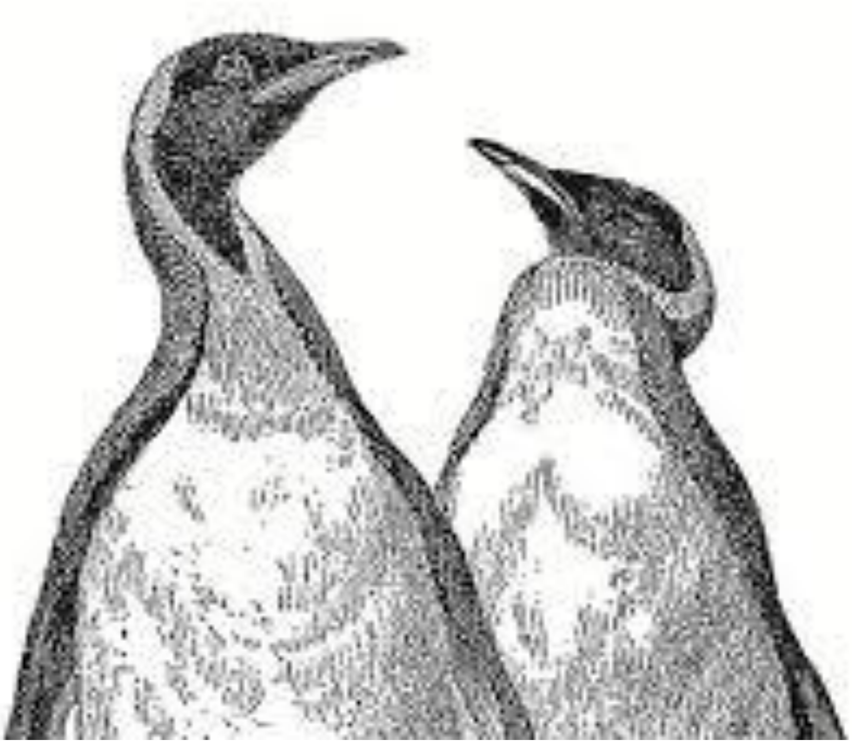
University of Pisa - MSc in Computer Engineering - Course of Concurrent and Distributed Systems

Scaling Up Distributed TensorFlow on GCP

A dissertation about performance of TensorFlow in distributed environment



Giuseppe Gagliano
14/02/2018

Index

1.Introduction

One of the main problem of today’s machine learning frameworks is to efficiently distribute computing in order to launch complex algorithms on huge data.

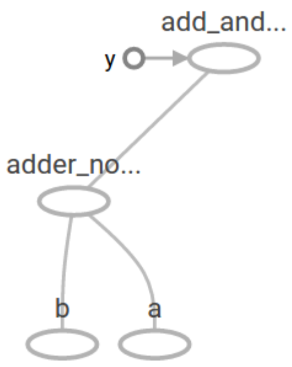
Moreover, the great popularity reached by scripting languages (i.e. Python, R, Julia, etc) in this field, brought necessity to abstract every aspect of the distributed environment to help data scientists to use simpler and well known APIs and paradigms. In this context (and many more) can be placed TensorFlow.

In the first part of this work, I’ll try to briefly explain how TensorFlow works and how it distributes computation to various devices. Then, I’ll tell how I tuned the available resources to find the best setup for the experiment. Finally I’ll show the results of experiments.

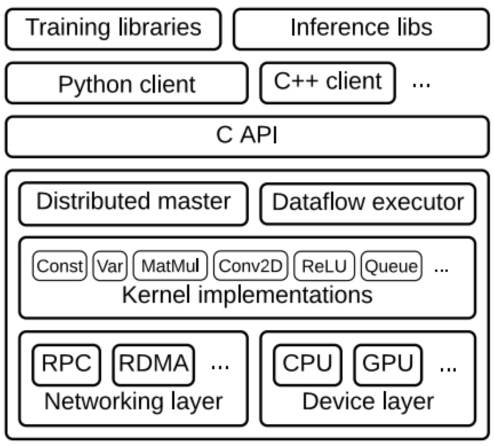
2. Tools

2.1 TensorFlow

TensorFlow is an open source software library for symbolic math and machine learning tasks using data flow graphs.



The key to TensorFlow, and the reason for its name, is that it takes a flow graph where the arcs correspond to tensors (multidimensional array) and the nodes are tensor operators. In TensorFlow all data is a multidimensional array and all operators combine tensors to produce new tensors. The resulting computation graph can be splitted and executed in parallel by several CPUs and GPUs.



The TensorFlow runtime is a cross-platform library, a C API separates user level code in different languages from the core runtime.

The distributed master translates user requests into execution across a set of tasks. Given a graph and a step definition, it prunes and partitions the graph to obtain sub-graphs for each participating device.

The dataflow executor (worker service) in each task handles requests from the master, schedules the execution of the local subgraph, and mediates direct communication between tasks.

The main highlights of TensorFlow are:

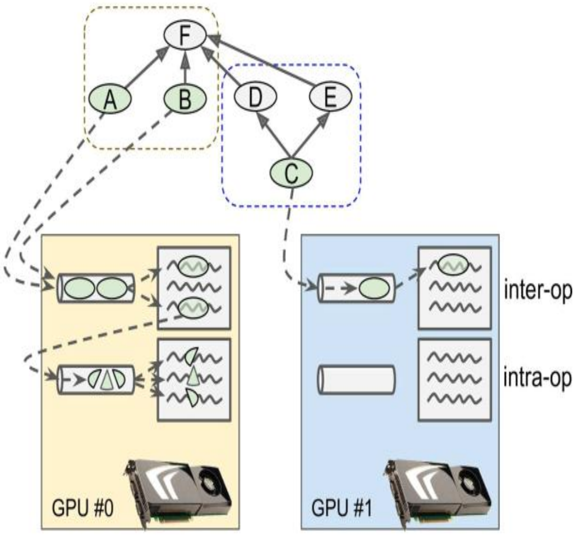
- It runs on Windows, macOS, Linux and from May 2017 also on mobile devices
- It provides a very simple Python API called *TF.Learn* compatible with Scikit-Learn
- High-level APIs have been built independently on top of TensorFlow (e.g. Keras)
- Its main Python API offers much more flexibility (at the cost of higher complexity) to create all sorts of computations, including any neural network architecture you can think of.
- It includes highly efficient C++ implementations of many ML operations, particularly those needed to build neural networks. There is also a C++ API to define your own high-performance operations.
- It also comes with a visualization tool called TensorBoard.

2.2 Distributed TensorFlow

TensorFlow's support of distributed computing is one of its main highlights compared to other neural network frameworks. It gives full control over how to split (or replicate) computation graph across devices and servers, and it lets parallelize and synchronize operations in flexible ways.

Multiple devices on single machine

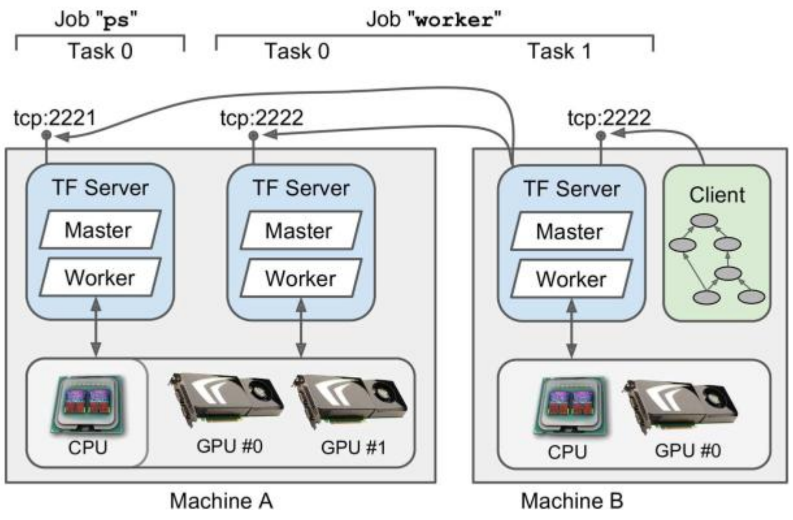
When TensorFlow runs a graph, it starts by finding out the list of nodes that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then starts evaluating the nodes with zero dependencies (source nodes). If these nodes are placed on separate devices, they get evaluated in parallel. If they are placed on the same device, they get evaluated in different threads, so they may run in parallel too (in separate GPU threads or CPU cores).



