# ID

This chapter emphasizes the implementation details of the proposed solution, how it was successfully applied in the related predictive analytics case study, as well as the working principles of TensorFlow that were applied in this project.

The entire project is implemented using Python language programming [1] and other libraries useful for:

* scientific computing in Python – NumPy [2];
* big-data manipulation and analysis – Pandas [3];
* evaluation and optimization of computations based on dataflow graphs – TensorFlow [4]

## TensorFlow graphs and sessions

Before the parallel evaluation/optimization of tensorial computation graphs using this current project, the dependencies between all the operations should be defined. TensorFlow provides an elegant and also efficient programming model for defining such graphs. The following code snippet presents a concrete example of creating the simple acyclic graph shown in TODO Figure 1 (replacing the generic “NonLinear transformation” with SELU – scaled exponential linear units [5]), using Python programming interface and NumPy scientific computing library for parameters initialization. Simultaneously, the code snippet illustrates the placement on the particular GPU device *‘/gpu:0’.*

**Code 3** Graph Definition in Tensorflow using Python programming interface

import tensorflow as tf

import numpy as np

np\_W = np.random.normal(size=(nr\_features, nr\_hidden)) # the parameters are chosen from a Gaussian distribution

tf\_graph = tf.Graph():

with tf\_graph.as\_default(), tf.device(’/gpu:0’):

tf\_X = tf.placeholder(dtype = tf.float32,

shape = (nr\_observations, nr\_features),

name = ’input\_X’)

tf\_W = tf.Variable(initial\_value = np\_W,

dtype = tf.float32,

name = ’parameters\_W’)

tf\_b = tf.Variable(initial\_value = tf.zeros([nr\_hidden]),

dtype = tf.float32,

name = ’parameters\_b’)

tf\_matmul = tf.matmul(tf\_X, tf\_W)

tf\_add = tf.add(tf\_matmul, tf\_b)

tf\_nonlinear = tf.nn.selu(tf\_add)

tf\_init = tf.global\_variables\_initializer() # operation that initializes the variables for this specific computation graph

tf.train.write\_graph(

graph\_or\_graph\_def = tf\_graph,

logdir = ’.’,

name = ’simple\_acyclic\_graph.pb’,

as\_text = False

) # write the graph definition that is loaded in the proposed graph evaluator and allocator

A TensorFlow Session represents the communication pipe between the programming interface and the C++ runtime that executes all the instructions. This session is set up with a TensorFlow graph and provides an operation (*Run*) which takes a list of outputs that should be processed and a list of input tensors that should be fed with data in order to compute the output tensors.

The proposed hypervisor is able to run a list of outputs, either for optimization or for evaluation tasks, using the configuration file that is mandatory for each job and to feed data for the input tensors using the generator. Code snippets 4 and 5 briefly present the routines that are executed by each thread for both use-cases: evaluation and optimization.

**Code 4** Summary of Thread Routine for Evaluation Task

tf\_session = tf.Session(graph=tf\_graph)

# signals the C++ runtime to allocate memory for the variables on the corresponding device (for example: ’/gpu:0’)

tf\_session.run(tf\_init)

**feed\_tensors = [tf\_graph.get\_tensor\_by\_name(n) for n in**

**json\_config[**’INPUT\_TENSORS’**]]**

**output\_tensors = [tf\_graph.get\_tensor\_by\_name(n) for n in**

**json\_config[**’EVALUATION’**][**’OPERATIONS’**]]**

generator\_obj = generator()

all\_results = []

while True:

try:

feed\_data = generator\_obj.next()

**except StopIteration: # all the chuncks were processed**

**break**

**feed\_dict = dict(zip(feed\_tensors, feed\_data))**

**all\_results.append(session.run(output\_tensors, feed\_dict=feed\_dict))**

**return all\_results**

**Code 4** Summary of Thread Routine for Optimization Task

tf\_session = tf.Session(graph=tf\_graph)

# signals the C++ runtime to allocate memory for the variables on the corresponding device (for example: ’/gpu:0’)

tf\_session.run(tf\_init)

**feed\_tensors = [tf\_graph.get\_tensor\_by\_name(n) for n in**

**json\_config[**’INPUT\_TENSORS’**]]**

**output\_tensors = [tf\_graph.get\_tensor\_by\_name(n) for n in**

**json\_config[**’OPTIMIZATION’**][**’OPERATIONS’**]]**

all\_results = dict()

# The optimization is executed iteratively

for iteration in **json\_config[**’OPTIMIZATION’**][**’NR\_ITERATIONS’**]:**

all\_results[iteration] = []

generator\_obj = generator()

while True:

try:

feed\_data = generator\_obj.next()

**except StopIteration: # all the chunks were processed**

**break**

**feed\_dict = dict(zip(feed\_tensors, feed\_data))**

**all\_results[iteration].append(session.run(output\_tensors,**

**feed\_dict=feed\_dict))**

**return all\_results**

## Data Analysis and Generation the Study Case in Predictive Analytics Area

The particular case analyzed in this thesis comes from the predictive analytics area and it refers to a real-life example of inferring customer buy propensities based on market basket analytics using tensorial graphs computation on massive amount of transactional information. These real commercial applications were released in 2018 for a large pharma retail company and the struggle (treated in this thesis) was to create a parallel tensorial graph numerical computation system that has to process all the graph architectures on resource limited mobile computing environments in a short amount of time.

