Asynchronous Approximate Distributed Computation for Machine Learning First call report

Gianmarco Calbi

February 15, 2018

Context

Machine learning training

The most exploited method to train neural network is the Gradient Descent.

Gradient descent (GD)

Iterative method to achieve a local optimum of a continuously differentiable function (*convex/concave optimization always achieves* ***global*** *optimum*).

Definition (Empirical risk). The *empirical risk* E_n is a function measuring the performance of the predicted model over the training set, is the one we'd like to minimize.

$$E_n(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n} loss(f_{\vec{w}}(\vec{x}_i), y_i)$$

where:

- n is the size of the training set;
- the training set is defined as $\mathcal{X} = \{(\vec{x}_i, y_i) : i \in [n]\}$ where $\vec{x}_i$ is an input for the neural network and y_i is the correct and desired output for such input;
- $\vec{w}$ is the weight vector;

- $loss(y_i, f_{\vec{w}}(\vec{x}_i))$ is the *loss* function which measures the cost of predicting $f_{\vec{w}}(\vec{x}_i)$ (usually referred as $\hat{y}_i$) when the correct output should be y_i; obviously, $loss(y_i, \hat{y}_i) = 0 \Leftrightarrow \hat{y}_i = y_i$.

Several loss functions have been defined for different types of learning tasks, at the moment we are considering the *squared loss function*:

$$loss(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

so that E_n is nothing but the *mean squared error* of the trained function $f_{\vec{w}}(\vec{x})$ prediction over the training set.

GD is the most suitable method to achieve a $\vec{w}$ such that $f_{\vec{w}}$ can be retained a "good" approximation of the target function y. The *goodness* of the approximation is directly related to E_n, usually when $E_n(\vec{w})$ get less than or equal to a user-defined small real number ϵ then $f_{\vec{w}}$ can be retained a good enough predictor.

GD method consists in a stepwise update of $\vec{w}$ starting from an initial $\vec{w}_0$ (either fixed or randomly picked):

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \alpha \nabla_{\vec{w}} E_n(\vec{w}^{(t)}) = \vec{w}^{(t)} - \frac{\alpha}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)(-\hat{y}_i)'$$

where α is the *learning rate* (an higher learning rate value leads to fast convergence but less accurate solutions, while a smaller one do the opposite, it is always about finding a trade-off).

The computation is stopped either upon reaching a good enough solution (as explained before), or reaching a user-defined maximum number of iterations, or even stating divergence (this case may be avoided).

Stochastic gradient descent (SGD)

When the size of the training set is such that the stepwise update of $\vec{w}$ requires an unaffordable computation effort, than one would rather rely on the *stochastic gradient descent (SGD)*, that is a simplification of the classic method which outperforms GD w.r.t. single step speed and ensures (almost always under certain conditions) convergence, as well.

In SGD, the empirical risk E_n computed on the whole training set $\mathcal{X}$, is replaced with $E(\vec{w}) = loss(f_{\vec{w}}(\vec{x}_p), y_p)$ where $(\vec{x}_p, y_p)$ is a sample randomly picked from $\mathcal{X}$. Hence we aim to minimize the error just on a single sample rather than the error on the whole training set' samples. As stated before, under certain condition on the training set, approximating E_n with E will lead to converge as well.

The stepwise update of w finally becomes

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \alpha \nabla_{\vec{w}} E(\vec{w}^{(t)}) = \vec{w}^{(t)} - \alpha (y_p - \hat{y}_p)(-\hat{y}_p)'.$$

Batch gradient descent (BGD) behaves like GD but, instead of E_n, it computes E_m by taking into account only a small subset of $\mathcal{X}$ of size m instead of considering all the n samples in the training set.

Training in a distributed environment

Let the distributed system be composed of k computational nodes, then the gradient descent is performed following the steps below:

1. the training set is split into k disjoint subsets $\mathcal{X}_k$;
2. $\mathcal{X}_u$ is assigned to computational node u;
3. each node u, which owns a local weight vector $\vec{w}_u$, performs a single step update of it and obtains $\vec{w}_u^{(t)}$ (w at t-th iteration);
4. then partial solutions from all nodes are averaged to a single $\vec{w}^{(t)}$;
5. each node updates his local model with $\vec{w}^{(t)}$;
6. repeat from 3 until a stop condition is met.

In such kind of systems the weight update can be done either with GD or SGD, according to the size of the problem. Usually, the fact of working within a distributed system implicitly suggests that we're dealing with big datasets, therefore SGD is basically mandatory.