TensorFlow manages a thread pool on each device to parallelize operations. These are called the inter-op thread pools. Some operations have multithreaded kernels: they can use other thread pools (one per device) called the intra-op thread pools.

Multiple devices on multiple machine

A TensorFlow cluster is composed of one or more TensorFlow servers, called tasks, typically spread across several machines. A job is a named group of tasks that typically have a common role, such as keeping track of the model parameters (such a job is usually named "ps" for parameter server), or performing computations (such a job is usually named "worker").



The client uses the gRPC protocol (the letters "gRPC" are a recursive acronym which means, gRPC Remote Procedure Call) to communicate with the server.
This is an efficient open source framework to call remote functions and get their outputs across a variety of platforms and languages. It is based on HTTP2, which opens a connection and leaves it open during the whole session, allowing efficient bidirectional communication once the connection is established.

Every TensorFlow server provides two services. The **master** service allows clients to open sessions and use them to run graphs. It coordinates the computations across tasks, relying on the **worker** service to actually execute computations on other tasks and get their results.

One client can connect to multiple servers by opening multiple sessions in different threads. One server can handle multiple sessions simultaneously from one or more clients.

2.3 Google Cloud Platform

GCP consists of a set of physical assets, such as computers and hard disk drives, and virtual resources, such as virtual machines (VMs), that are contained in Google's data centers around the globe. Each data center location is in a global *region.* Each region is a collection of *zones*, which are isolated from each other within the region. Each zone is identified by a name that combines a letter identifier with the name of the region. For example, zone a in the East Asia region is named asia-east1-a.

Some resources can be accessed by any other resource, across regions and zones. These global resources include preconfigured disk images, disk snapshots, and networks. Some resources can be accessed only by resources that are located in the same region. These regional resources include static external IP addresses. Other resources can be accessed only by resources that are located in the same zone. These zonal resources include VM instances, their types, and disks.

Any GCP resources allocated must belong to a project which can be thought as a namespace. The resources that each project contains remain separate across project boundaries, can be interconnected through an external network connection. However, some resource names must be globally unique.

The *Google Cloud Platform Console* provides a web-based, graphical user interface to manage GCP projects and resources. Can be used to create a new project, or choose an existing project, and use the within that project.

An equivalent tool to GCP Console is the *gcloud command-line tool*. The gcloud tool can be used to manage both development workflow and GCP resources.

GCP also provides Cloud Shell, a browser-based, interactive shell environment:

- A temporary Compute Engine virtual machine instance.
- Command-line access to the instance from a web browser.
- A built-in code editor.
- 5 GB of persistent disk storage.
- Pre-installed Google Cloud SDK and other tools.
- Language support for Java, Go, Python, Node.js, PHP, Ruby and .NET.
- Web preview functionality.
- Built-in authorization for access to GCP Console projects and resources.

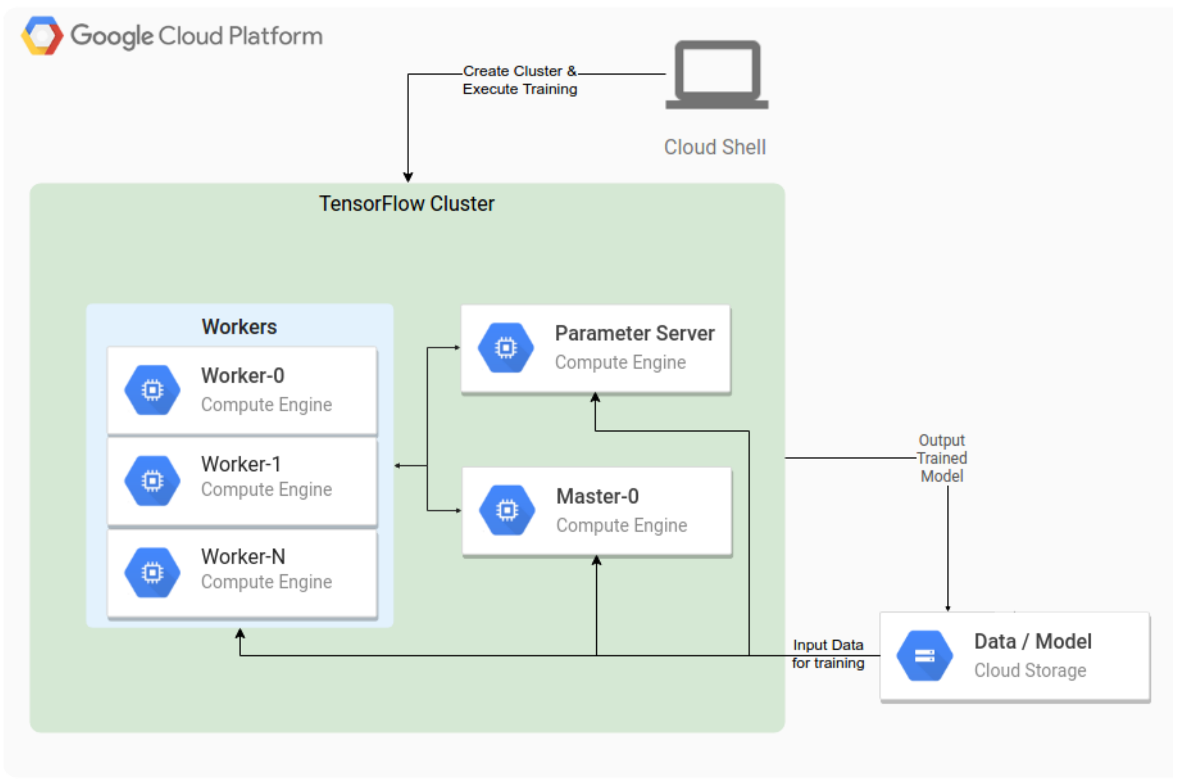
3. Project

3.2 Architecture

The architecture is the one represented in the figure below, I've used the Cloud Shell to create the instances, launch jobs and start TensorBoard server to show loss and accuracy.

The architecture needs at least 4 machines:

- *Client*, which in this case is the Cloud Shell
- *Master*, it is responsible for coordinating the work across the services.
- *Parameter servers*, whom transmit model parameters over the network
- *Workers*, whom do the compute intensive tasks (e.g. computing gradients)



3.3 Dataset

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

I’ve downloaded the dataset from https://storage.googleapis.com/cvdf-datasets/mnist/.

3.4 Model

The target model is a Convolutional Neural Network defined by the function *_cnn_model_fn()* within *model.py*, as follows:

- **Input Layer** MNIST dataset is composed of monochrome 28x28 pixel images, so the desired shape for our input layer is 28, 28, 1 (monochrome).
- **Convolutional Layer #1** Applies 32 5x5 filters (extracting 5x5-pixel subregions), with ReLU activation function
- **Pooling Layer #1** Performs max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap)
- **Convolutional Layer #2** Applies 64 5x5 filters, with ReLU activation function
- **Pooling Layer #2** Again, performs max pooling with a 2x2 filter and stride of 2
- **Dense Layer** 1,024 neurons, with dropout regularization rate of 0.4 (probability of 0.4 that any given element will be dropped during training)
- **Logits Layer** 10 neurons, one for each digit target class (0–9)

I’ve took the most of the code from a tutorial about running Distributed TensorFlow on Google Cloud (see 7. References).

