

Synchronous SGD

In this part we implemented the synchronous SGD algorithm based on two example codes provided. One example code shows how the input queues work in tensor flow environment. The second example code shows a simple synchronous update example.

We based our implementation on the second example code and added three main functions to it: `get_datapoint_iter`, `calc_gradient`, and `calc_precision`. These functions handle different aspects of our synchronous SGD from getting the training/testing dataset, pre-process them and finally use them in training procedure and testing

The function `get_datapoint_iter`, is responsible for creating and configuring the input queues. It gets a list of `tf` recordqueue file names and an integer `batch_size` parameter as input. Then it initialize input queues based on the filenames, cast the data read from output and finally use `shuffle_batch` method for generating batches of input. Each batch of data points is consist of a batch sparse tensor with dense shape of `[batch_size,num_features]`.

The function, `calc_gradient`, gets the training data parameters and labels as well as the model parameters as input. Then based on the given equation for calculating the gradient it calculates the gradient of the given batch. It must be noted that as the training data parameters are extremely sparse tensors, we used tensorflow's sparse arithmetic library to calculate sparse-dense multiplications required for gradients. Using sparse arithmetic library has advantages in both memory and cpu power used for computation. In order to profile our code to find bottlenecks in our code, we used tensorflow's profiling tool and profiled our codes runtime during on training cycle (figure 1). As shown in the profiling and the detailed result in table 1, most of the computation in this part is spent on last `sparse_tensor_dense_matmul` operator. This big runtime is directly caused by the large output of the operation which is a dense `[batch_size,num_features]` tensor where each row is the gradient related to a training datapoint. Although this large matrix is aggregated and reduced in next operator, it is still the most expensive part of calculating gradient computationally. This large matrix will not have direct effect on communication cost of algorithm due to the aggregation operator placed before transmission.

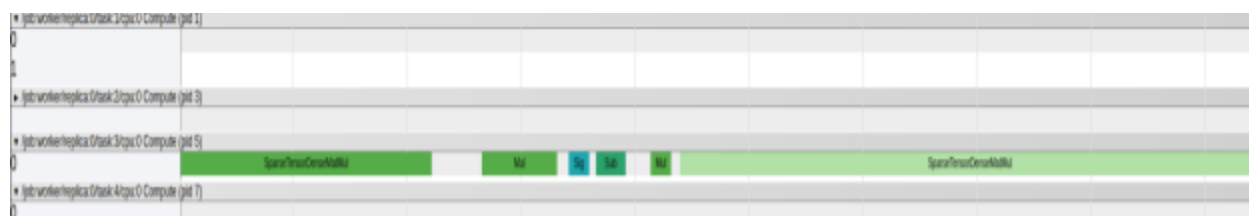


Fig1) Timing profile of one iteration of gradient calculation (batch_size = 10)

Table 1

Line of code	Output size	Duration (ms)
<code>pred = tf.sparse_tensor_dense_matmul(X, W)</code>	<code>[batchsize,1]</code>	2.02
<code>tf.mul(Y,pred)</code>	<code>[batchsize,1]</code>	0.03
<code>error = tf.sigmoid(_)</code>	<code>[batchsize,1]</code>	.009
<code>error_m1 = error-1</code>	<code>[batchsize,1]</code>	.013
<code>error_Y = tf.mul(Y,error_m1)</code>	<code>[batchsize,1]</code>	.009
<code>gradient = tf.sparse_tensor_dense_matmul(X_T,error_Y)</code>	<code>[batchsize,num_features]</code>	80.91
<code>return tf.reduce_sum(gradient,1)</code>	<code>[1,num_features]</code>	42.45

Finally the function, `calc_precision` is responsible for reading the testing datasets, calculate the prediction based on learned parameter and finally report the precision to console.

Asynchronous SGD

In this part we talk about our implementation of asynchronous SGD. Like the previous part we used three functions to handle different aspects of training. However in this part we based on design on the provided example Async code.

A brief summary of how ML on TF differs from implementing ML on Spark. Write what you think are the pros and cons of each framework.

Spark is general purpose framework that handles different type of workloads from SQL processing to variety of machine learning algorithms. On the other hand, tf is built for special kind of ML algorithms - deep learning algorithms.