Bulk synchronous processing (BSP) gradient descent

Nowadays distributed systems usually implement BSP model for SGD computation. In such model each node u ending iteration t (and producing $\vec{w}_u^{(t)}$) is forced to wait for all the others to finish, in order to average all outputs to a single weight vector $\vec{w}^{(t)}$; only after such shuffle phase (called barrier) a node can go on to the following step $t+1$.

Definition (Dependency graph). Let $G = (V, E)$ be a digraph. G is called a *dependency graph* if its edges suggest a dependency relationship among vertices, e.g. $\forall e = (u, v) \in E$, v "depends on" u. In our case, a dependency graph suggests that, for an iteration t, before to proceed to the following one, each node v needs to average its own local weight vector with all its dependencies $u \in V : e = (u, v) \in E$ and $v \neq u$.

Asynchronous gradient descent

The dependencies among nodes within a BSP system can be rendered as a complete digraph $K_n = (V = [n], E = V \times V)$ composed of as many vertices as computational units in the system. The presence of a barrier after each stage (iteration) in BSP leads the overall computation dynamic to advance with time determined by slowest nodes. Indeed stragglers (nodes which experience a huge delay compared to the others) would heavely slow down the whole system. Removing barriers from BSP model should lead to straggler mitigation, e.g. an overall speed enhancement. The reasonable way to avoid barriers is to have few dependencies within the dep. graph.

Internship goal

We're going to test different graphs' configurations to assess the advantages of an asynchronous model with respect to BSP. In particular we're interested in discovering a trade-off between amount of dependencies, convergence speed improvements and correctness of the solutions.

Model overview

We would like to emulate the behavior of a distributed system (that is a cluster of machines running in parallel) within a single process (single-thread too, so far) to reproduce in details the distributed computation of SGD. Upon finding some interesting result we would also like to reproduce all tests on a real distributed environment.

Current results/outputs

Model implementation progress

At the moment the model can emulate a system with an arbitrary number of units computing either GD, SGD or BGD. The time taken by a node to perform a single step update of w is determined by exploiting the exponential distribution.

Model setup

Python application emulating a distributed system with 20 computational units.

I have defined 3 different distributed model w.r.t dependency graph.

1) Complete graph (BSP)

```
n = 20
adjmat1 = GraphGenerator.generate_complete_graph(n)
cluster = Cluster(adjmat1)
```

2) Cycle graph

```
n = 20
adjmat2 = GraphGenerator.generate_d_regular_graph_by_edges(
    n,
    ["i->i+1"]
)
cluster = Cluster(adjmat2)
```

Dependencies are set such that there is an edge from node i to node $i+1$.

3) A kind of expander graph

```
n = 20
adjmat3 = GraphGenerator.generate_d_regular_graph_by_edges(
    n,
    ["i->i+1", "i->i-1", "i->i+{}".format(int(n/2))]
)
cluster = Cluster(adjmat3)
```

Dependencies are set such that for node i there are three edges $(i, i-1)$, $(i, i+1)$ and $(i, (i+10)\%20)$.

Training set

The training set is generated by exploiting the function below

```
X, y = mltoolbox.sample_from_function(
    10000, 100, mltoolbox.LinearYHatFunction.f,
    domain_radius=10,
    domain_center=0,
    subdomains_radius=2,
    error_mean=0,
    error_std_dev=1,
    error_coeff=1
)
```

- *10000*: samples in the training set;

- *100*: number of features;
- *mltoolbox.LinearYHatFunction.f*: target function used to compute the correct y value for each $x \in \mathcal{X}$, in particular for this test is

$$f(x) =< X, \vec{w} >$$

- the domain will be discussed in the next paragraph;
- *error*: noise added to each sample target function value:

```
# pseudocode
# generate x matrix
noise = np.random.normal(error_mean, error_std_dev)
noise *= error_coeff
y[i] = f(x[i]) + noise
```

  error coefficient equal to zero leads to a noiseless training set.

Samples' domain. Since the target function is in the form $y =< X, \vec{w} >$, e.g.

$$y_i =< X_i, \vec{w} >= \sum_{j=1}^{k} x_{ij} w_j,$$

then the error function assumes the form of a paraboloid. To give evidence to our results, we would deal with functions on which SGD is supposed to be not trivial, so we are taking into account ill conditioned functions.

The actualization of such idea consists in taking samples features values from different domain's ranges.

Let k be the number of features, e.g. k is the dimension of each sample $\vec{x} \in \mathcal{X}$, then for each $j \in \{1, ..., k\}$, is randomly (from the uniform distribution) defined a compact $I_j \subset \mathcal{X}$ such that its radius is equal to *subdomains radius*. The key point is to consider a subsamples radius of orders of magnitude smaller than the domain radius.