3.5 Evaluation Metrics

In this project I’ve focused evaluation on the **speedup**, **execution time** and the number of **training steps per second** as metrics of scalabillity.

Since the nature of the task, I’ve also reported the results in term of accuracy and loss on the test set, although I didn’t tune the model to improve it.

4. Experiments

Using the free credits from GCP leads large limits on the evaluation of scaling up tasks in a distributed system. The major limitation is the GPUs unavailability, in fact, I only used CPUs in this dissertation. The other limit is the maximum number of virtual CPUs per zone, which is 8.

These constraints cut down a lot of possible configurations, so the major goal of this analysis is to get the best out of what's there.

The virtual instances in the following experiments were created from the same image, which contains the following software:

OS	Ubuntu 16.04
Python	2.7
Tensorflow	1.4

4.1 Initial tuning

In first analysis I've tried to discover which kind of machines are needed for the different TensorFlow services.

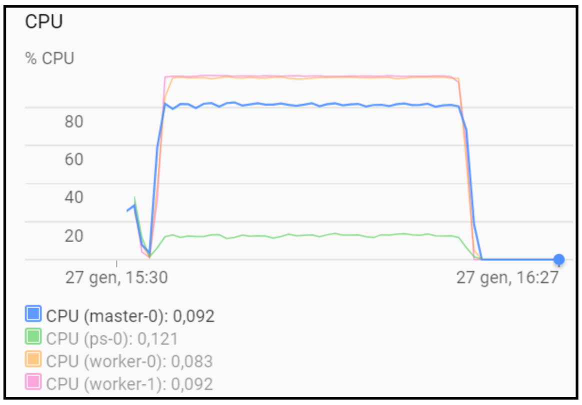
4.1.1 Configuration 1

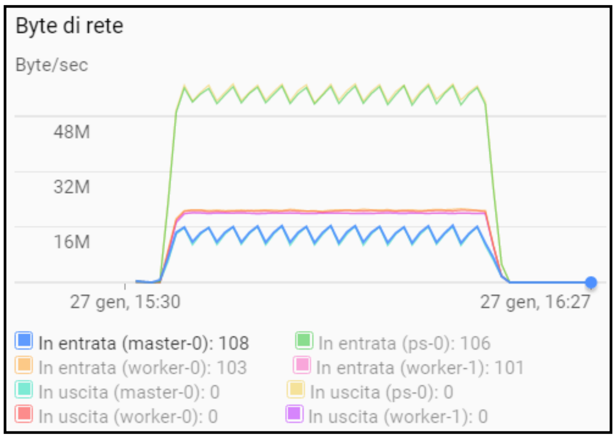
As initial configuration I've used the *n1-standard-2* type for all the services

Service	Instance type	# of instances	# of CPUs	RAM (GB)
Master	*n1-standard-2*	1	2	7.50
Parameter server	*n1-standard-2*	1	2	7.50
Worker	*n1-standard-2*	2	2	7.50

Training the model leads to the following results:

Training Time (s)	Training steps #	Accuracy
3253	10 003	0.9897





As can be seen both the Parameter Server (aka PS) and the Master don't need too much CPU, especially the PS who only need to send and receive parameters from the network.

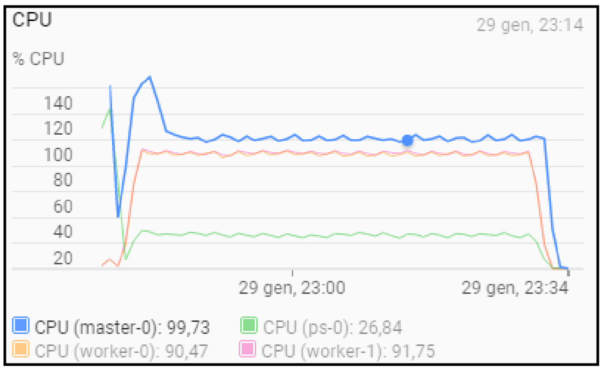
4.1.2 Configuration 2

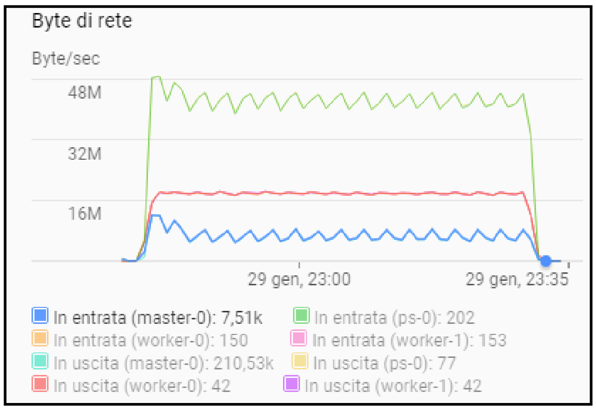
Since the test on Configuration 1 shown that PS and Master are quite idle from the CPU point of view, I've tried with smaller instance type for both:

Service	Instance type	# of instances	# of CPUs	RAM (GB)
Master	*n1-standard-1*	1	1	3.75
Parameter server	*n1-standard-1*	1	1	3.75
Worker	*n1-standard-2*	2	2	7.50

The results show how PS reaches good CPU utilization but Master is undersized:

Configuration #	Training Time (s)	Training steps #	Accuracy
2	3253	10 003	0.9897





This configuration seems to be undersized for the Master, but works well for PS, so, in the next experiment I'll take a *n1-standard-2* for the Master and a *n1-standard-1* for the PS.

4.1.3 Configuration 3

Finally, I've checked if having one single PS instance may limit the parallelization due to overhead. If it is , it should appear heavily when there are the maximum number of workers.

For this purpose I've tried to add one more PS instance with 4 worker instances, and I compared this configuration with respect to one PS Instance and 4 worker instances.

Here are the results:

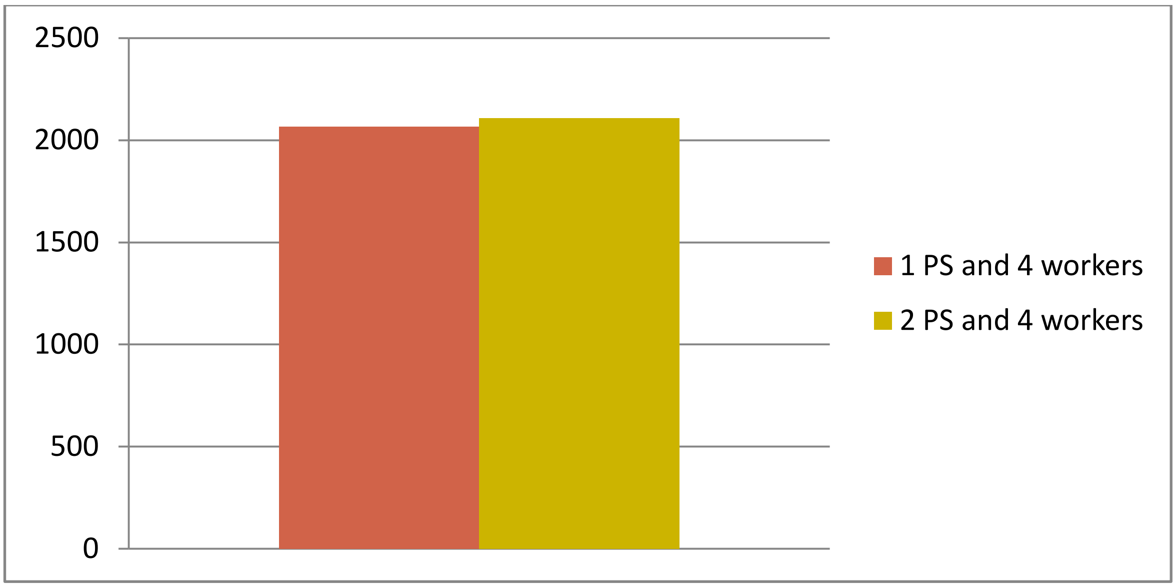
Service	Instance type	# of instances	# of CPUs	RAM (GB)
Master	*n1-standard-2*	1	2	7.50
Parameter server	*n1-standard-1*	2	1	3.75
Worker	*n1-standard-1*	4	1	3.75

The results show how PS reaches good CPU utilization but Master is undersized:

Configuration #	Training Time (s)	Training steps #	Accuracy
3	2 110	10 007	0.9904

As will be presented in the paragraph Scaling with 1 vCPU workers the training time using 4 worker instances and one PS is 2067 s.

