PERFORMANCE EVALUATION OF MINI-BATCH LINEAR REGRESSION ON CPU AND GPU

Distributed Optimization and Games

Student: Simona Prudente

ABSTRACT

In this work I've examinated the time performances of a linear regression algorithm trained with mini-batch gradient descent when using Python for developing high-performance computing codes running on the graphics processing unit (GPU) and on the central processing unit (CPU). I've investigate in particular the performance of Compute Unified Device Architecture (CUDA).

1. INTRODUCTION

In order to perform the iterative minimization, one of the simplest methods is the gradient descent. The gradient descent is a first order iterative optimization method for finding the minimum of a function. How does it work? It's necessary to compute the slope of a function (which is the first-order derivative of the function at the current point) and move in the opposite direction with respect to the slope increase from the current point by the computed amount. Substantially there are two main methods to do this: the stochastic gradient descent (SGD) and the full batch gradient descent (FB). In SGD we have to start from a simple dataset point at a time to take a single step. Since we are considering just one example at a time, the cost fluctuates in the long run (it will never reach the minima but it will keep oscillating around it). SGD can be used for larger datasets because it converges faster as it causes updates to the parameters more frequently. In FB, the whole dataset is taken into account to take a single step. We take the average of the gradients of all the training samples and then use the mean gradient to update our parameters (the cost is a smooth curve and it keeps decreasing over the epochs). There is a method which takes the advantages of both methods mentioned above in order to reach better performances. It is the mini batch gradient descent. We use a batch of fixed number of training samples (which is less than the entire dataset), we calculate the mean gradient of mini-batch which corresponds to a single step in the optimization process. Mini batch ensures a tradeoff between time for iterations and accuracy.
In the following, the formula of the gradient, where n_k is the mini batch size. [1]

$$g(w_k, \xi_k) = \begin{cases} \nabla f(w_k; \xi_k) \\ \dfrac{1}{n_k} \displaystyle\sum_{i=1}^{n_k} \nabla f(w_k; \xi_{k,i}) \\ H_k \dfrac{1}{n_k} \displaystyle\sum_{i=1}^{n_k} \nabla f(w_k; \xi_{k,i}), \end{cases}$$

When dataset becomes very large, the CPU could start to perform in slower way the computations. For this reason there are several methods to speed up the computations and one of this is to deliver the computational load to GPU. GPU performs better with huge dataset because of its inner architecture. Let's take a look.

1.1 CPU vs GPU PROCESSING

Because GPUs can perform parallel operations on multiple sets of data, they are also commonly used for non-graphical tasks such as machine learning and scientific computation. Designed with thousands of processor cores running simultaneously, GPUs enable massive parallelism where each core is focused on making efficient calculations. While GPUs can process data several orders of magnitude faster than a CPU due to massive parallelism, GPUs are not as versatile as CPUs. CPUs have large and broad instruction sets, managing every input and output of a computer, which a GPU cannot do.

2. IMPLEMENTATION AND TESTS

In order to deploy the tests, I have chosen to use a linear regression problem with a single feature x.
The linear regression problem is the following one:

$$y = ax + b + e$$

in which a e b are parameters and e can be considered noise.

2.1 DATASET GENERATION and LINEAR REGRESSION

I have generated data by using with a vector of several points for the feature x and I've set a and b equals to 1 and 2 respectively. I've used all the dataset for training the model.

In order to differentiate the problem between CPU and GPU computations, I've used the library Pytorch because it permits to maintain the structure of the problem in both cases and because it allows pragmatically to solve the problem of speed up the computations. [2]

For training the model, I've performed two initialization steps:

- Random initialization of parameters a and b
- Initialization of hyper-parameters (in this case learning rate and number of epochs)

The random seed, to ensure reproducibility of results, has been set to 42.

For each epoch there are four training steps:

- Computation of predictions: $\mathrm{yhat} = \mathrm{a} + \mathrm{b} * \mathrm{x\_train}$
- Computation of loss function, that in my case is a MSE (Mean Square Error)
- Computation of the gradients for every parameter
- Update of parameters

In the implementation of the mini batch there is a function in the inner loop which takes a random smaller number of samples with respect to the whole set of datapoints.

In my case, I've performed three kinds of measures: the first one assuming as dataset population the number of pixels belonging to an image (approximatively), the second one corresponds to the number of pixels performed in a second of video and finally the third one is done considering even more pixels.

I've set the mini batch size equal to 50.

In order to check if my linear regression was performing well, I've used the library Scikit-Learn to verify if the values computed, after the training, were similar to the ones computed by my mini batch implementation.

