Tensorflow Integration with Twister2

Dimuthu Wannipurage

Introduction

Twister2 is a distributed big data framework that supports data streaming, pipelines and analytics that runs on various execution platforms like MPI, Nomad and Kubernetes. With the rapid evolve of deep neural network frameworks like Tensorflow and Pytorch and highly accessible GPU clusters all around the world, running deep neural networks in large distributed environments has been a popular topic these days. In this project I'm trying to evaluate the feasibility of the integration of the Tensorflow Deep Neural Network framework with Twister2 data pipelines and capture the performance numbers running Tensorflow with Twsiter2 both in single node and distributed modes.

API Evaluations

Twister2 API

For this project I used the python bindings of Twister2 as it's easy to integrate with Tensorflow with its python API

Twister2 Data Models

I used the TSet Source [1] to generate and feed the data to Tensorflow models. When retrieving data from a TSet Source I used the both approaches of streaming and caching provided by the framework.

Tensorflow Data Models

Tensorflow has its own way of loading data into a model called TF Data API. Using this API, we can load data either through static files or dynamically calling external data endpoints. Further,

Tensorflow implements an approach called data generators to load and convert external data endpoints into the TF Data models.

Methodology

Main challenge in this integration work is to efficiently transform the data models that are emitting from Twister2 into a compatible data type of Tensorflow. I decided to go with the Data Generator approach mentioned in the previous section as it's convenient to integrate with external data sources like TSet Source. Pseudo code of how generators are integrating with Tensorflow models are as below

```
def generator(external_data):  # External data is the input data
array
     for data in external_data:
          yield(data.x, data.y)

external_data = fetch_from_twister2()
dataset = tf.data.Dataset.from_generator(lambda:
generator(external_data), (tf.int32, tf.int32), (x_size, y_size))

tf_mode.fit(data_set) # Train the model
```

So if we can find a way to map the output coming from Twister2 into an iterable object, then we can feed it into the data generator to convert into TF Data objects.

Streaming training vs cached training

Twister2 supports both streaming and cached data loading from sources. Both have some downsides. In streaming, you have to wait until the next data element is available and it might be the bottleneck for real time training. In caching, we might run out of memory / disk space of the worker node depending on the size of the data. However we need the support of both methods for various use cases. For example in reinforcement learning, you might need a stream of events happening in the real time to train the model on demand. In classic classification and regression training scenarios, you need the data to be available / saved as you are reusing the same data over many number of training iterations.

Twister2 in streaming mode

When we use the streaming mode, Twister2 accepts a callback function to be invoked at the appearance of each data point.

```
a = source_x.compute(compute_func)
a.for_each(lambda i : i)
```

Because this compute_func is invoked asynchronously, it's challenging to create the data flowing path from the data arriving into the compute_func to the data generator of the Tensorflow. So the solution I came up with is that, instead of creating a programmatic data flow, bridge it with an unix socket. Doing so, you can fully decouple the Twister2 process with Tensorflow process even though you see that they are running in a same process. Following figure demonstrates the data path.