From this last result it can be said that one PS instance is sufficient and adding one more instance wouldn't lead any improvement.



4.2 Non distributed execution

To have a wider understanding of the problem, I've trained the same model using a single workstation in a non distributed environment.

Moreover, I've executed the code on single vCPU and dual vCPU, since the experiments will involve these two cases. Here are the results:

# of vCPU	Training Time (s)	Training steps #	Accuracy
1	6677	10 003	0.9905
2	6251	10 007	0.9912

4.3 Scaling with 1 vCPU workers

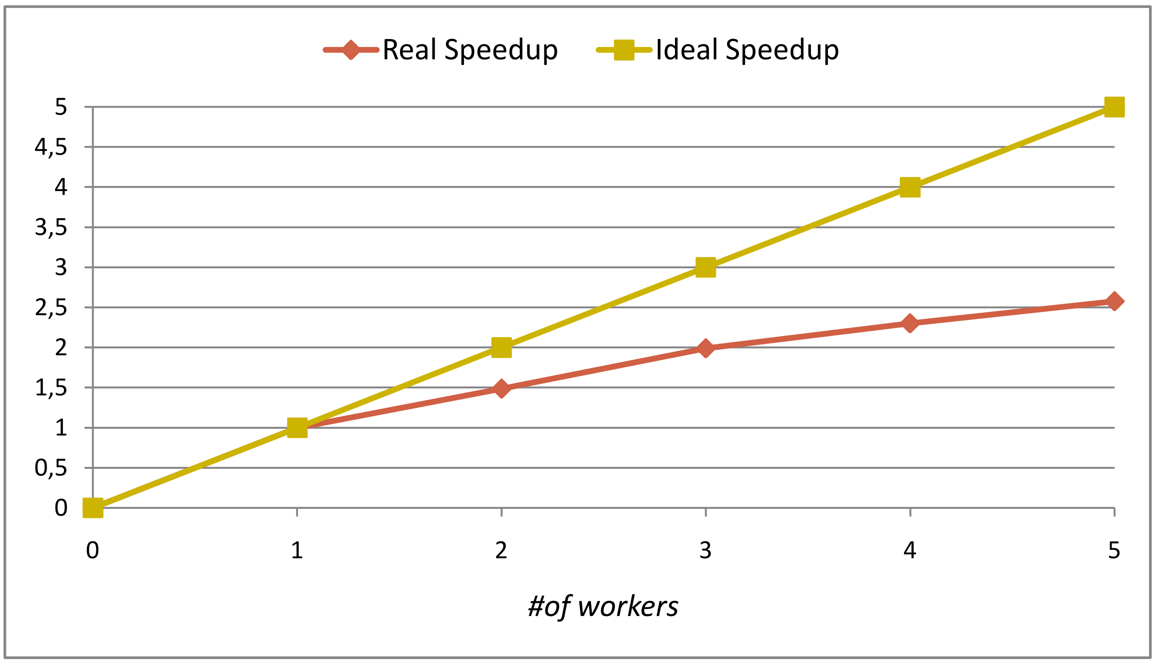
Once tuned PS and Master, I've trained the model increasing the number of *n1-standard-1* Worker instances. The configuration for these trials is the following:

Service	Instance type	# of instances	# of CPUs	RAM (GB)
Master	*n1-standard-2*	1	2	7.50
Parameter server	*n1-standard-1*	1	1	3.75
Worker	*n1-standard-1*	1 up to 5	1	3.75

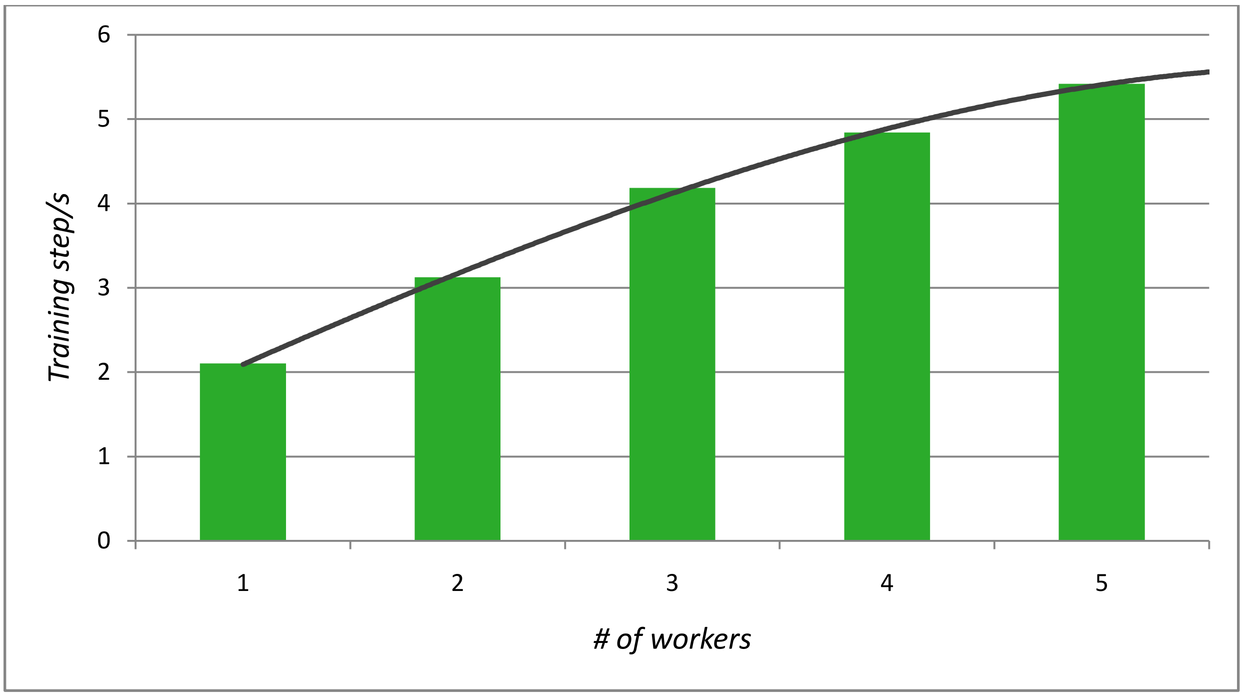
In this case the training task the model led to the following results:

# of Workers	Training Time (s)	Training steps #	Accuracy
1	4759	10 003	0.9897
2	3201	10 002	0.9916
3	2340	10 004	0.9916
4	2067	10 007	0.9906
5	1847	10 008	0.9903

