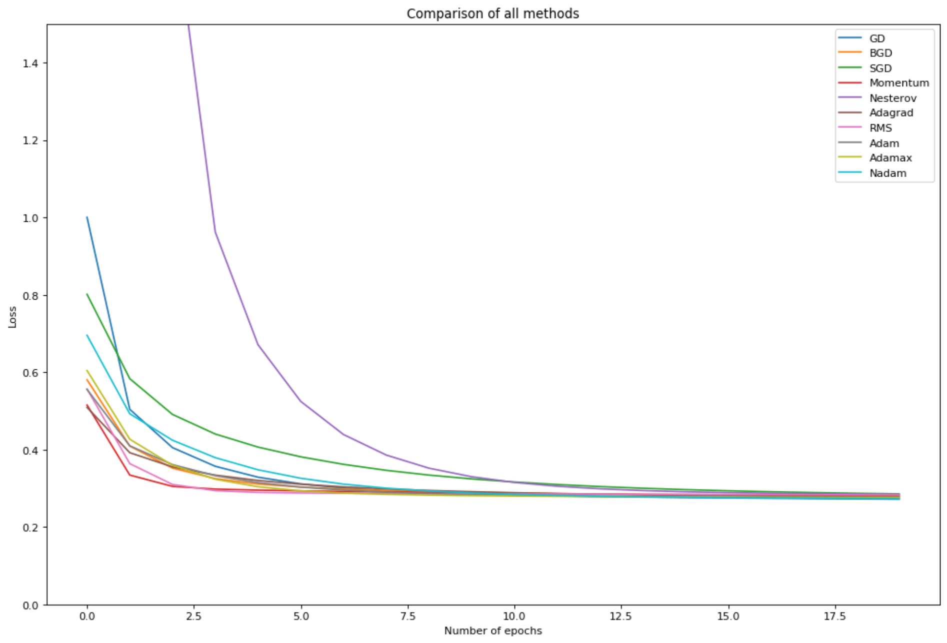

Figure 6: Comparison of all methods on the Boston Housing Dataset