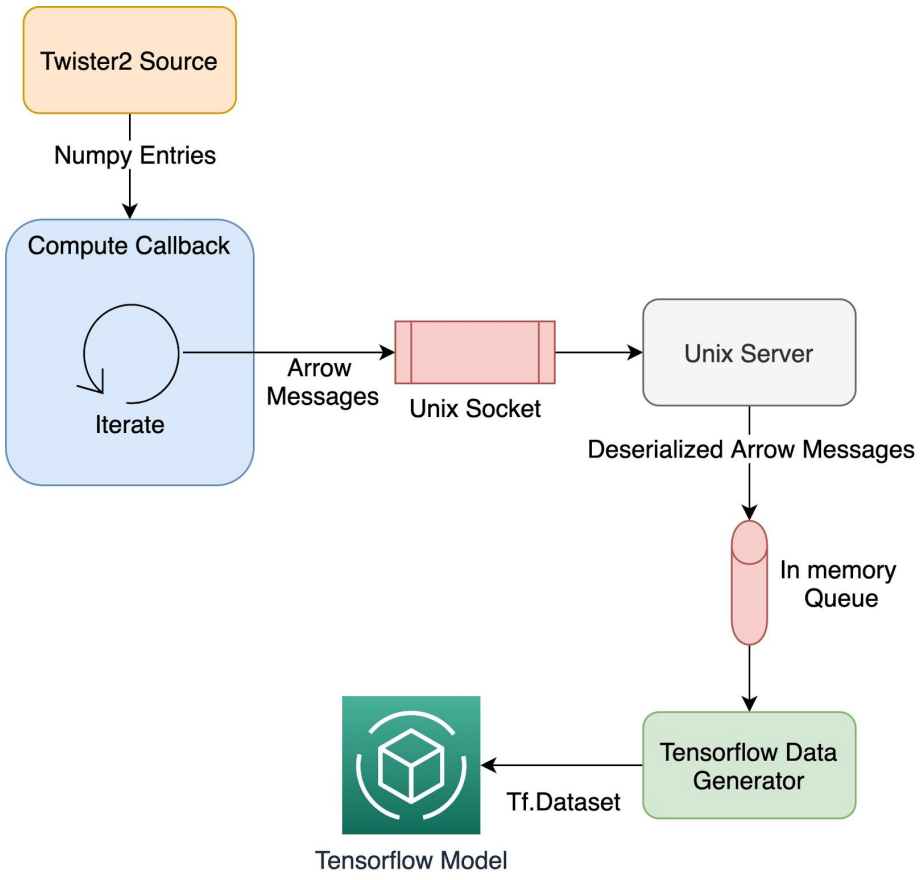

Figure 1: Dataflow from a streaming source to Tensorflow

Detailed message path is as below.

1. Source generates MNIST image data and sends (x,y) entries to compute callback
2. Compute function iterates over the dataset and put each entry into an external unix socket endpoint after converting data into Arrow format
3. Unix server keeps listening to the messages coming into the socket and once a message is read, deserialize the Arrow format and convert it into pandas format.
4. Converted data is pushed into an in-memory, thread safe blocking queue
5. There is a Data Generator waiting for the messages coming into the queue and provides on demand and batched training input to the Tensorflow model

Source code for the above solution is provided in the appendix section.

Twister2 in cached mode

This is a straightforward approach for pipelining the Twister2 data sources into the Tensorflow data generators. We can use `cache_train = source_x.cache()` method to prefetch all the data into the memory before starting the Tensorflow training. Once the prefetch is done, it can be directly sent to the data generator and then start the training. One restriction that poses in this approach is that Tensorflow is running inside the same process as Twister2 is running. This method is very useful in the cases where you have multiple epochs of training with data reuse.

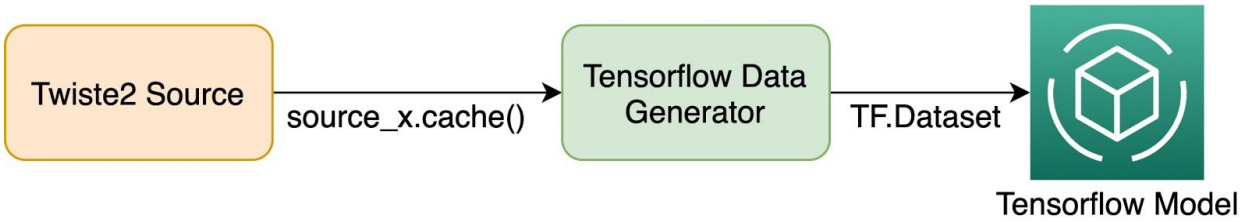

Figure 2: Dataflow from a cached source to Tensorflow

Tensorflow in distributed mode

Both of the above approaches were described assuming Tensorflow is running in a single node non clustered environment. But the actual advantage is takens when Twister2 is running in multiple nodes with Tensorflow configured to run in distributed mode. Tensorflow already has it's own data parallel distributed mode [2]. So we need to figure out a way to auto generate these configurations when we run Tensorflow inside Twister2 in either of the above methods. Luckily, code changes that are required to translate a single node Tensorflow model to a distributed model is very small. You only have to add below lines to do that