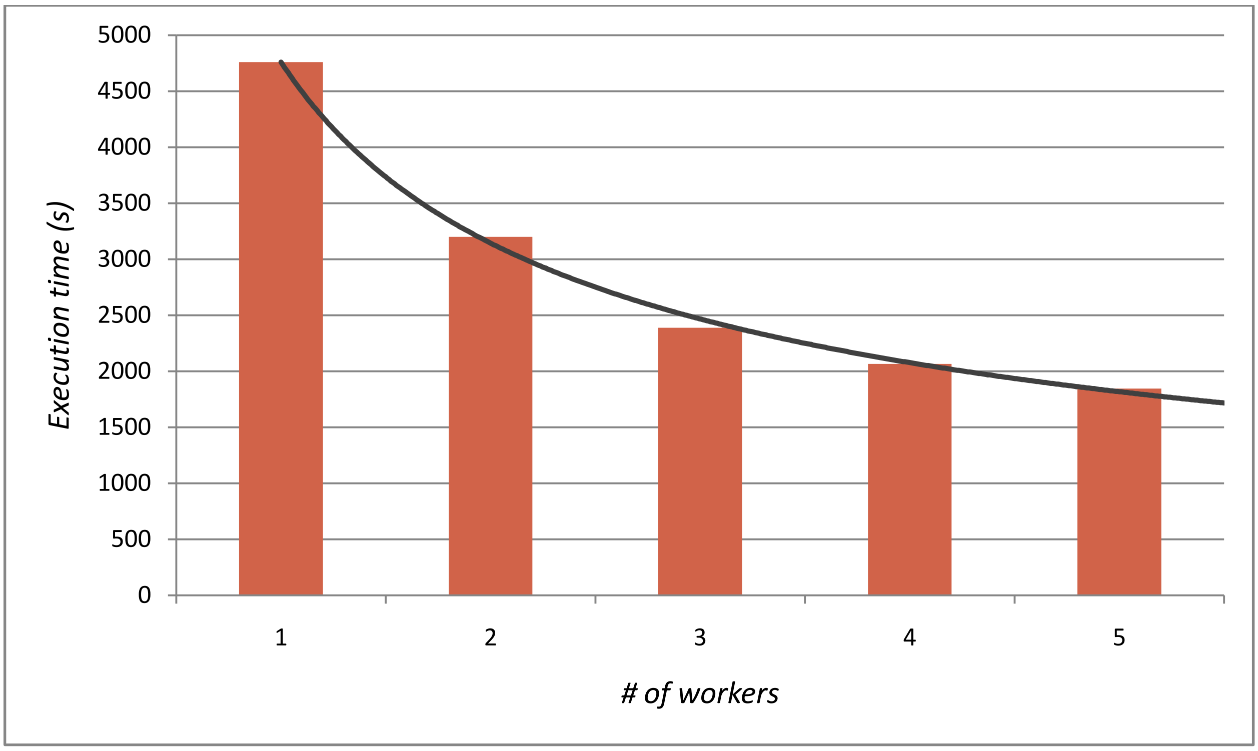
The next plot shows the results:



In the following plot I've reported the number of training steps per second (higher is better), highlighting the logarithmic trend of improvement adding more workers:



Finally I've plotted the execution time (lower is better), that, as expected decreases exponentially:



4.4 Scaling with 2 vCPU workers

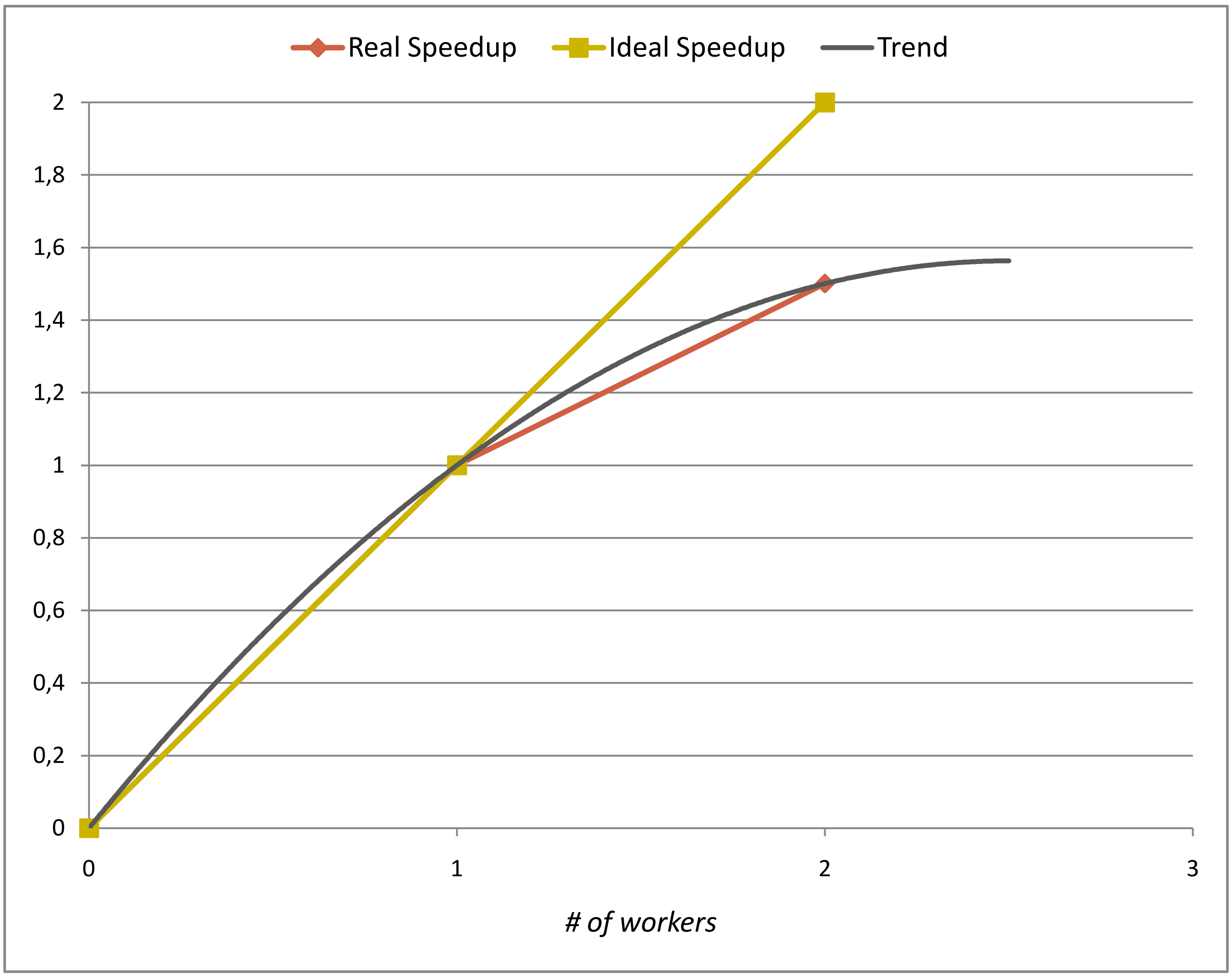
The other set of trials I've evaluated regards the following configurations:

Service	Instance type	# of instances	# of CPUs	RAM (GB)
Master	*n1-standard-2*	1	2	7.50
Parameter server	*n1-standard-1*	1	1	3.75
Worker	*n1-standard-2*	1 up to 2	2	7.50