Spark works on RDD abstraction which is by nature a distributed data type for handling fast and reliable computation across cluster of machines, whereas tf works on tensors which are single cluster node datatypes for storing model parameters only. So the single purpose and specialized design of tf datatypes makes them more suitable for communication across nodes and single node computations in training deep networks. However this single node design comes with a cost that multi-node computations are more complex and require a lot of tuning from programmer.

So overall dealing with distributed models in spark is easiest and a lot of complications are handled automatically with the framework, however the general purpose datatypes and distributed nature of them results in slower training and performance compared to the specialized tf model.

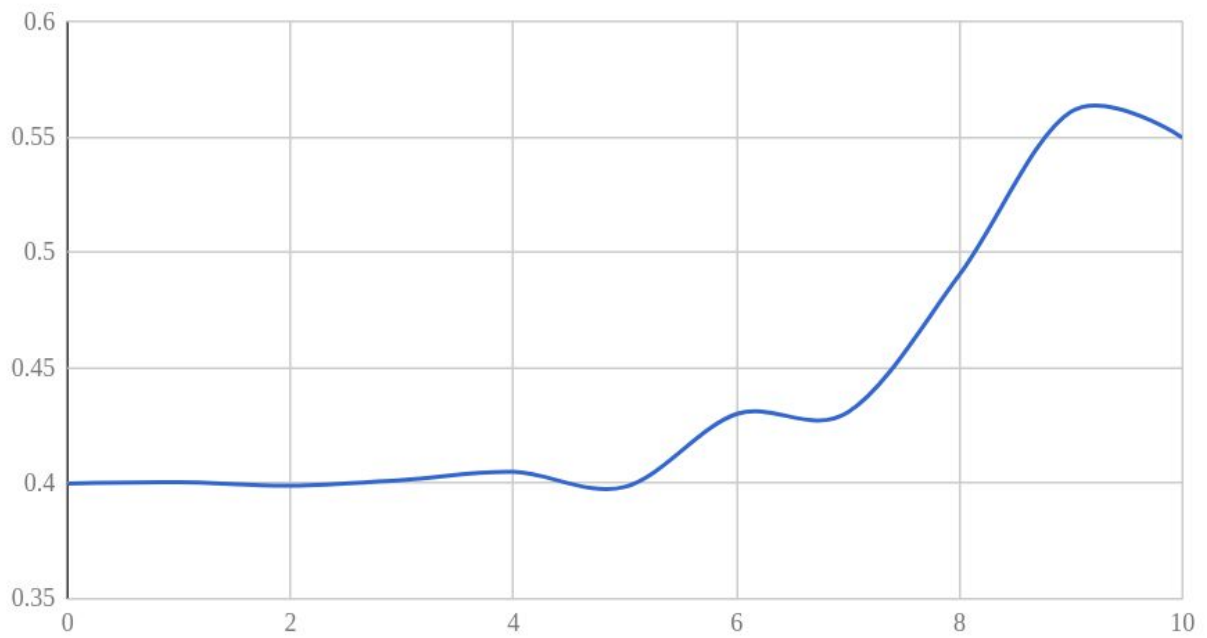
Graphs plotting the variation in test error (for every 100,000 iteration) for both Synchronous and Asynchronous-SGD

In order to get this precision graphs we run both algorithms to train on 200,000 points of dataset and calculated the precision after every 20,000 points training (5000 on each node). After tuning the algorithms (discussed in next section) we used training batch size of 100, for both types of implementations that resulted in a balance workload both in terms of memory, CPU and network. For the testing process and calculation the precision we used smaller sized batches as the computations were fast compared to training phase and we wanted to bound the memory used during testing. (each of the trainings take around 100 minutes)