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
with strategy.scope():
      ## Your model training code
```

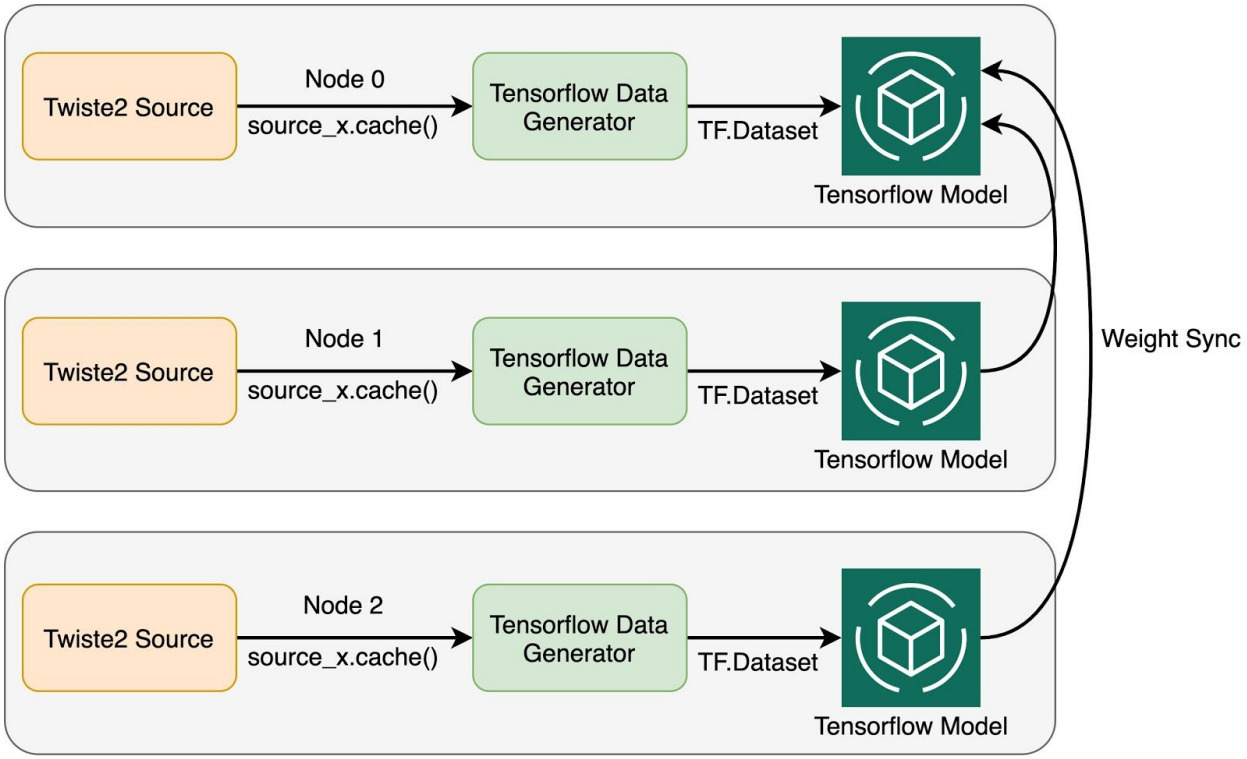

Figure 3: Dataflow of distributed Tensorflow with Twister2 sources

In addition to that, Tensorflow needs a cluster configuration json exported in TF_CONFIG environment variable to configure the cluster. Sample configuration json is as below. In the worker section, you should have all the worker IPs that are going to include into the cluster. In the index field, you should have the id of the current worker. It should be one in range 0 - (numberor workers - 1). Worker with id = 0 becomes the master of the cluster by default.

```
{
   'cluster': {
       'worker': [worker1_ip:port, worker_2_ip:port, ....]
   },
   'task': {'type': 'worker', 'index': current_worker_id}
}
```

However, if we need Tensorflow distributed to run in Twister2 seamlessly, we need to generate this configuration for each worker. To generate the worker section, we can use the `twister2_environemnt.peers()` method to get peer information. Using

`twister2_environemnt.worker_id`, we can derive the index. Sample code to generate the cluster configuration is as below