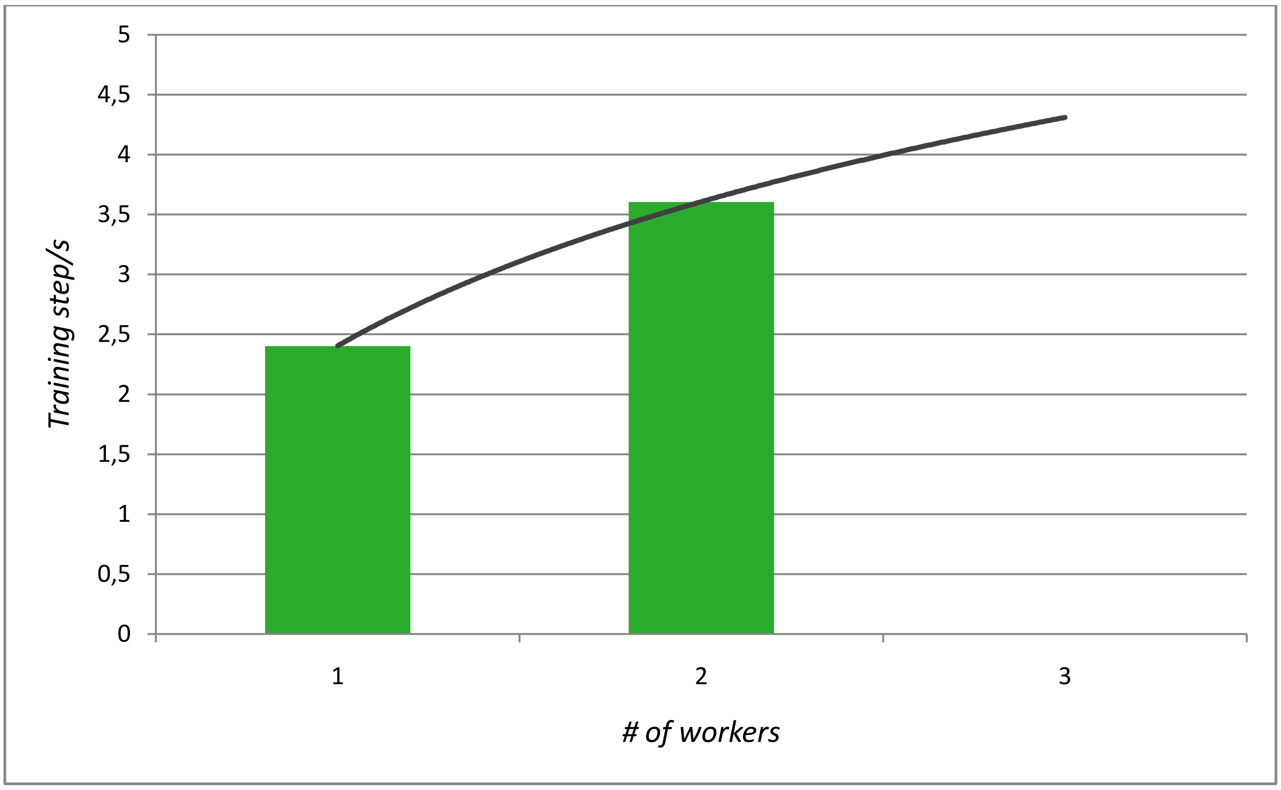
In this case, there are only 2 simulations since GCP doesn't allow to allocate more than 2 instances with 2 vCPUs. The results are the following:

# of Workers	Training Time (s)	Training steps #	Accuracy
1	4162	10 002	0.9901
2	2774	10 004	0.991

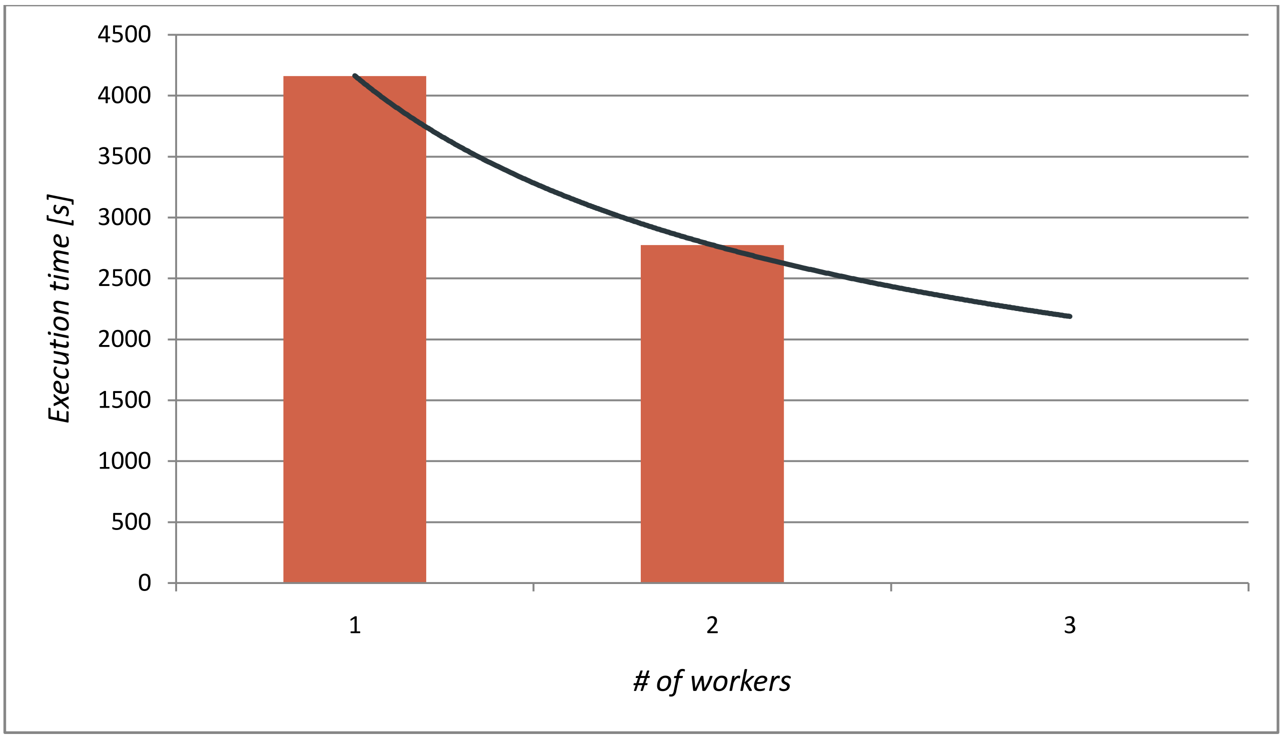
The speedup plot is quite poor with only 2 points, so I've added the best fitting regression just to hint the trend:



The training steps per second:



