

@Subaru

Enterprise IT & Software Technologies

□

POSTED BY

SUBARU KOKUBUN

POSTED ON

APRIL 10, 2018

POSTED UNDER

CLOUD, INTELLIGENCE

End-to-End Example for Distributed TensorFlow

Current computing technologies for AI (by giant Google, Microsoft, etc) is focusing more practical IT infrastructures or services, which enables to run high-throughput and massive workloads with cloud infrastructure and device-acceleration (GPU or TPU) integrated. In such a situation, the programming for distributed computing is an important piece of technologies.

In this blog I want to show several fundamentals and options for running distributed training with popular TensorFlow framework, and here (in this post) we begin the basic end-to-end example for your first start with programming, running, and evaluation.

There are several ideas to run distributed training with tensorflow (like Horovod, etc), but here I show the regular Distributed TensorFlow with standard `MonitoredTrainingSession`.

In the next post I'll show other (advanced) options based on this example.

Here I use only pure tensorflow library without Keras or other helper functions, and you can soon run this code on your computing environment in house or cloud infrastructures.

I don't go so far into topology or programming patterns for distributed computing ideas itself (I focus on the tutorial for your first understanding), but I hope this post helps you to start your distributed computing by popular tensorflow.

Preparing your multiple machines (cluster)

Before starting, you must prepare your multiple machines with python and TensorFlow installed and configured (GPU-accelerated, etc).

Here I don't explain how to setup your environment, but you can also use **Google Cloud Machine Learning Engine (Cloud ML)** or **Azure Batch AI**, which significantly simplifies your provisioning rather than the basic cloud infrastructure (IaaS) like Google Compute VM, Amazon EC2, or Azure VM. (Google Cloud ML is fully-managed and Azure Batch AI is IaaS based. For both Google Cloud ML and Azure Batch AI, you need to specify the following cluster spec and start the following `MonitoredTrainingSession` in your code.)

In this post I used Azure Batch AI for the following samples and [here](https://tsmatz.wordpress.com/2018/01/30/azure-batch-ai-how-it-works/) (<https://tsmatz.wordpress.com/2018/01/30/azure-batch-ai-how-it-works/>) is useful resource for your first start of Azure Batch AI. In the practical execution, you can also use infiniband network for inter-node communications.

Topology (Brief Overview)

There exist a lot of resources (articles) explaining the topology and programming of Distributed TensorFlow, but let me summarize brief concepts before starting. (You need to know for running your code.)

Distributed TensorFlow consists of 3 types of computing nodes, called "**parameter node**", "**worker node**", and "**master node**".

Computing session runs on multiple (or single) worker nodes. Computed parameters are kept by the parameter node and shared with workers. (You can also run multiple parameter nodes for each parameter blocks, which enables high-throughput IOs for writing and reading parameters. If you don't specify ps (=parameter server) tasks for variables, the round-robin strategy over all ps tasks is applied.) One of workers (among worker nodes) must be master node (chief) and master coordinates all workloads in each worker nodes.

Programming Sample

Because I want to focus only on our concerns, here I use simple graph (neural network) with well-known MNIST (hand-writing digits) and I referred [here](https://github.com/tsmatsuz/tensorflow-mnist-batch-read-and-train-tutorial/blob/master/mnist_tf.py) (https://github.com/tsmatsuz/tensorflow-mnist-batch-read-and-train-tutorial/blob/master/mnist_tf.py) (non-distributed training example) which only use standard tensorflow functions without any other helper classes or functions.

Now I modified this original code for Distributed TensorFlow and the following is our complete code for distributed training.

