Asynchronous Approximate Distributed Computation for Machine Learning First call report

Gianmarco Calbi

February 15, 2018

Internship brief description

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 an optimum that is* ***global***).

Definition 0.1 (Empirical risk). The *empirical risk* is a function measuring the training set performance, is the one we'd like to minimize.

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

where:

- n is the size of the training set;
- the training set is defined as $\mathcal{X} = \{(\vec{x}_i, y_i) : i \in \mathbb{N}\}$ 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;
- $loss(y_i, f_{\vec{w}}(x_i))$ is the *loss* function which measures the cost of predicting $f_{\vec{w}}(x_i)$ (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

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

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 $f_{\vec{w}}$ could be retained a "good" approximation of the target function y. The *goodness* of the approximation is directly related to E_n, usually E_n getting less than or equal to a user-defined small real number ϵ states that $f_{\vec{w}}$ is good enough predictor.

GD method consists in a stepwise update of w starting from an initial w_0 (either fixed or randomly picked):

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

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

The computation is stopped upon either reaching good solution (as explained right 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 the stepwise update of 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 given only by a single sample rather than the error on the whole training set' samples. As stated before, under certain condition on the training set, approximate E_n with E will lead to converge as well.

The stepwise update of w finally becomes

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

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 obtaining $\vec{w}_u^{(t)}$ (w at t-th iteration);
4. then partial solutions from all nodes are averaged to a single $w^{(t)}$;

5. each node updates his local model with $w^{(t)}$;

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 with SGD according to the size of the problem. Usually, the fact of working within a distributed system implicitly suggests that one is dealing with big datasets, therefore SGD is basically mandatory.

Bulk Synchronous Processing

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). Our intention consists, given a number N of computational units (nodes), in finding a trade-off between N and the number of connections established among each machine.

```
print('Hello!')
```

Model setup

Current results/outputs