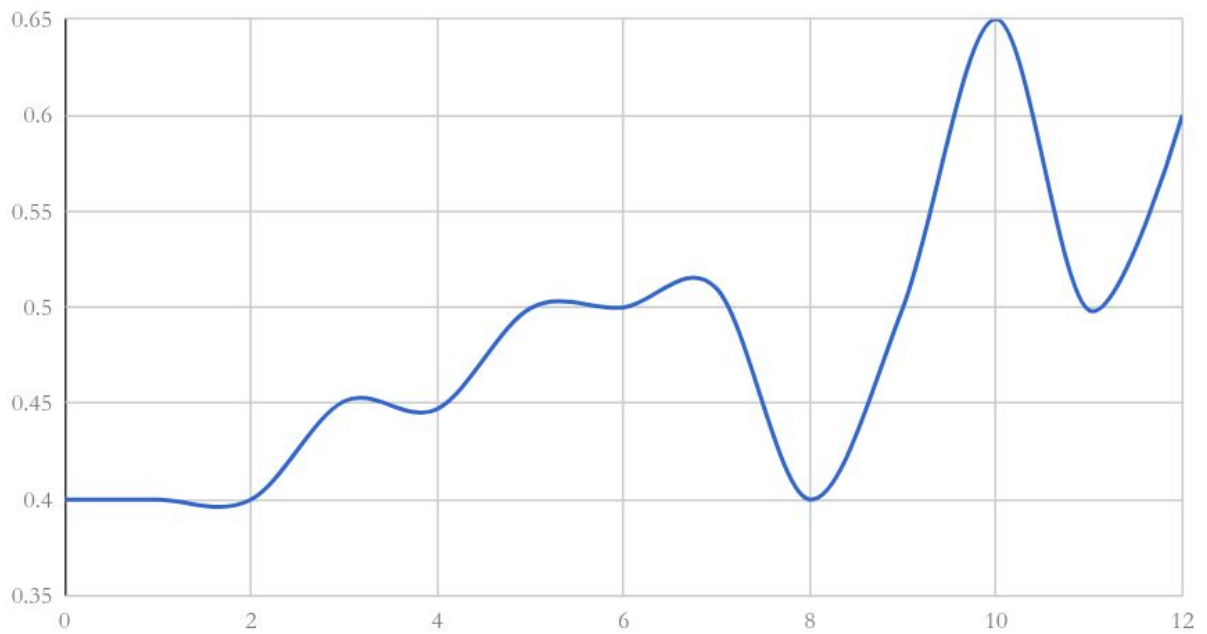
Table 2 below summarizes our hyperparameters during both implementations .

num_features	Number of parameter in model	33762578
s_batch	Training batch size	100
s_test	Testing batch size	20
train_test_ratio	Number of training iterations before each testing period	20000
total_trains	Total number of used training data points	200,000

Synchronous SGD training precision



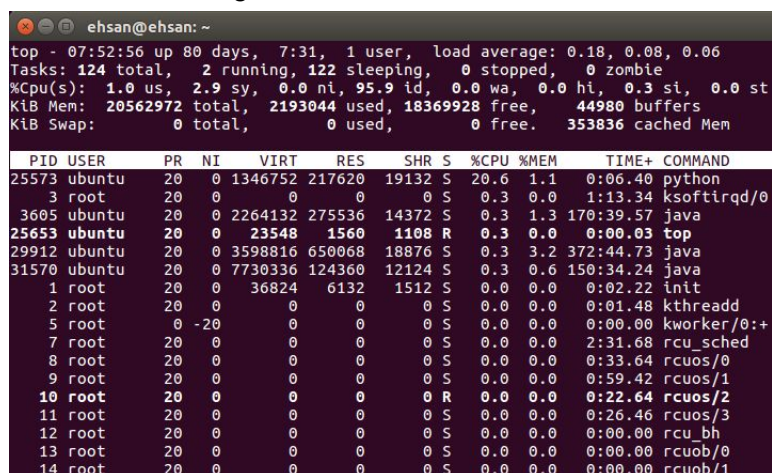
Asynchronous SGD Training Precision



Explain with supporting numbers what is the bottleneck resource for running synchronous and asynchronous-SGD. Is it the disk IO? CPU? Network?

In this section we used three different tools to measure resource usages in both implementations. In the following sections we first introduce our tools and then provide the discussion on measuring bottlenecks and the tuning process of algorithms accordingly.