The highlighted line is modified for Distributed TensorFlow. Please compare the following distributed one with the original non-distributed one. (Here we use asynchronous training, with which each nodes has independent training loop.)

```
1  from __future__ import absolute_import
2  from __future__ import division
3  from __future__ import print_function
4
5  import sys
6  import argparse
7  import math
8
9  import tensorflow as tf
10
11  FLAGS = None
12  batch_size = 100
13
14  cluster = None
15  server = None
16  is_chief = False
17  num_workers = 2
18
19  class _MasterNodeHook(tf.train.SessionRunHook):
20      def begin(self):
21          # start without checkpoint
22          self._step = -1
23
24  def main(_):
25      with tf.device(tf.train.replica_device_setter(
26          worker_device="/job:%s/task:%d" % (FLAGS.job_name, FLAGS.task_index),
27          cluster=cluster)):
28
29          ###
30          ### Training
31          ###
32
33          #
34          # read training data
35          #
36
37          # image - 784 (=28 x 28) elements of grey-scaled integer value [0, 1]
38          # label - digit (0, 1, ..., 9)
39          train_queue = tf.train.string_input_producer(
40              [FLAGS.train_file],
41              num_epochs = 2) # data is repeated and it raises OutOfRange when data i
42          train_reader = tf.TFRecordReader()
43          _, train_serialized_exam = train_reader.read(train_queue)
44          train_exam = tf.parse_single_example(
45              train_serialized_exam,
46              features={
47                  'image_raw': tf.FixedLenFeature([], tf.string),
48                  'label': tf.FixedLenFeature([], tf.int64)
49              })
50          train_image = tf.decode_raw(train_exam['image_raw'], tf.uint8)
51          train_image.set_shape([784])
52          train_image = tf.cast(train_image, tf.float32) * (1. / 255)
53          train_label = tf.cast(train_exam['label'], tf.int32)
54          train_batch_image, train_batch_label = tf.train.batch(
55              [train_image, train_label],
56              batch_size=batch_size)
57
```

```

58 #
59 # define training graph
60 #
61
62 # define input
63 plchd_image = tf.placeholder(
64     dtype=tf.float32,
65     shape=(batch_size, 784))
66 plchd_label = tf.placeholder(
67     dtype=tf.int32,
68     shape=(batch_size))
69
70 # define network and inference
71 # (simple 2 fully connected hidden layer : 784->128->64->10)
72 with tf.name_scope('hidden1'):
73     weights = tf.Variable(
74         tf.truncated_normal(
75             [784, 128],
76             stddev=1.0 / math.sqrt(float(784))),
77         name='weights')
78     biases = tf.Variable(
79         tf.zeros([128]),
80         name='biases')
81     hidden1 = tf.nn.relu(tf.matmul(plchd_image, weights) + biases)
82 with tf.name_scope('hidden2'):
83     weights = tf.Variable(
84         tf.truncated_normal(
85             [128, 64],
86             stddev=1.0 / math.sqrt(float(128))),
87         name='weights')
88     biases = tf.Variable(
89         tf.zeros([64]),
90         name='biases')
91     hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)
92 with tf.name_scope('softmax_linear'):
93     weights = tf.Variable(
94         tf.truncated_normal(
95             [64, 10],
96             stddev=1.0 / math.sqrt(float(64))),
97         name='weights')
98     biases = tf.Variable(
99         tf.zeros([10]),
100        name='biases')
101     logits = tf.matmul(hidden2, weights) + biases
102
103 # define optimization
104 global_step = tf.train.create_global_step() # start without checkpoint
105 optimizer = tf.train.GradientDescentOptimizer(
106     learning_rate=0.07)
107 loss = tf.losses.sparse_softmax_cross_entropy(
108     labels=plchd_label,
109     logits=logits)
110 train_op = optimizer.minimize(
111     loss=loss,
112     global_step=global_step)
113
114 #

```

```

115     # run session
116     #
117
118     if is_chief:
119         hooks = [_MasterNodeHook()]
120     else:
121         hooks = []
122     with tf.train.MonitoredTrainingSession(
123         master=server.target,
124         checkpoint_dir=FLAGS.out_dir,
125         hooks=hooks,
126         is_chief=is_chief) as sess:
127
128         # when data is over, OutOfRangeError occurs and ends with MonitoredSess
129
130         step = 0
131         array_image, array_label = sess.run(
132             [train_batch_image, train_batch_label])
133         while not sess.should_stop():
134             feed_dict = {
135                 plchd_image: array_image,
136                 plchd_label: array_label
137             }
138             _, loss_value, array_image, array_label = sess.run(
139                 [train_op, loss, train_batch_image, train_batch_label],
140                 feed_dict=feed_dict)
141             step += 1
142             if step % 100 == 0:
143                 print("Worker: Step %d (Loss: %.2f)" % (step, loss_value))
144
145             print('training finished')
146
147 if __name__ == '__main__':
148     parser = argparse.ArgumentParser()
149     parser.add_argument(
150         '--train_file',
151         type=str,
152         default='/home/demouser/train.tfrecords',
153         help='File path for the training data.')
154     parser.add_argument(
155         '--out_dir',
156         type=str,
157         default='/home/demouser/out',
158         help='Dir path for the model and checkpoint output.')
159     parser.add_argument(
160         '--job_name',
161         type=str,
162         required=True,
163         help='job name (parameter or worker) for cluster')
164     parser.add_argument(
165         '--task_index',
166         type=int,
167         required=True,
168         help='index number in job for cluster')
169     FLAGS, unparsed = parser.parse_known_args()
170
171     # start server