```
def configure_tf_dist_env(tw2env, base_port = 12345):
      """ Configures Tensorflow distributed environment. This genretates Tensorflow specific
cluster configs """
    peer_map = tw2env.peers()
    peer_summary = Counter(peer_map.values()) # Handles workers schedule in same node
    tf_worker_connections = []

    for i in range(len(peer_map)):
        worker_host = peer_map[i]
        peer_summary[worker_host] = peer_summary[worker_host] - 1
        # If there are more than 1 worker in same node, increment base port to avoid conflicts
                    tf_worker_connections.append(worker_host   +   ":"   +   str(base_port   +
peer_summary[worker_host]))

    print("Tensorflow Worker Connections " + str(tf_worker_connections))

    os.environ['TF_CONFIG'] = json.dumps({
        'cluster': {
            'worker': tf_worker_connections
        },
        'task': {'type': 'worker', 'index': tw2env.worker_id}
    })
```

Full Source code for the above solution is provided in the appendix section.

Results

I performed preliminary tests to make sure that data parallel training on Tensorflow with Twister2 actually reduces the training time and the data loading time. Tests were performed on the Tensorflow distributed mode with Twister2 source caching. Training task was to train a network in distributed mode to predict digits on the MNIST data set and to reduce the running time, only first 10000 images and labels of the MNIST data set were selected. I used 7 machines in Juliet cluster with the CPU backend of Tensorflow and Twister2 with MPI backend for all the tests. Training model contained 3 fully connected layers with respective 768, 64 and 10 neurons at each layer with last layer softmax activation. When training, Twister2 TSet source divided the dataset to the number of nodes evenly and at each the end of each epoch, Tensorflow master is updated by the rest of the workers with weight updates. I measured the average training time, accuracy of the final model and the average time to load data into each worker by running the test from 1 to 7 nodes for 5 epochs. Figure 4 shows the average time in seconds taken to train the model with different parallelism levels. Figure 5 shows the average accuracy over parallelism and Figure 6 shows the average time taken to load the data.

Based on the observations, I have seen that the time to train and time to load the data have reduced when the parallelism increased. It was a steep curve until the parallelism = 3 and after that, the rate of drop was decreased. In all cases, accuracy of the model remained consistent except the sudden spike in parallelism = 3. As a summery, I could see that distributed training on Tensorflow with Twister2 gave expected result of reducing the model training time and data loading time as the data is splitted across the amount of workers of the cluster

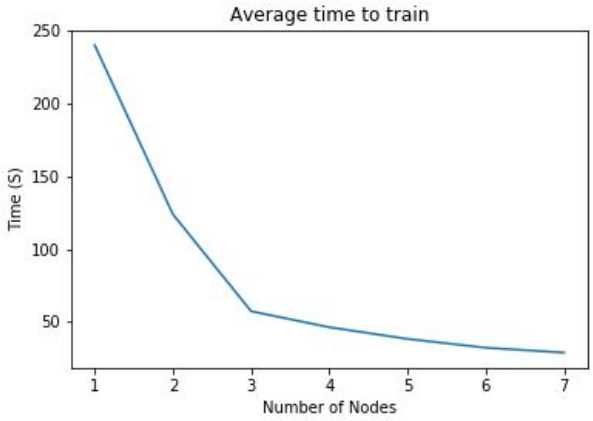

Figure 4: Average time taken to train the model for different parallelisms

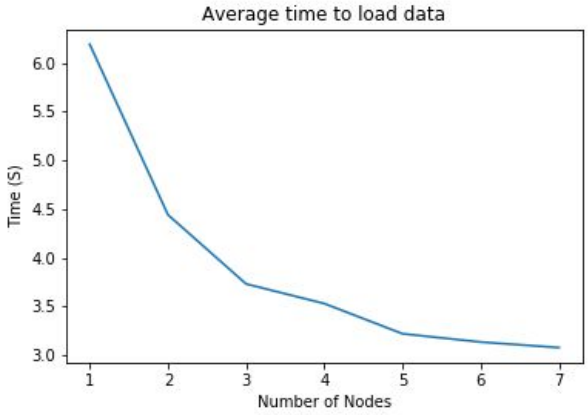

Figure 5: Data loading time for different parallelisms

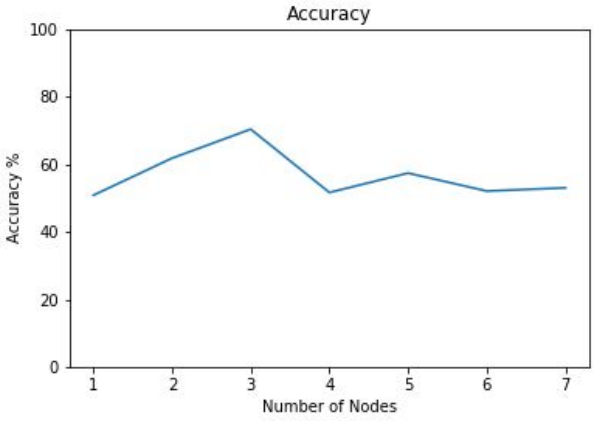

Figure 6: Training accuracy for different parallelisms

Future Work

Current performance testing was carried out for cached datasets where data is pre-loaded before training begins. Another aspect that we have to test is how the framework reacts to streaming datasets as it is extremely important for reinforcement learning use cases. In addition to that, we can evaluate how this can be translated into the Cylon project and see how Cylon data models can improve the performance of the data loading section.

We can further make these changes as first class machine learning components in the Twister2 Python API as same as how Spark is doing in their machine learning library.

References

[1] https://twister2.org/docs/examples/tset/tset_source
[2] https://www.tensorflow.org/guide/distributed_training

Appendix

Twister2 integration with Tensorflow in streaming mode