To measure CPU usage and memory usage of our implementation we used 's "top" tool. Top (table of processes) is a task manager program found in many Unix-like operating systems. It produces an ordered list of running processes selected by user-specified criteria, and updates it periodically. Default ordering is by CPU usage, and only the top CPU consumers are shown. top shows how much processing power and memory are being used, as well as other information about the running processes. Some versions of top allow extensive customization of the display, such as choice of columns or sorting method.



```
ehsan@ehsan: ~
top - 07:52:56 up 80 days, 7:31, 1 user, load average: 0.18, 0.08, 0.06
Tasks: 124 total, 2 running, 122 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.0 us, 2.9 sy, 0.0 ni, 95.9 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem: 20562972 total, 2193044 used, 18369928 free, 44980 buffers
KiB Swap: 0 total, 0 used, 0 free. 353836 cached Mem

  PID USER      PR  NI   VIRT    RES    SHR  S  %CPU  %MEM   TIME+ COMMAND
25573 ubuntu    20   0 1346752 217620 19132  S   20.6   1.1   0:06.40 python
    3 root       20   0      0      0      0  S    0.3   0.0   1:13.34 ksoftirqd/0
3605 ubuntu    20   0 2264132 275536 14372  S    0.3   1.3 170:39.57 java
25653 ubuntu    20   0  23548   1560  1108  R    0.3   0.0   0:00.03 top
29912 ubuntu    20   0 3598816 650068 18876  S    0.3   3.2 372:44.73 java
31570 ubuntu    20   0 7730336 124360 12124  S    0.3   0.6 150:34.24 java
    1 root       20   0  36824   6132  1512  S    0.0   0.0   0:02.22 init
    2 root       20   0      0      0      0  S    0.0   0.0   0:01.48 kthreadd
    5 root       0 -20      0      0      0  S    0.0   0.0   0:00.00 kworker/0:
    7 root       20   0      0      0      0  S    0.0   0.0   2:31.68 rcu_sched
    8 root       20   0      0      0      0  S    0.0   0.0   0:33.64 rcuos/0
    9 root       20   0      0      0      0  S    0.0   0.0   0:59.42 rcuos/1
   10 root       20   0      0      0      0  R    0.0   0.0   0:22.64 rcuos/2
   11 root       20   0      0      0      0  S    0.0   0.0   0:26.46 rcuos/3
   12 root       20   0      0      0      0  S    0.0   0.0   0:00.00 rcu_bh
   13 root       20   0      0      0      0  S    0.0   0.0   0:00.00 rcuob/0
   14 root       20   0      0      0      0  S    0.0   0.0   0:00.00 rcuob/1
```

To measure network usage of our program we used "iftop" tool. iftop is a command-line system monitor tool that produces a frequently updated list of network connections. By default, the connections are ordered by bandwidth usage, with only the "top" bandwidth consumers shown. The iftop website gives the following description: "iftop does for network usage what top(1) does for CPU usage. It listens to network traffic on a named interface and displays a table of current bandwidth usage by pairs of hosts. Handy for answering the question 'why is our ADSL link so slow?'".

		191Mb	381Mb	572Mb	763Mb	954Mb
vm-32-3	=>	vm-32-1	265Mb	196Mb	141Mb	
	<=		5.26Mb	111Mb	156Mb	
vm-32-3	=>	vm-32-2	992b	4.97kb	4.71kb	
	<=		624b	2.91kb	2.85kb	
vm-32-3	=>	vm-32-5	992b	3.80kb	3.52kb	
	<=		576b	2.73kb	2.28kb	
vm-32-3	=>	vm-32-4	992b	3.80kb	3.49kb	
	<=		576b	1.62kb	1.51kb	
TX:		cum:	246MB	peak:	289Mb	rates:
RX:			272MB		534Mb	265Mb
TOTAL:			519MB		541Mb	196Mb
						141Mb
						5.26Mb
						111Mb
						156Mb
						270Mb
						307Mb
						296Mb

Finally to measure, disk io usages in our program we used “iotop”. iotop watches I/O usage information output by the Linux kernel and displays a table of current I/O usage by processes or threads on the system. iotop displays columns for the I/O bandwidth read and written by each process/thread during the sampling period. It also displays the percentage of time the thread/process spent while swapping in and while waiting on I/O. For each process, its I/O priority (class/level) is shown. In addition, the total I/O bandwidth read and written during the sampling period is displayed at the top of the interface.