Besides the development of such a system (named <PLACEHOLDER>) composed of a) the hypervisor which initializes threads that runs session in a parallel and synchronized fashion and b) the resource evaluator and allocator engine, the project needed to adopt a general method for data generation. The solution proposed that for each computation graph to exist a generator – provider of data, and the hypervisor to call this generator independent of the structure of the tensorial DAG.

For this particular study case, the transactional information dataset has well over 200 million observations summing over 50GB of data. Loading all the data into the virtual memory of the host is not even possible because, nowadays, the limited structures can rarely reach 16GB RAM. Therefore, a general solution is proposed which use Pandas library that can easily manipulate big-data by reading the files incrementally (chunks). **Code** snippet **6** presents the approach for reading data from the transactional dataset, which is applied for any dataset. The *yield* mechanism in Python results in a generator which does not keep all the values in memory, generating them “on-the-fly”.

**Code 6** Data Generation for Market Basket Analysis

import pandas as pd

# handle to the transactional dataset file.

pd\_reader = pd.read\_csv(’transactions.csv’, chunksize = chunksize, names = [TIMESTAMP’, ’TRAN\_UNIQUE\_ID’, ’PRODUCT\_ID’, ’CUSTOMER\_ID’]

def mba\_generator(file\_handle):

for pd\_chunk in file\_handle:

inputs = process\_chunk(pd\_chunk)

yield inputs

# reference that is passed to the <PLACEHOLDER> hypervisor along with the graph and

# the configuration JSON for that graph

generator = mba\_generator(pd\_reader)

# RESULTS

This chapter will show up the results obtained using the developed parallel DAG computing system and resource memory allocator (<PLACEHOLDER>) for the particular case of predictive analytics tensorial graphs.

Using the Deep Learning techniques presented in [6], the resulting graphs generate semantical and behavioral information about the products (TODO Appendix 1) and the users of the big pharma retail company used for the case study. The Deep Learning techniques are meant to discover common behaviors between people, products that can be placed in the same market baskets (recommendations), buying propensities of each customer for any of the existing products, or to predict the next basket that a customer will prefer.

Before the development of this project (<PLACEHOLDER>), all the graphs resulted from the particular presented Deep Learning techniques were sequentially run. With the completion of this work, they can be evaluated or optimized in a parallel fashion with partial or full data sharing, based on two main components: a) the hypervisor which initializes threads that runs session in a parallel and synchronized fashion and b) the resource evaluator and allocator engine.

This development leads to the ability of presenting real-time, using limited resource infrastructures, the results of the Deep Learning techniques based for any potential client, based on their confidential transactional data which cannot be processed with cloud support.

For the pharma-retail scenario, there are pharmaceutical products ever marketed and people who have been involved in at least a purchase. Three different graphs which use the embeddings technique [7] [8] [9] are defined for this scenario and they are further presented:

* P2V (Product2Vector): represents all products on a multi-dimensional space of embeddings. The graph is composed of an embedding subgraph and a simple acyclic subgraph which receives an input with features (TODO figure 1);
* U2V (User2Vector): represents all products and all users on two multi-dimensional spaces of and embeddings. The graph is composed of 2 embedding subgraphs and a simple acyclic subgraph which receives an input with features;
* NBP (Next Basket Prediction): applies the same principles as above using two multi-dimensional spaces of embeddings, but they are processed using then a sequence-based tensorial graph (TODO Figure 3).

TODO Table x presents the <PLACEHOLDER> resource evaluator results for each of these graphs, for different values of and , based on equations TODO ecuatii

| **GRAPH** | **TASK** | **64** |  | **64; 32** | **32** |
| --- | --- | --- | --- | --- | --- |
| **P2V** | *Evaluation* | 13.27MB | 26.44MB |  |  |
| *Optimization* | 26.42MB | 52.63MB |  |  |
| **U2V** | *Evaluation* |  |  | 546.97MB | 560.14MB |
| *Optimization* |  |  | 1093.76MB | 1119.98MB |
| **NBP** | *Evaluation* |  |  | 590.56MB | 597.34MB |
| *Optimization* |  |  | 1180.94MB | 1194.37MB |

# CONCLUSIONS