My objective is to demonstrate that under some assumptions, the GPU computations are faster than CPU ones. In order to achieve this result, I’ve used a PyTorch library and in particular it supports CUDA tensor types, that implement the same function as CPU tensors, but they utilize GPU for computations (even more GPUs).

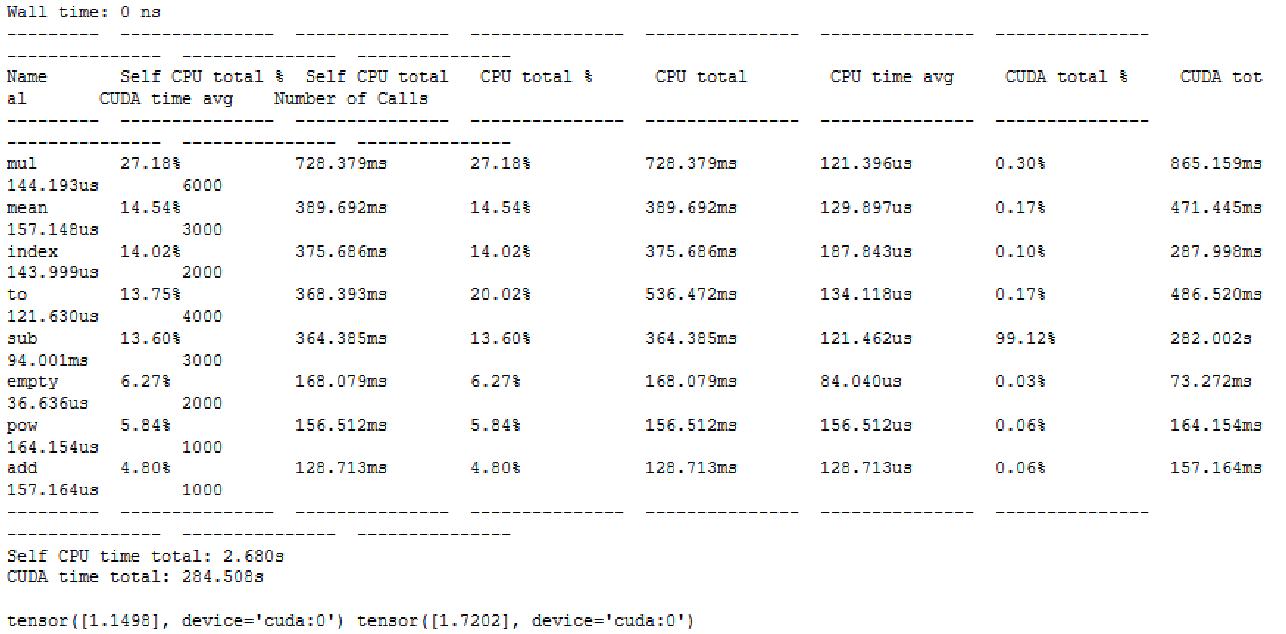
My data was in Numpy arrays, so I’ve transformed them into PyTorch’s Tensors and I’ve sent them to a chosen device: in this case the GPU.

The following step has been to create random parameters with PyTorch and I’ve assigned them to a device at the moment of their creation.

Once done all this preliminary stuff, I’ve executed the same code of mini batch gradient descent applied to a linear regression problem, with tensors.

In this way has been possible to compute the time performances thanks to the command line %time in case of CPU computations and, for what concerns GPU computations, I’ve put the mini batch implementation into some lines of codes capable to give me as output the profile of CPU and CUDA consumptions.

In particular, as seen in the following picture taken from my measurements, I’ve obtained the amount of time spent by CPU to send data on the GPU (2.680 s) and the relative time used by CUDA to compute my mini batch (284.508 s).



```
Wall time: 0 ns
---------  ---------------  ---------------  ---------------  ---------------  ---------------  ---------------
---------------  ---------------  ---------------
Name       Self CPU total %  Self CPU total   CPU total %      CPU total        CPU time avg     CUDA total %     CUDA tot
al       CUDA time avg    Number of Calls
---------  ---------------  ---------------  ---------------  ---------------  ---------------  ---------------
---------------  ---------------  ---------------
mul        27.18%           728.379ms        27.18%           728.379ms        121.396us        0.30%            865.159ms
144.193us        6000
mean       14.54%           389.692ms        14.54%           389.692ms        129.897us        0.17%            471.445ms
157.148us        3000
index      14.02%           375.686ms        14.02%           375.686ms        187.843us        0.10%            287.998ms
143.999us        2000
to         13.75%           368.393ms        20.02%           536.472ms        134.118us        0.17%            486.520ms
121.630us        4000
sub        13.60%           364.385ms        13.60%           364.385ms        121.462us        99.12%           282.002s
94.001ms         3000
empty      6.27%            168.079ms        6.27%            168.079ms        84.040us         0.03%            73.272ms
36.636us         2000
pow        5.84%            156.512ms        5.84%            156.512ms        156.512us        0.06%            164.154ms
164.154us        1000
add        4.80%            128.713ms        4.80%            128.713ms        128.713us        0.06%            157.164ms
157.164us        1000
---------  ---------------  ---------------  ---------------  ---------------  ---------------  ---------------
---------------  ---------------  ---------------
Self CPU time total: 2.680s
CUDA time total: 284.508s

tensor([1.1498], device='cuda:0') tensor([1.7202], device='cuda:0')
```