```

```

172 cluster = tf.train.ClusterSpec({
173     'ps': ['10.0.0.6:2222'],
174     'worker': [
175         '10.0.0.4:2222',
176         '10.0.0.5:2222'
177     ]})
178 server = tf.train.Server(
179     cluster,
180     job_name=FLAGS.job_name,
181     task_index=FLAGS.task_index)
182 if FLAGS.job_name == "ps":
183     server.join()
184 elif FLAGS.job_name == "worker":
185     is_chief = (FLAGS.task_index == 0)
186     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

Now I pick up and explain about our Distributed TensorFlow example.

First, we start `tf.train.Server` in each nodes to communicate each other with gRPC protocol. As you can see below, each server is having corresponding each role (later we set `job_name` and `task_index` using command line options), and here I assume one parameter node (10.0.0.6) and two worker nodes (10.0.0.4 and 10.0.0.5), in which 10.0.0.4 is master (chief) node.

```

1  def main(_):
2      with tf.device(tf.train.replica_device_setter(
3          worker_device="/job:%s/task:%d" % (FLAGS.job_name, FLAGS.task_index),
4          cluster=cluster)):
5          ...
6
7  if __name__ == '__main__':
8      ...
9
10 # start server
11 cluster = tf.train.ClusterSpec({
12     'ps': ['10.0.0.6:2222'],
13     'worker': [
14         '10.0.0.4:2222',
15         '10.0.0.5:2222'
16     ]})
17 server = tf.train.Server(
18     cluster,
19     job_name=FLAGS.job_name,
20     task_index=FLAGS.task_index)
21 if FLAGS.job_name == "ps":
22     server.join()
23 elif FLAGS.job_name == "worker":
24     is_chief = (FLAGS.task_index == 0)
25     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

In this example, we're assigning each one task (role) for each node (computing machine), but you can also assign one task for each devices (CPUs or GPUs) on the single machine as follows.

```

1  ...
2  worker_device = "/job:%s/task:%d/gpu:%d" % (FLAGS.job_name, FLAGS.task_index, 6
3  with tf.device(tf.train.replica_device_setter(
4      worker_device=worker_device,
5      ps_device="/job:ps/cpu:0",
6      cluster=cluster)):
7  ...

```

In this example, we set 2 as `num_epochs` for reading data (`train.tfrecords`). Therefore, after data is read (iterated) twice by cycle, `OutOfRangeError` occurs and session (`MonitoredTrainingSession`) is correctly closed.

```

1  ...
2  train_queue = tf.train.string_input_producer(
3      [FLAGS.train_file],
4      num_epochs = 2)
5  ...

```

After the graph is constructed, we run the distributed session using `tf.train.MonitoredTrainingSession`. This session monitors each tasks in each nodes and it's the core component for distributed computation. (You can also use `tf.train.Supervisor` for distributed running, but `tf.train.MonitoredTrainingSession` is recommended for the current version.) In the session we read `train_batch_image` and `train_batch_label` (each 100 rows of features and labels) sequentially and run `train_op` and estimate loss .

I note that the [original sample \(https://github.com/tsmatsuz/tensorflow-mnist-batch-read-and-train-tutorial/blob/master/mnist_tf.py\)](https://github.com/tsmatsuz/tensorflow-mnist-batch-read-and-train-tutorial/blob/master/mnist_tf.py) (non-distributed one) manually started queue runner (`QueueRunner`) for batch-reading. But in our example, `tf.train.MonitoredSession` launches queue thread in the background, and we don't need to start queue runner explicitly.

```

1  ...
2  with tf.train.MonitoredTrainingSession(
3      master=server.target,
4      checkpoint_dir=FLAGS.out_dir,
5      hooks=hooks,
6      is_chief=is_chief) as sess:
7
8      step = 0
9      array_image, array_label = sess.run(
10         [train_batch_image, train_batch_label])
11     while not sess.should_stop():
12         feed_dict = {
13             plchd_image: array_image,
14             plchd_label: array_label
15         }
16         _, loss_value, array_image, array_label = sess.run(
17             [train_op, loss, train_batch_image, train_batch_label],
18             feed_dict=feed_dict)
19         step += 1
20         if step % 100 == 0:
21             print("Worker: Step %d (Loss: %.2f)" % (step, loss_value))
22     ...

```

In asynchronous approach, you can also implement to detect the completion of each nodes automatically with enqueue/ dequeue mechanism and you can proceed the consequent steps as follows.