```
from twister2.TSetContext import TSetContext
from twister2.Twister2Environment import Twister2Environment
from twister2.tset.fn.SourceFunc import SourceFunc
import requests
import scipy.io as sio
import numpy as np
import tensorflow as tf
from tensorflow import keras
import threading
import queue
import time
import socket
import sys
import os
import pandas as pd
import pyarrow as pa

cache = []        # Cache for already seen data
batch_size = 100  # Minibatch size

class MNISTTrainingSource(SourceFunc):
```

```
    """ Source for MNIST dataset. This returns an image (28 x 28) with class index per
entry"""
    def __init__(self, worker_id):
        super(MNISTTrainingSource, self).__init__()
        import tensorflow as tf
        import os

        data_url = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'
        data_path = tf.keras.utils.get_file('mnist.npz', data_url)
        self.index = -1
        self.worker_id = worker_id
        with np.load(data_path) as data:
            self.train_x = data['x_train']
            self.train_y = data['y_train']

    def has_next(self):
        return self.index < self.train_x.shape[0] - 1

    def next(self):
        self.index += 1
        return np.array([self.train_x[self.index], self.train_y[self.index]])

class QueuStat:
    """Keep the queue and state of the queue to transfer data from socket server to tf data
generator"""
    done = False           # if done = True, this queue is no longe usable.
    q = queue.Queue()      # Thread safe blocking queue

    def put(self, data):
        self.q.put(data)

class StreamReaderServer(threading.Thread):
    """THis is the unix socket server to accept messages from compute client"""
    def init(self, path, queue_stat):

        try:
            os.unlink(path)
        except OSError:
            if os.path.exists(path):
                raise
        self._sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)  # Create UNIX socket
server
        self._sock.bind(path)
        self._sock.listen(1)
        self._schema = None
        self._batches = []
        self._table = None
        self._queue_stat = queue_stat
        return path

    def run(self):
        connection, client_address = self._sock.accept()
        print("Connection established")
        try:
            source = connection.makefile(mode='rb')
            print("Reading incoming messages")
            while (True):
```

```
                reader = pa.ipc.open_stream(source)             # Accepts messages as Arrow
format
                self._schema = reader.schema                    # Get Arrow schema
                for i, batch in enumerate(reader):
                    #print(batch.to_pandas())
                    self._queue_stat.put(batch.to_pandas())     # Translate Arrow message to
pandas and push into queue.
                                                                # NOTE: This is a place to
optimization. Try to use binary Arrow format
                                                                #
https://arrow.apache.org/docs/python/generated/pyarrow.binary.html
        except:
            print("Loop exited")            # This is a nsty hack I did to capture the end of
stream. Client sends a 1 byte request to fail the server at the end of the iterator
            self._queue_stat.done = True    # Queue is no longe being used
        finally:
            connection.close()

def compute_func(itr, collector, ctx:TSetContext):
    """ Twister2 compute callback"""

    class StreamingClient:
        def connect(self, server_path):
            self._sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
            self._sock.connect(server_path)
            self.sink = self.get_sink()

        def get_sink(self):
            return self._sock.makefile(mode='wb')

        def get_writer(self, sink, schema):
            return pa.RecordBatchStreamWriter(sink, schema)

        def flush(self):
            self.sink.flush()

        def close(self):
            self.flush()
            self._sock.close()

    print("Starting client on unix path " + unix_path )
    sc = StreamingClient()
    sc.connect(unix_path)

    for entry in itr:
        dfx = pd.DataFrame(np.array(entry[0], dtype='uint8').flatten())  # Fetch x (28,28) ->
(784,)
        dfy = pd.DataFrame([entry[1]])                                   # Fetch y (1)
        dfx = dfx.append(dfy)                               # Append both x and y entries into a
single dataframe (1 x 785)
        batch = pa.RecordBatch.from_pandas(dfx)             # Wrapping with Arrow batch record.
        writer = sc.get_writer(sc.sink, batch.schema)
        writer.write_batch(batch)                           # Write to socket in Arrow format
        writer.close()

    sc.flush()                                              # If itr is completed, flush
whatever remained
```