Figure 1 :output generated by computations on GPU (CUDA)

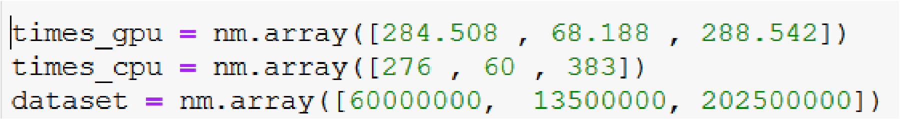
These computations have been done on a GPU GeForce GTX 1650 with 896 NVIDIA CUDA cores.[3]

3. CONCLUSIONS

The aim of my project was to show the difference between performances in CPU and GPU computations.

I’ve found that when the number of datapoints become very huge, the time performance of GPU is better than the CPU one.

In particular, as it’s possible to see in the following picture, three measurements have been done.



```
times_gpu = nm.array([284.508 , 68.188 , 288.542])
times_cpu = nm.array([276 , 60 , 383])
dataset = nm.array([60000000,  13500000, 202500000])
```

Figura 2: Trials

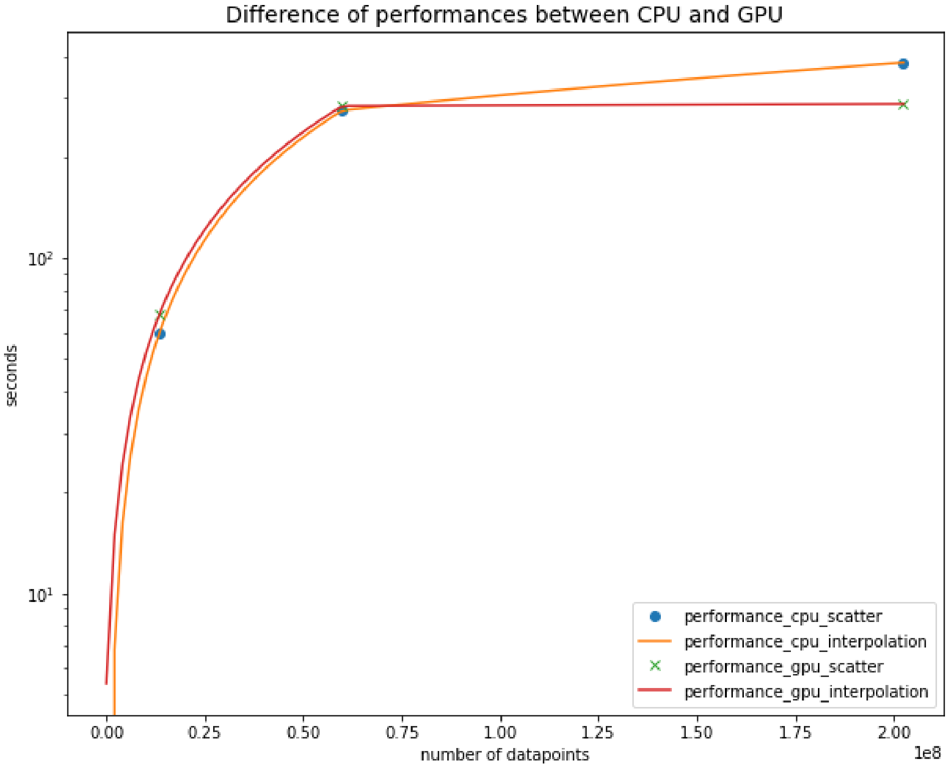


Figura 3: GPU and CPU performances

The first measurement has been done on 60000000 datapoints. The performances on CPU and GPU are almost the same. In case of the second measurement (by using less datapoints, let’s say 13500000 points), the performances generated are more or less the same as the first one.