Run and Check Our Model

Now let's start our program with the following command.

As I mentioned earlier, here we're assuming one parameter server (10.0.0.6) and two worker servers (10.0.0.4 and 10.0.0.5).

Parameter Node (10.0.0.6)

```
1 | python mnist_tf_dist.py \
2 |   --job_name=ps \
3 |   --task_index=0
```

Worker Node 1 (10.0.0.4)

```
1 | python mnist_tf_dist.py \
2 |   --job_name=worker \
3 |   --task_index=0
```

Worker Node 2 (10.0.0.5)

```
1 | python mnist_tf_dist.py \
2 |   --job_name=worker \
3 |   --task_index=1
```

Once the parameter server is started and all workers enter in `MonitoredTrainingSession`, the training (the loop of `train_op`) starts in multiple workers and each nodes respectively (independently) read data on corresponding node.

If you use the shared NFS storage or cloud storage, all nodes can read the same data respectively. (You don't need to deploy data in multiple nodes respectively.)

Output (2 workers – 10.0.0.4 and 10.0.0.5)

```
ibuted_runtime/zpc/grpc_channel.cc:215] Initialize
GrpcChannelCache for job ps -> {0 -> 10.0.0.6:2222}
2018-04-13 01:31:36.515201: I tensorflow/core/distr
ibuted_runtime/zpc/grpc_channel.cc:215] Initialize
GrpcChannelCache for job worker -> {0 -> localhost:
2222, 1 -> 10.0.0.5:2222}
2018-04-13 01:31:36.516759: I tensorflow/core/distr
ibuted_runtime/zpc/grpc_server_lib.cc:324] Started
server with target: grpc://localhost:2222
2018-04-13 01:31:36.954535: I tensorflow/core/distr
ibuted_runtime/master_session.cc:1024] Start master
session 2b4a4a8bfcaf2c47 with config:
Worker: Step 100 (Loss: 0.44)
Worker: Step 200 (Loss: 0.41)
Worker: Step 300 (Loss: 0.22)
Worker: Step 400 (Loss: 0.20)
Worker: Step 500 (Loss: 0.33)
Worker: Step 600 (Loss: 0.13)
Worker: Step 700 (Loss: 0.20)
Worker: Step 800 (Loss: 0.32)
Worker: Step 900 (Loss: 0.21)
Worker: Step 1000 (Loss: 0.11)
training finished
10.0.0.4 ~]$

ibuted_runtime/zpc/grpc_channel.cc:215] Initialize
GrpcChannelCache for job ps -> {0 -> 10.0.0.6:2222}
2018-04-13 01:31:35.019988: I tensorflow/core/distr
ibuted_runtime/zpc/grpc_channel.cc:215] Initialize
GrpcChannelCache for job worker -> {0 -> 10.0.0.4:2
222, 1 -> localhost:2222}
2018-04-13 01:31:35.021338: I tensorflow/core/distr
ibuted_runtime/zpc/grpc_server_lib.cc:324] Started
server with target: grpc://localhost:2222
2018-04-13 01:31:37.489268: I tensorflow/core/distr
ibuted_runtime/master_session.cc:1024] Start master
session 04ad6804917a49de with config:
Worker: Step 100 (Loss: 0.38)
Worker: Step 200 (Loss: 0.41)
Worker: Step 300 (Loss: 0.25)
Worker: Step 400 (Loss: 0.28)
Worker: Step 500 (Loss: 0.41)
Worker: Step 600 (Loss: 0.16)
Worker: Step 700 (Loss: 0.17)
Worker: Step 800 (Loss: 0.35)
Worker: Step 900 (Loss: 0.23)
Worker: Step 1000 (Loss: 0.13)
training finished
[10.0.0.5 ~]$
```

(https://netweblog.files.wordpress.com/2018/04/run_output1.jpg).