```
    pa.serialize(np.zeros((1))).write_to(sc.sink)         # HACK!! Faorce the server to fail
with junk input
    sc.flush()

def mnist_generator(stat):
    """ Generator function for TF """
    stream_data = True  # Flag to turn on straming / straightforward generating

    if (stream_data):
        # Use twister2 source to generate
        if (not stat.done):                                          # Check if queue is still
being used
            while(True):

                if (stat.q.empty()):                                 # This is to check the
termination of stream. One indiation is an emty queue but you can't say 100% without checking
stat.done state
                    timed_out = False
                    retry_count = 20
                    for retry in range(retry_count):
                        time.sleep(1)                               # NOTE: This is a naive
fixed timed valdation logic. Replace this with exponential decaying algorithm
                        if stat.done or not stat.q.empty():
                            break
                        if (retry_count - 1) == retry:
                            timed_out = True
                    if stat.done or timed_out:                      # If stat.done = True of
timed out, exit loop.
                        print("Breaking ........" + str(stat.done))
                        break

                out = stat.q.get()        # if there is data in queue, pull next one
                cache.append(out)         # append to cache for next epochs
                yield (out[:784].to_numpy().reshape((28, 28)),
out[784:].to_numpy().flatten()[0])   # Return (x = (28,28), y = (1))

        else:
            for out in cache:             # If the queue is not being used, ie: epochs > 1,
use cache to generate
                yield (out[:784].to_numpy().reshape((28, 28)),
out[784:].to_numpy().flatten()[0])

    else:
        # This section is for benchmarking. This doesn't use streaming apis
        data_url = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'
        data_path = tf.keras.utils.get_file('mnist.npz', data_url)
        with np.load(data_path) as data:
            print(data['y_train'].shape)
            for x, y in zip(data['x_train'], data['y_train']):
                yield (x, y)

def get_dataset(batch_size):
    """ Returns TF.Dataset from generator """
    generator = lambda: mnist_generator(stat)
    return tf.data.Dataset.from_generator(
        generator, (tf.int32, tf.int32), ((28, 28), ())).batch(batch_size)
```

```
def model_fit(ds):
    """Create and fit a Keras logistic regression model."""
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    model.compile(
        loss=tf.keras.losses.sparse_categorical_crossentropy,
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001, amsgrad=True),
        metrics=['accuracy'])
    model.fit(ds, epochs=5, shuffle=False)
    return model

env = Twister2Environment(resources=[{"cpu": 1, "ram": 512, "instances": 1}])

print("Hello from worker %d" % env.worker_id)

source_x = env.create_source(MNISTTrainingSource(TSetContext.worker_id), 1)
unix_path = "sockpath-" + str(env.worker_id)
stat = QueuStat()

server = StreamReaderServer()
server.init(unix_path, stat)

print("Starting unix sokcket server " + str(env.worker_id))
server.start()       # Start socket server

print("Stating unix socket client " + str(env.worker_id))

def read_from_source():
    a = source_x.compute(compute_func)
    a.for_each(lambda i : i)

source_t = threading.Thread(target=read_from_source, args=())  # Start collector in a seperate
thread to utilize main thread for TF
source_t.start()

model_fit(get_dataset(batch_size))    # Running model
```

Twister2 integration with Tensorflow distributed with cached mode

```
from twister2.TSetContext import TSetContext
from twister2.Twister2Environment import Twister2Environment
from twister2.tset.fn.SourceFunc import SourceFunc
import requests
import scipy.io as sio
import numpy as np
import tensorflow as tf
from tensorflow import keras
import threading
import queue
import time
import socket
import sys
```