ehsan@ehsan: ~							
Total DISK READ :				Total DISK WRITE :			
0.00 B/s				3.89 K/s			
Actual DISK READ:				Actual DISK WRITE:			
0.00 B/s				0.00 B/s			
TID	PRI	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
3708	be/4	ubuntu	0.00 B/s	3.89 K/s	0.00 %	0.00 %	java -ser-supervisor
1	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	init
2	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kthreadd]
3	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[ksoftirqd/0]
5	be/0	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kworker/0:0H]
7	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcu_sched]
8	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuos/0]
9	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuos/1]
10	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuos/2]
11	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuos/3]
12	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcu_bh]
13	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuob/0]
14	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuob/1]
15	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuob/2]
16	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcuob/3]
17	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[migration/0]
18	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[watchdog/0]
19	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[watchdog/1]
20	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[migration/1]
21	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[ksoftirqd/1]
23	be/0	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kworker/1:0H]

In order to detect bottlenecks of each implementation and tune the hyperparameters of our algorithm, we first used default batch_size of 20 and run our both implementations on our cluster. Table 3 summarizes the peak usage of each resource measured on vm32-3 during execution.

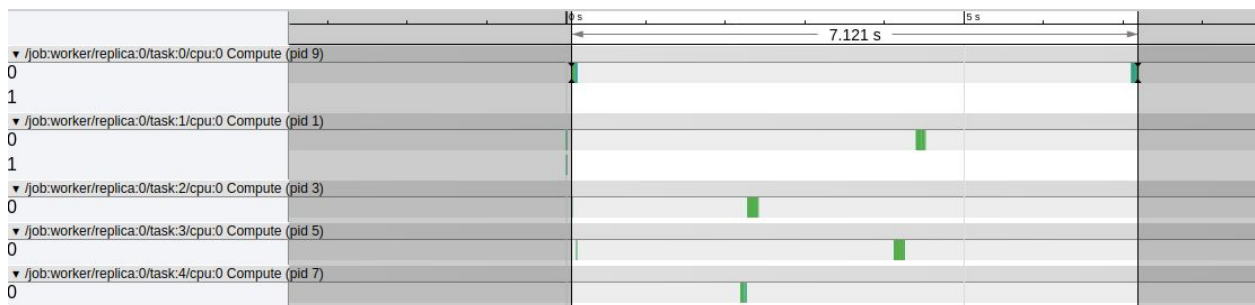
Table 3

	Synch	Asynch
Peak IO bandwidth	240 KBps	235 KBps
Peak CPU (%)	38%	96%
Peak Memory (%)	18%	46%
Peak Network bandwidth	349 MBps	395 MBps

Based on the above measurements we can have two main observations about our implementations. First in our synchronous SGD implementation, the io and memory bandwidths are manageable in this scale and not hindering our performance. on the other hand the amount of network usage is close to maximum available bandwidths. As a result this implementation is limited by our peak network's bandwidths as a result of excessive communication with parameter server. To support this measurements and conclusion we tested our hypotheses using tensorflow's profiling tool.

During our training we observed that each training round with batch size of 20 takes 7.121 seconds. Including communication, disk IO and computation. We used tensorflow's profiling tool to profile the computation on our parameter server node in the cluster (figure below).

As shown in the profiling output, only a small portion of this 7 second period is used for computation and parameter updates related to training model. The first row is the task responsible for updating the model parameters that is executed twice in this run and we measure the computation between them. The remainder of rows are responsible for computing the gradient of the related batch and computing the precision twice for this run. So based on the tensorflow's profiling memory and CPU cannot be candidates for bottleneck.



To proof that the network is indeed our limiting resource we made following conclusions. Our model has 33762578 parameters; if each be a floating point then overall we have 257 MB of data which needs to be transferred twice during each iteration of training. Considering the peak BW used by tensorflow to be 350MBps it should take around 7 seconds. This time is used to only transfer this data huge data twice in network which is close to what was the actual spent

time on each round of synchronous SGD. this results show the effect of bandwidth on performance of our synchronous SGD implementation and supports our earlier conclusion of having the network bandwidth as limiting resource.

To measure the limiting resource in the asynchronous SGD implementation we used the same procedure. Based on the results in table 3, CPU is the main bottleneck of algorithm as it is mostly saturated by the algorithm run.