The real gain of this measurements is the third one because it’s possible to see that by feeding the mini batch algorithm with a huge number of datapoints (in my case 202500000), the performances of GPU are better with respect to the CPU one of an amount of 100 seconds more or less. It hasn’t been possible to do trials with more points because of the limited memory dedicated to PyTorch library.

Nevertheless, a challenge could be the implementation of this algorithm by changing different parameters as the learning rate, the mini batch size, the number of iterations and, surely, by adding more and more datapoints to the train of the model.

An interesting approach would be also the possibility to implement this with several GPUs, so in a distributed fashion.

REFERENCES

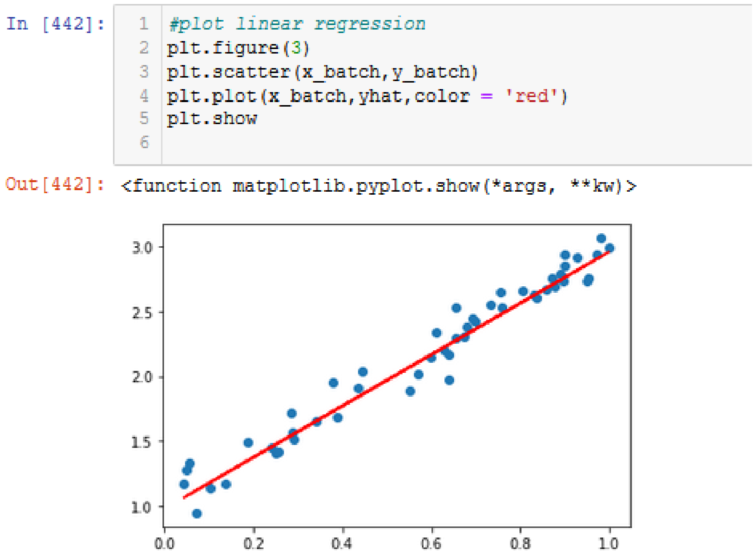
[1] Leon Bottou, Frank E. Curtis and Jorge Nocedal, Optimization Methods for Large-Scale Machine Learning, February 12, 2018

[2] https://pytorch.org/docs/stable/index.html

[3] https://www.nvidia.com/it-it/geforce/graphics-cards/gtx-1650/

APPENDIX

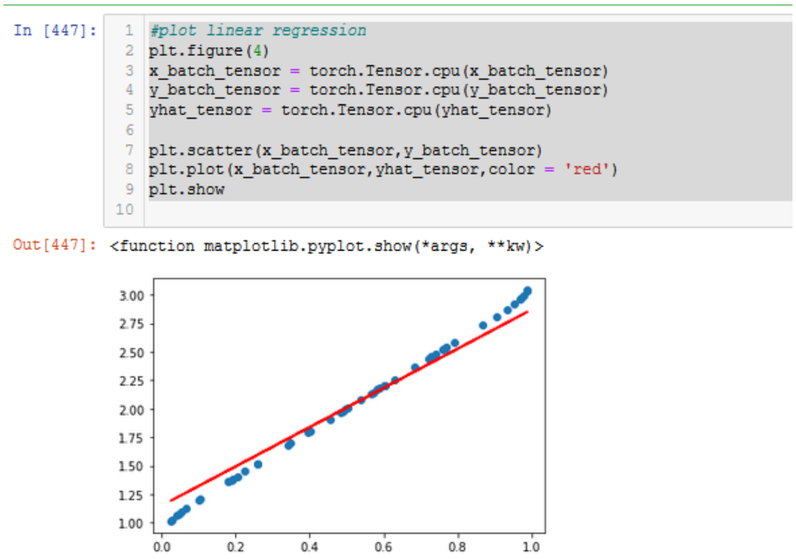
Linear regression performed on CPU:



```
In [442]:
#plot linear regression
plt.figure(3)
plt.scatter(x_batch,y_batch)
plt.plot(x_batch,yhat,color = 'red')
plt.show

Out[442]: <function matplotlib.pyplot.show(*args, **kw)>
```

Linear regression performed on GPU:



```
In [447]:
#plot linear regression
plt.figure(4)
x_batch_tensor = torch.Tensor.cpu(x_batch_tensor)
y_batch_tensor = torch.Tensor.cpu(y_batch_tensor)
yhat_tensor = torch.Tensor.cpu(yhat_tensor)

plt.scatter(x_batch_tensor,y_batch_tensor)
plt.plot(x_batch_tensor,yhat_tensor,color = 'red')
plt.show

Out[447]: <function matplotlib.pyplot.show(*args, **kw)>
```