```
import os
import pandas as pd
import pyarrow as pa
import json
import traceback
from collections import Counter

class MNISTTrainingSource(SourceFunc):
    """ Source for MNIST dataset. This returns an image (28 x 28) with class index per
entry"""
    def __init__(self, worker_id, train, num_workers):
        super(MNISTTrainingSource, self).__init__()
        import tensorflow as tf
        import os

        data_url = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'
        data_path = tf.keras.utils.get_file('mnist.npz', data_url)
        self.data_points = 50000
        self.data_points_per_worker = int(self.data_points/ num_workers)
        self.start_index = self.data_points_per_worker * int(worker_id)
        if self.start_index % 2 == 1:
             self.start_index += 1
        self.index = self.start_index
        self.worker_id = worker_id
        self.train = train
        with np.load(data_path) as data:
            self.train_x = data['x_train']
            self.train_y = data['y_train']

    def has_next(self):
        #return self.index < self.train_x.shape[0] * 2 - 1
        return self.index < self.start_index +  self.data_points_per_worker # For simplicity,
go with a small data set

    def next(self):
        data = []
        if self.train:
            if (self.index % 2) == 0:
                data = self.train_x[int(self.index/ 2)].flatten()
            else:
                data = self.train_y[int((self.index -1) /2)].flatten()
        else:
            data = self.test_x[self.index].flatten()
        self.index += 1
        return data


def mnist_generator(train_data, worker_id, num_workers):
    data_index = 0
    posted_count = 0
    train_point = []
    for partition in train_data.get_partitions():
        for consumer in partition.consumer():
            if (data_index % 2) == 0:
                train_point = consumer
            else:
                yield(train_point, consumer)
```

```
            posted_count += 1
            data_index += 1
    print("POSTED COUNT " + str(posted_count))

def get_dataset(batch_size, train_data, worker_id, num_workers):
    """ Returns TF.Dataset from generator """
    generator = lambda: mnist_generator(train_data, worker_id, num_workers)
    return tf.data.Dataset.from_generator(
        generator, (tf.int32, tf.int32), ((784), (1))).batch(batch_size)

def model_fit_dist(ds, strategy):
    try:
        with strategy.scope():
            model = tf.keras.Sequential([
                tf.keras.layers.Dense(784, activation='relu'),
                tf.keras.layers.Dense(64, activation='relu'),
                tf.keras.layers.Dense(10, activation='softmax')
            ])
            model.compile(
                loss=tf.keras.losses.sparse_categorical_crossentropy,
                optimizer=tf.keras.optimizers.Adam(learning_rate=0.001, amsgrad=True),
                metrics=['accuracy'])

        model.fit(ds, epochs=5, shuffle=False, steps_per_epoch=70)
    except:
        print("Model build failed....",  sys.exc_info()[0])
        traceback.print_exc()

def configure_tf_dist_env(tw2env, base_port = 12345):
    """ Configures Tensorflow distributed environment. This genretates Tensorflow specific
cluster configs """
    peer_map = tw2env.peers()
    peer_summary = Counter(peer_map.values()) # Handles workers schedule in same node
    tf_worker_connections = []

    for i in range(len(peer_map)):
        worker_host = peer_map[i]
        peer_summary[worker_host] = peer_summary[worker_host] - 1
        # If there are more than 1 worker in same node, increment base port to avoid conflicts
        tf_worker_connections.append(worker_host + ":" + str(base_port +
peer_summary[worker_host]))

    print("Tensorflow Worker Connections " + str(tf_worker_connections))

    os.environ['TF_CONFIG'] = json.dumps({
        'cluster': {
            'worker': tf_worker_connections
        },
        'task': {'type': 'worker', 'index': tw2env.worker_id}
    })

WORKERS = 4

env = Twister2Environment(resources=[{"cpu": 1, "ram": 2048, "instances": WORKERS}])
print("Hello from worker %d" % env.worker_id)
configure_tf_dist_env(env)
```

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

data_start = time.time()
source_train = env.create_source(MNISTTrainingSource(env.worker_id, train = True, num_workers
= len(env.peers())), WORKERS)
cache_train = source_train.cache()
train_data = cache_train.get_data()
data_end = time.time()


fit_start = time.time()
model_fit_dist(get_dataset(batch_size = 16, train_data = train_data, worker_id =
env.worker_id, num_workers = len(env.peers())), strategy)
fit_end = time.time()

print("DATA LOADING TIME")
print(data_end - data_start)

print("TRAINING TIME")
print(fit_end - fit_start)
```