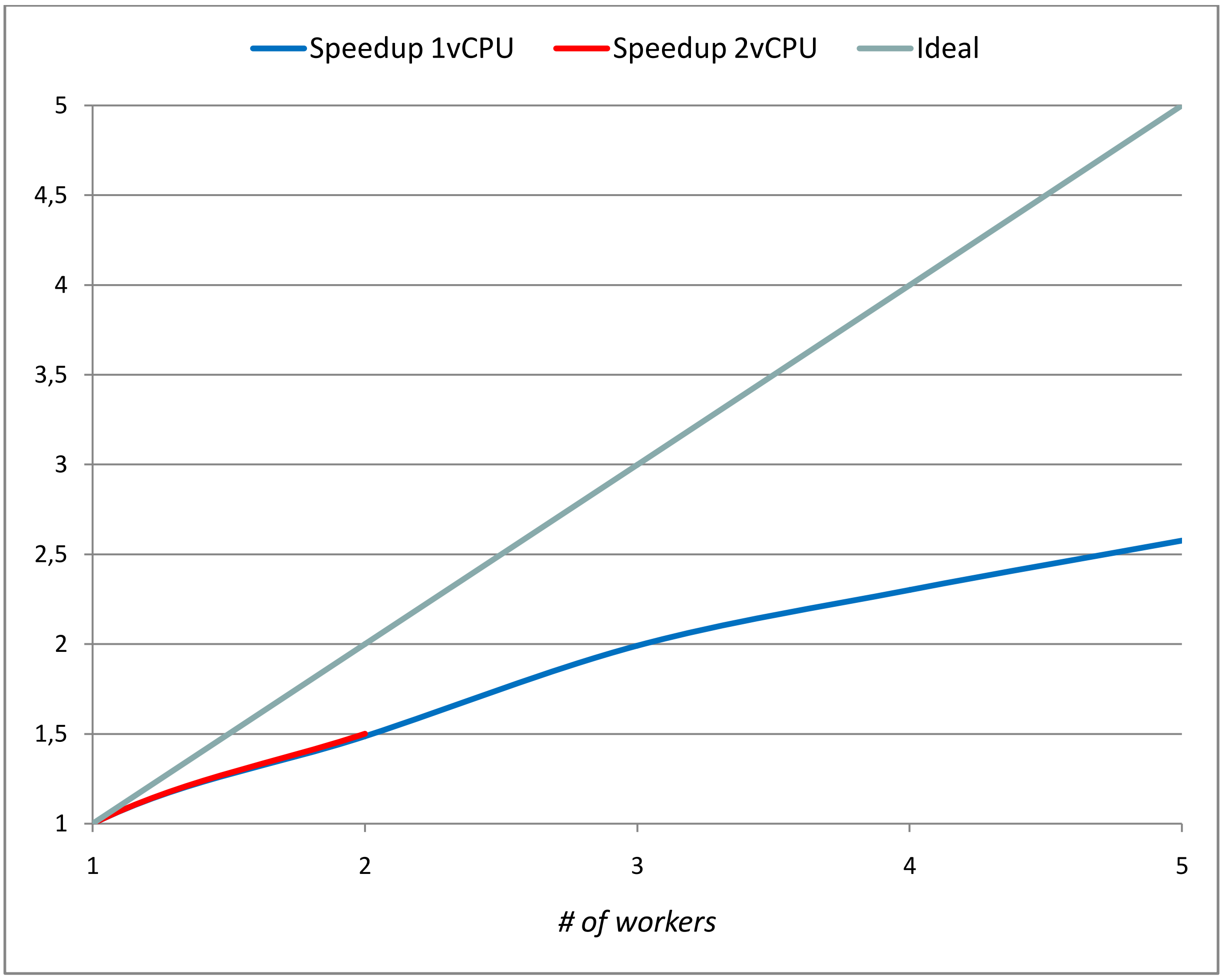
and the execution time



5. Results

Speedup

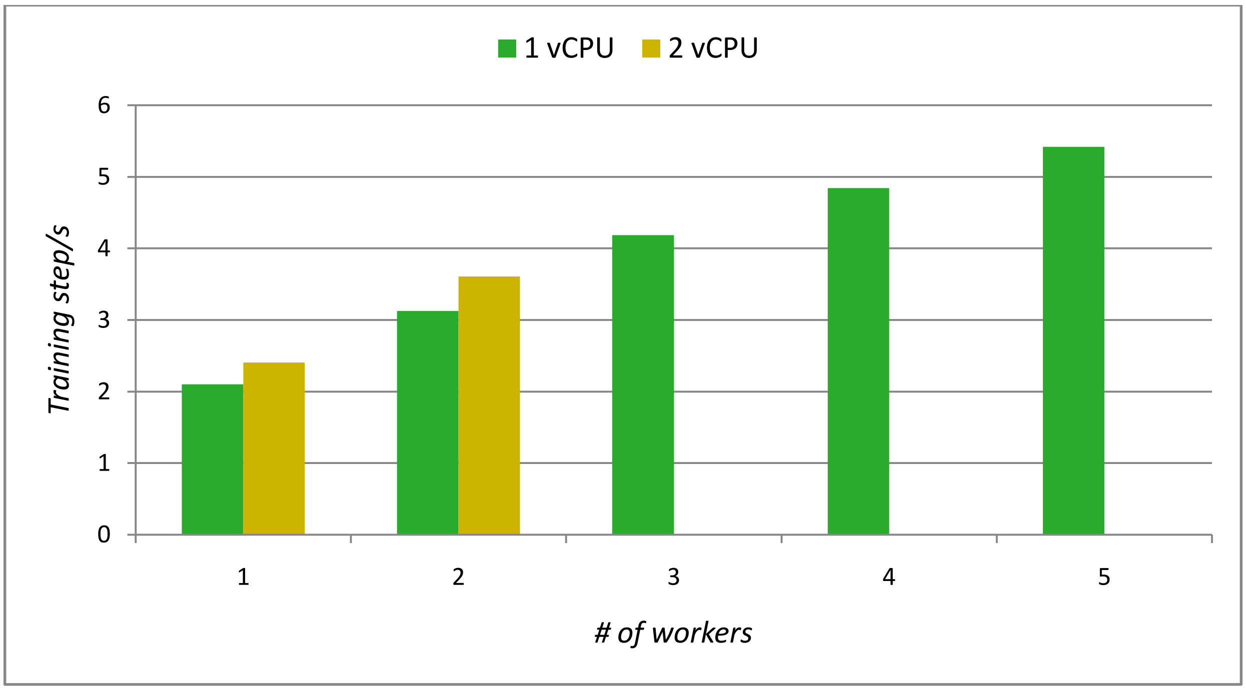
In the following plots I've reported both speedups, they appears quite overlapped, though the speedup using 2vCPU is slightly higher:



Both speedups are largely above the linearity, and, as expected, they increase slowly as the number of workers grows, meaning that the contribute of the network latency is strengthening.

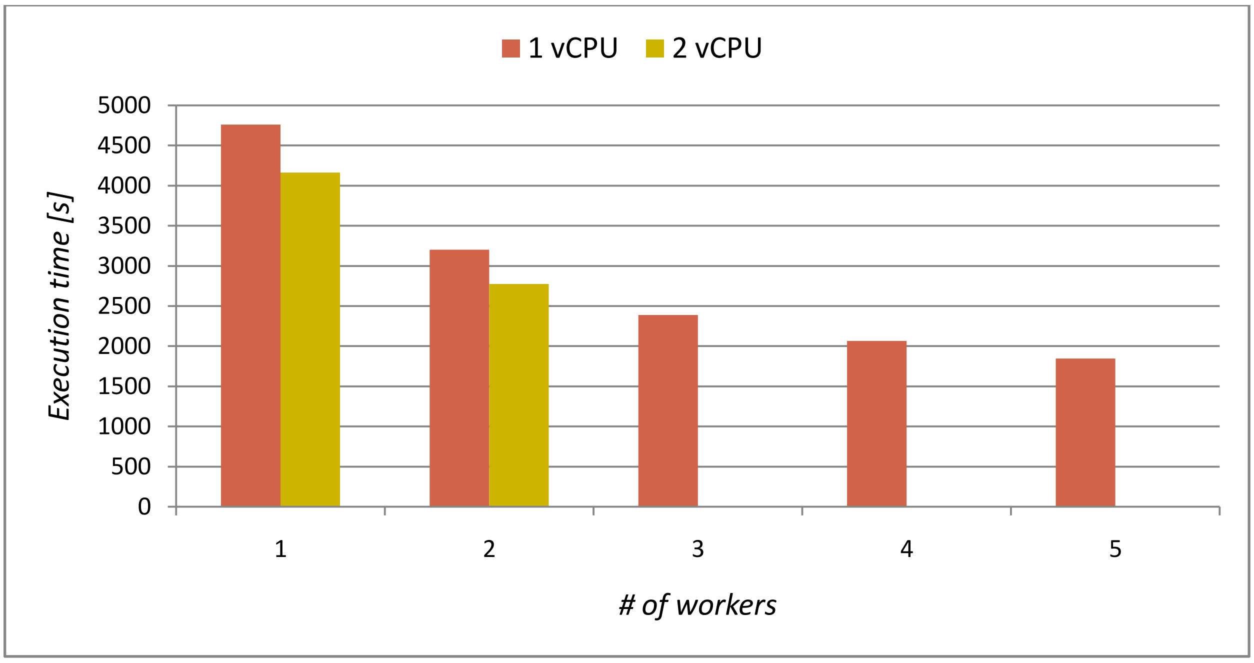
Training steps per second

As expected the 2 vCPUs instances works faster (higher is better) but cannot be said more because of the poor data I was able to get.