(Actually I have to do more study and research on this part from a theoretical point of view, so don't get stuck if it doesn't seems so clear.)

Training phase setup

```
cluster.setup(
    X, y, mltoolbox.LinearYHatFunction,
    max_iter=4000,
    method="stochastic", # "classic", "stochastic", or "batch"
    batch_size=20, #matters only for batch GD
```

```
    activation_func=None,
    loss=mltoolbox.SquaredLossFunction,
    epsilon = 0.01,
    alpha=0.0005,
    metrics="all",
    shuffle=True
)
```

- the maximum number of iterations is set to 4000 *can be whatever one wants or can be even "None" or "math.inf" to never stop*;
- the method is set to stochastic but can be also batch or classic gradient descent;
- batch size matters only is the method is set to batch;
- activation function is set to "None", it means that the output is not normalized (the activation function is required for classification problems instead);
- I am using the classical squared loss $\frac{1}{2}(y_i - \hat{y}_i)^2$ as loss function;
- epsilon is the goodness threshold;
- actually alpha is what I've always called "learning rate";
- in metrics, one can specify the metrics the system should produce as output (mean squared error, mean average error, classification score, etc., or even "all" to compute all available metrics);
- shuffle specify whether the cluster may shuffle the training set before doing anything else, it's important when one doesn't know whether and/or how the samples in the training set are ordered *(usually it is worth to shuffle the training set whether not sure about its order).*

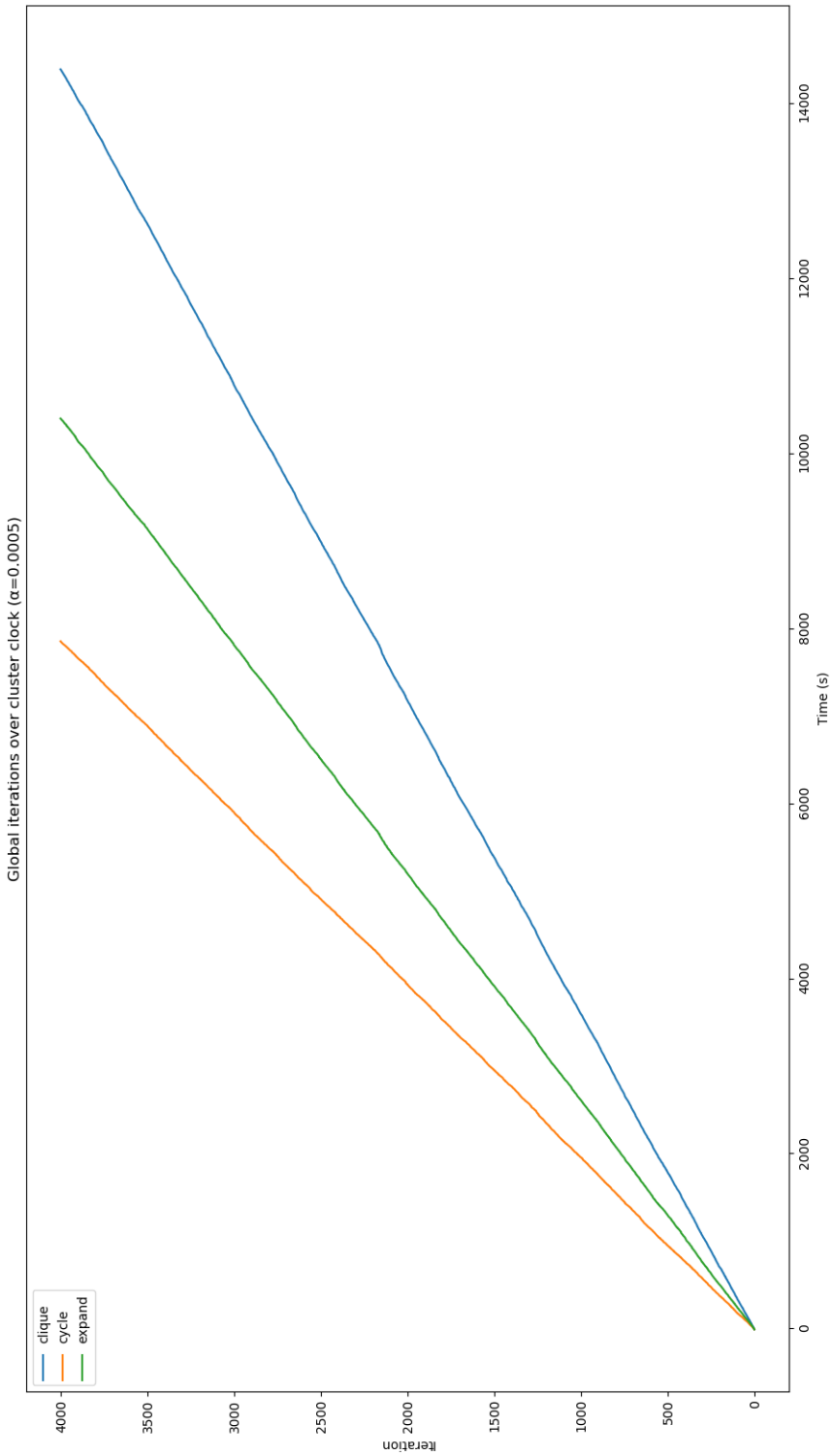
I made everything from scratch (e.g. all the functions etc. used have been defined by me), so I have the complete control over every single instruction within the computation.

Sample tests

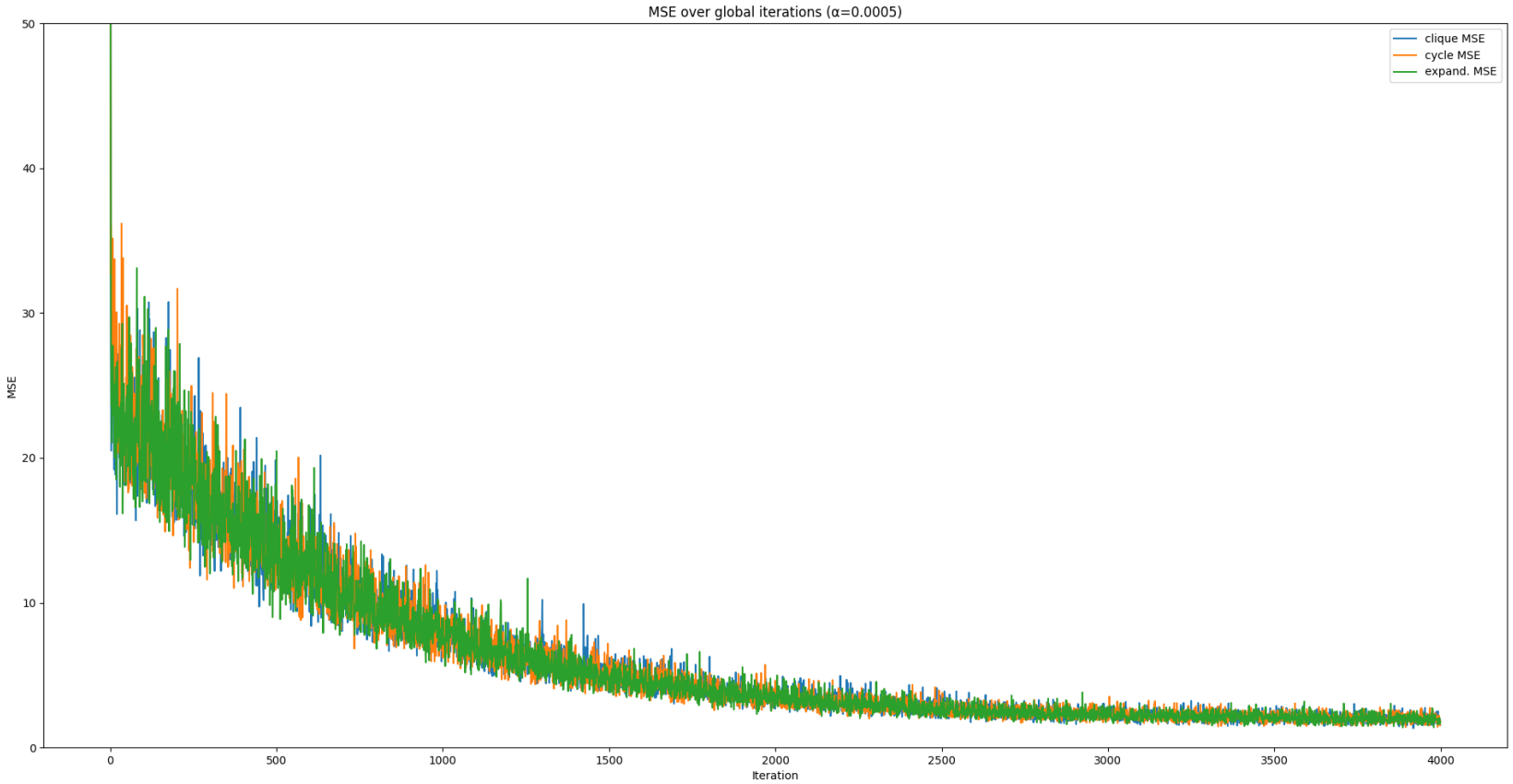
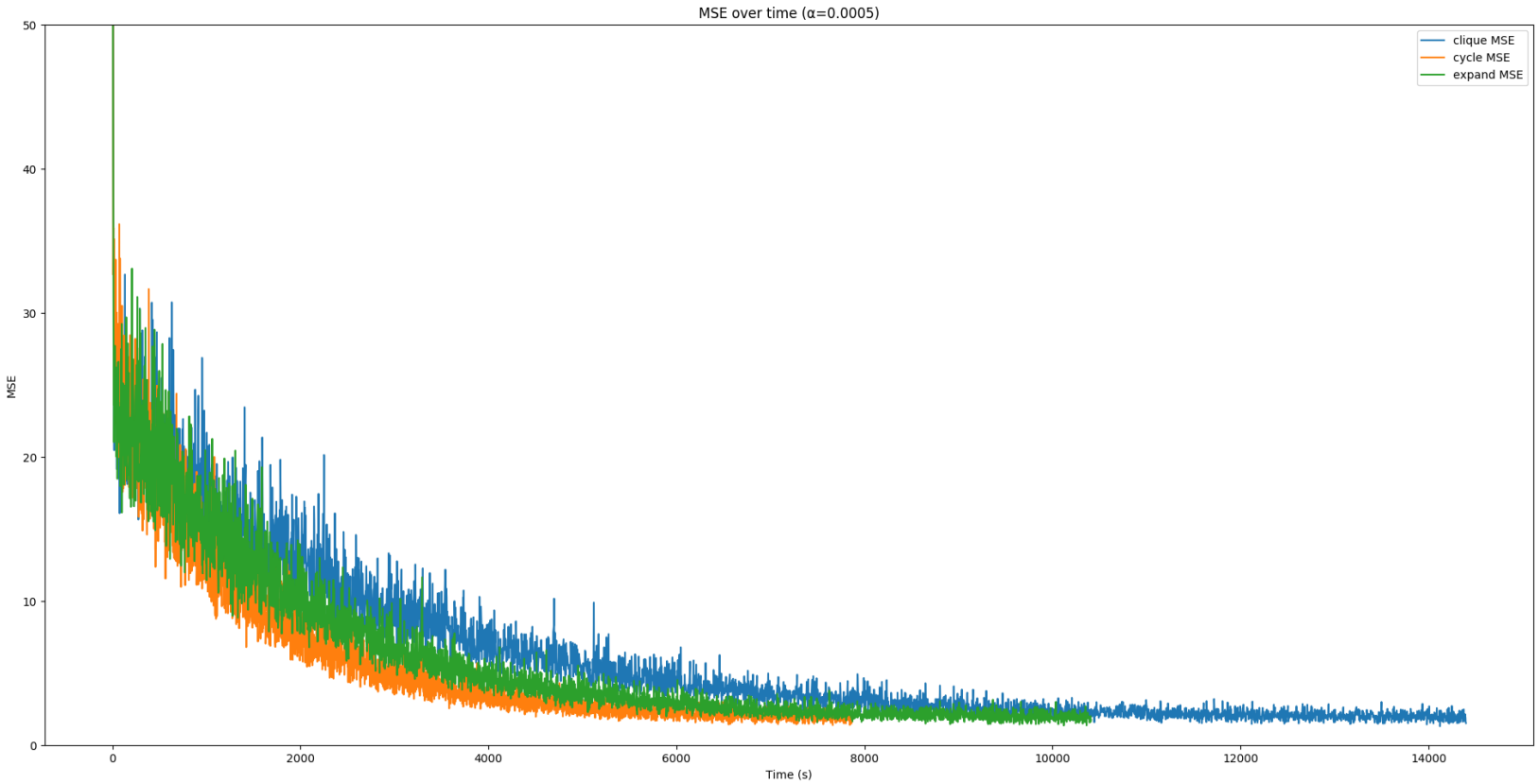
Actually I experienced some issues in producing the metrics plots, therefore don't get mad to give an interpretation to the graphs below, just note that, as expected, the clique is the most accurate w.r.t. the number of iteration but

is the worst if one considers the MSE over time; cycle and expander systems perform in a similar way but we are expecting that in future (and for sure more significant) tests this won't be so.

Global iterations over cluster clock



MSE over global iterations



MSE over cluster clock