As you can see in our source code, we're setting `checkpoint_dir` in session. By this setting, the checkpoint data (tensor objects, variables, etc) are all saved in the directory and you can restore and restart your training later again. You can use gathered checkpoint data as you need. (retrain, test, etc) For example, when you copy these checkpoint data in the specific local folder, you can test the model's accuracy using the following local (non-distributed) python code. (Note that the meta is saved in master node, but the index and data is saved in parameter node separately. You must gather these data in one location for running the following code.)

```
1  from __future__ import absolute_import
2  from __future__ import division
3  from __future__ import print_function
4
5  import sys
6  import argparse
7  import math
8
9  import tensorflow as tf
10
11  FLAGS = None
12  batch_size = 100
13
14  def main(_):
15      #
16      # define graph (to be restored !)
17      #
18
19      # define input
20      plchd_image = tf.placeholder(
21          dtype=tf.float32,
22          shape=(batch_size, 784))
23      plchd_label = tf.placeholder(
24          dtype=tf.int32,
25          shape=(batch_size))
26
27      # define network and inference
28      with tf.name_scope('hidden1'):
29          weights = tf.Variable(
30              tf.truncated_normal(
31                  [784, 128],
32                  stddev=1.0 / math.sqrt(float(784))),
33              name='weights')
34          biases = tf.Variable(
35              tf.zeros([128]),
36              name='biases')
37          hidden1 = tf.nn.relu(tf.matmul(plchd_image, weights) + biases)
38      with tf.name_scope('hidden2'):
39          weights = tf.Variable(
40              tf.truncated_normal(
41                  [128, 64],
42                  stddev=1.0 / math.sqrt(float(128))),
43              name='weights')
44          biases = tf.Variable(
45              tf.zeros([64]),
46              name='biases')
47          hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)
48      with tf.name_scope('softmax_linear'):
```

```

49 weights = tf.Variable(
50     tf.truncated_normal(
51         [64, 10],
52         stddev=1.0 / math.sqrt(float(64))),
53     name='weights')
54 biases = tf.Variable(
55     tf.zeros([10]),
56     name='biases')
57 logits = tf.matmul(hidden2, weights) + biases
58
59 #
60 # Restore and Testing
61 #
62
63 ckpt = tf.train.get_checkpoint_state(FLAGS.out_dir)
64 idex = int(ckpt.model_checkpoint_path.split('/')[0].split('-')[-1])
65
66 saver = tf.train.Saver()
67
68 with tf.Session() as sess:
69     # restore graph
70     saver.restore(sess, ckpt.model_checkpoint_path)
71     graph = tf.get_default_graph()
72
73     # add to graph - read test data
74     test_queue = tf.train.string_input_producer(
75         [FLAGS.test_file],
76         num_epochs = 1) # when data is over, it raises OutOfRange
77     test_reader = tf.TFRecordReader()
78     _, test_serialized_exam = test_reader.read(test_queue)
79     test_exam = tf.parse_single_example(
80         test_serialized_exam,
81         features={
82             'image_raw': tf.FixedLenFeature([], tf.string),
83             'label': tf.FixedLenFeature([], tf.int64)
84         })
85     test_image = tf.decode_raw(test_exam['image_raw'], tf.uint8)
86     test_image.set_shape([784])
87     test_image = tf.cast(test_image, tf.float32) * (1. / 255)
88     test_label = tf.cast(test_exam['label'], tf.int32)
89     test_batch_image, test_batch_label = tf.train.batch(
90         [test_image, test_label],
91         batch_size=batch_size)
92
93     # add to graph - test (evaluate) graph
94     array_correct = tf.nn.in_top_k(logits, plchd_label, 1)
95     test_op = tf.reduce_sum(tf.cast(array_correct, tf.int32))
96
97     # run
98     sess.run(tf.initialize_local_variables())
99     coord = tf.train.Coordinator()
100     threads = tf.train.start_queue_runners(sess=sess, coord=coord) # for data
101     num_test = 0
102     num_true = 0
103     array_image, array_label = sess.run(
104         [test_batch_image, test_batch_label])
105     try:

```

```
106     while True:
107         feed_dict = {
108             plchd_image: array_image,
109             plchd_label: array_label
110         }
111         batch_num_true, array_image, array_label = sess.run(
112             [test_op, test_batch_image, test_batch_label],
113             feed_dict=feed_dict)
114         num_true += batch_num_true
115         num_test += batch_size
116     except tf.errors.OutOfRangeError:
117         print('Scoring done !')
118         precision = float(num_true) / num_test
119         print('Accuracy: %0.04f (Num of samples: %d)' %
120             (precision, num_test))
121
122 if __name__ == '__main__':
123     parser = argparse.ArgumentParser()
124     parser.add_argument(
125         '--test_file',
126         type=str,
127         default='/home/demouser/test.tfrecords',
128         help='File path for the test data.')
129     parser.add_argument(
130         '--out_dir',
131         type=str,
132         default='/home/demouser/out',
133         help='Dir path for the model and checkpoint output.')
134     FLAGS, unparsed = parser.parse_known_args()
135
136     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

Next I'll show Distributed TensorFlow using Estimator or Experiment. (Recent samples are using these high-level APIs.)