Execution time

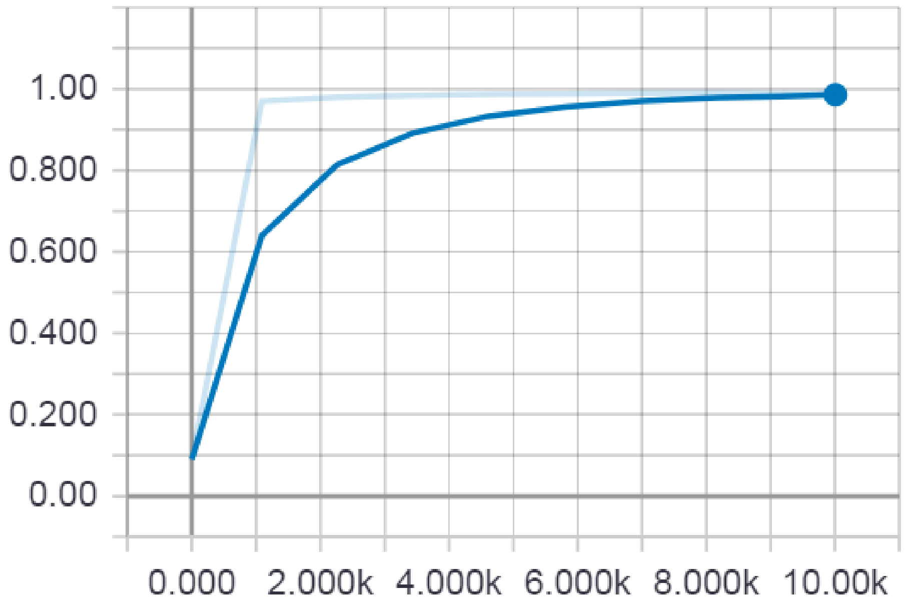
In this case lower is better, and, as above, the 2vCPU instances accomplishes the job in a lower time.



Accuracy

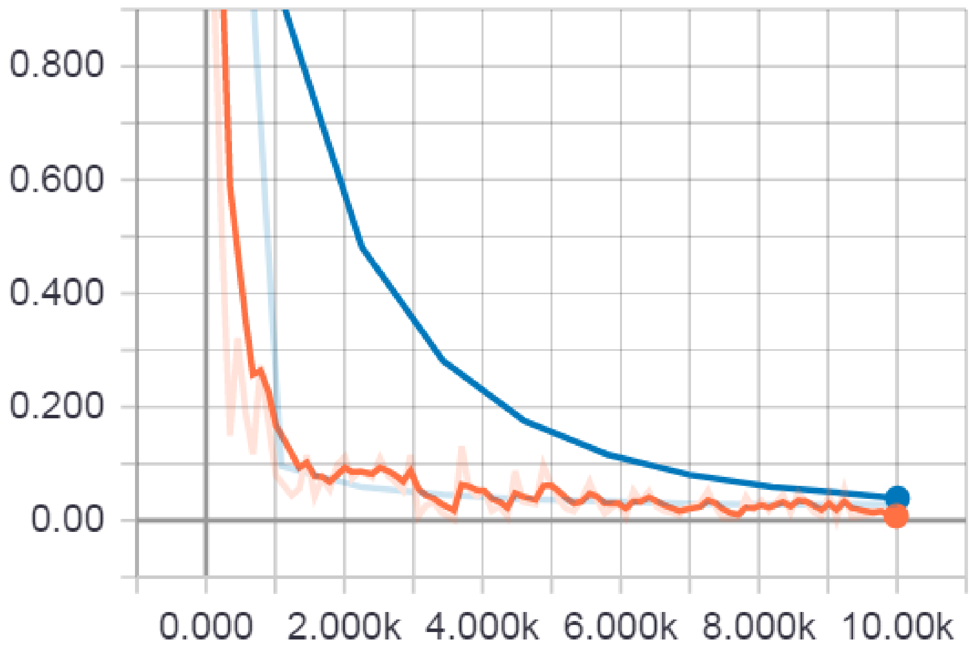
From TensorBoard I've got the following accuracy, where the light blue curve is the real accuracy at each step, while the strong blue it's the smoothed curve.

After 10 000 steps the estimated accuracy is about 0.99.



Cross Entropy

Again, the bold curves are the smoothed losses and the lighter are the real. The blue line represents the loss related to the test set, the orange one is the training loss.



6. Conclusion

In this work I got in the problem of scalability evaluation of Distributed TensorFlow. I've focused the evaluation on the training task of a CNN model, and I've addressed the scaling using single vCPU instances and dual vCPU instances.

In first analysis I've studied all the tools and implemented the architecture. Then, I've identified the key performance metrics to identify scalability, and finally, I've run the trials and got insights.

Some preliminary tests had shown how using 1 vCPU Master may lead to very worse performances. Instead using a less powerful PS is good, since it only needs to transmit and receive parameters from the workers.

The results from the experiments clearly shown that using CPUs doesn't lead to great performances in TensorFlow distributed environment, but still yields some sort of improvement.

The improvement due to increasing number of nodes is sub-linear, this may be due to the fact that distributed learning introduces communication overhead between computing nodes, which reduces unit effectiveness.

The analysis stopped here because the search space finished due to GCP free credits limitation. For a deeper understanding of the Distributed TensorFlow capabilities using GPUs I suggest the Benchmark section of the TensorFlow official website, there, they show quasi-linear speedups for several ML models training.

As final consideration, it should be said that Google is working a lot on the tooling for machine learning, both from software point of view (e.g. TensorFlow APIs) and from hardware as a service (i.e. Google Cloud). As an example, some days ago it was publicly released the availability of TPUs on GCP, which are optimized processing unit to improve speedup and efficiency for machine learning tasks.

7. References

- Distributed src: https://github.com/giuseppegagliano/cloudml-dist-mnist-example
- Non-distributed src: https://github.com/giuseppegagliano/tensorflow-mnist.git
- MNIST: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/learn/python/learn/datasets/mnist.py
- Official paper: http://web.eecs.umich.edu/~mosharaf/Readings/TensorFlow.pdf
- Official website: https://www.tensorflow.org/
- https://en.wikipedia.org/wiki/TensorFlow
- https://github.com/tensorflow
- http://www.i-programmer.info/news/105-artificial-intelligence/9158-tensorflow-googles-open-source-ai-and-computation-engine.html
- http://e-lin.github.io/wiki/jekyll/update/2017/02/20/First-Insight-of-TensorFlow-Architecture.html
- https://cloud.google.com/solutions/running-distributed-tensorflow-on-compute-engine
- https://www.tensorflow.org/api_docs/python/tf/contrib/learn/Experiment
- https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator
- http://yann.lecun.com/exdb/mnist/
- https://en.wikipedia.org/wiki/MNIST_database
- https://www.tensorflow.org/tutorials/layers
- https://cloud.google.com/vpc/docs/advanced-vpc
- https://www.altoros.com/performance-benchmark-distributed-tensorflow.html
- https://cloud.google.com/docs/overview/
- https://www.tensorflow.org/performance/benchmarks
- https://cloud.google.com/ml-engine/docs/distributed-tensorflow-mnist-cloud-datalab

8. Bibliography

- Hands-On Machine Learning with Scikit-Learn and TensorFlow - Concepts, Tools, and Techniques to Build Intelligent Systems By Aurélien Géron
- Big Data Analytics for Cloud Computing: A Cognitive Machine Learning Approach by Kai Hwang
- Introduction to parallel computing: design and analysis of algorithms By Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis