# E-63 Big Data Analytics - Assignment 10 - TensorFlow


## Shanaka De Soysa


In [1]:
```
import sys
import tensorflow as tf

print(sys.version)
print(sys.version_info)
print("TensorFlow Version: {0}".format(tf.__version__))
```

```
3.5.2 |Anaconda 4.2.0 (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)]
sys.version_info(major=3, minor=5, micro=2, releaselevel='final', serial=
0)
TensorFlow Version: 1.0.1
```

### Problem 1.

Please use tf_upgrade.py utility, which you could find on the TensorFlow GitHub site to upgrade attached Python script vectorized_graph.py to TensorFlow 1.x. Demonstrate that upgraded script will run and produce TensorBoard graph and summaries. Provide working upgraded script and images of your graphs and calculated summaries. (25%)

In [7]:
```
!python tf_upgrade.py --infile vectorized_graph.py --outfile vectorized_grap
```

```
TensorFlow 1.0 Upgrade Script
-----------------------------
Converted 1 files

Detected 0 errors that require attention
--------------------------------------------------------------------------
-------


Make sure to read the detailed log 'report.txt'
```

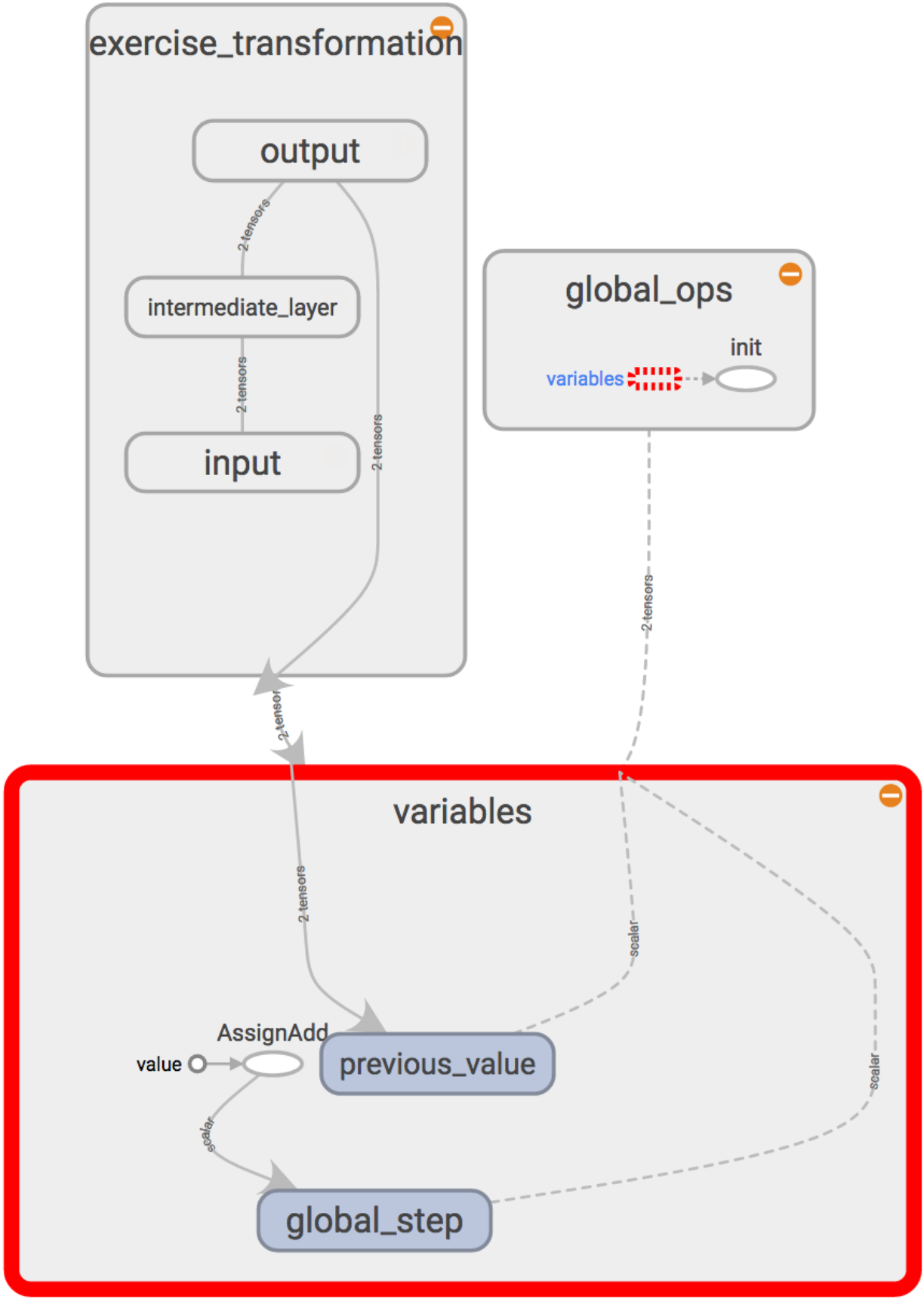
#### Inspect the report.txt file

#### Upgraded script

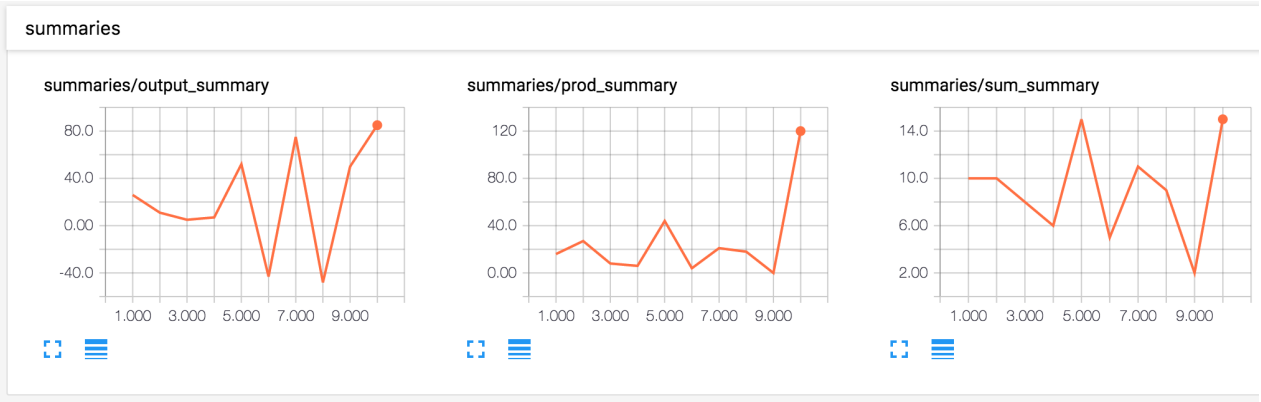
In [ ]:
```
import tensorflow as tf
import numpy as np

LOG_FILE = 'logs/improved_graph'

# Explicitly create a Graph object
graph = tf.Graph()

with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, name="global_step")

        # Increments the above `global_step` Variable, should be run whenev
        # the graph is run
        increment_step = global_step.assign_add(1)

        # Variable that keeps track of previous output value:
        previous_value = tf.Variable(0.0,
                                     dtype=tf.float32,
                                     name="previous_value")

    # Primary transformation Operations
    with tf.name_scope("exercise_transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32,
                               shape=[None],
                               name="input_placeholder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")

        # Separate output layer
        with tf.name_scope("output"):
            d = tf.add(b, c, name="add_d")
            output = tf.subtract(d, previous_value, name="output")
            update_prev = previous_value.assign(output)

    # Summary Operations
    with tf.name_scope("summaries"):
        # Creates summary for output node
        tf.summary.scalar("output_summary" ,output)
        tf.summary.scalar("prod_summary", b)
        tf.summary.scalar("sum_summary", c)

    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.global_variables_initializer()
        # Collect all summary Ops in graph
```

```
        merged_summaries = tf.summary.merge_all()

# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)

# Open a SummaryWriter to save summaries
writer = tf.summary.FileWriter(LOG_FILE, graph)

# Initialize Variables
sess.run(init)


def run_graph(input_tensor):
    """
    Helper function; runs the graph with given input tensor and saves summar
    """
    feed_dict = {a: input_tensor}
    output, summary, step = sess.run(
        [update_prev, merged_summaries, increment_step],
        feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)


# Run the graph with various inputs
run_graph([2, 8])
run_graph([3, 1, 3, 3])
run_graph([8])
run_graph([1, 2, 3])
run_graph([11, 4])
run_graph([4, 1])
run_graph([7, 3, 1])
run_graph([6, 3])
run_graph([0, 2])
run_graph([4, 5, 6])

# Writes the summaries to disk
writer.flush()

# Flushes the summaries to disk and closes the SummaryWriter
writer.close()

# Close the session
sess.close()

# To start TensorBoard after running this file, execute the following commar
# $ tensorboard --logdir='./improved_graph'
```

**Tensorboard Graph**

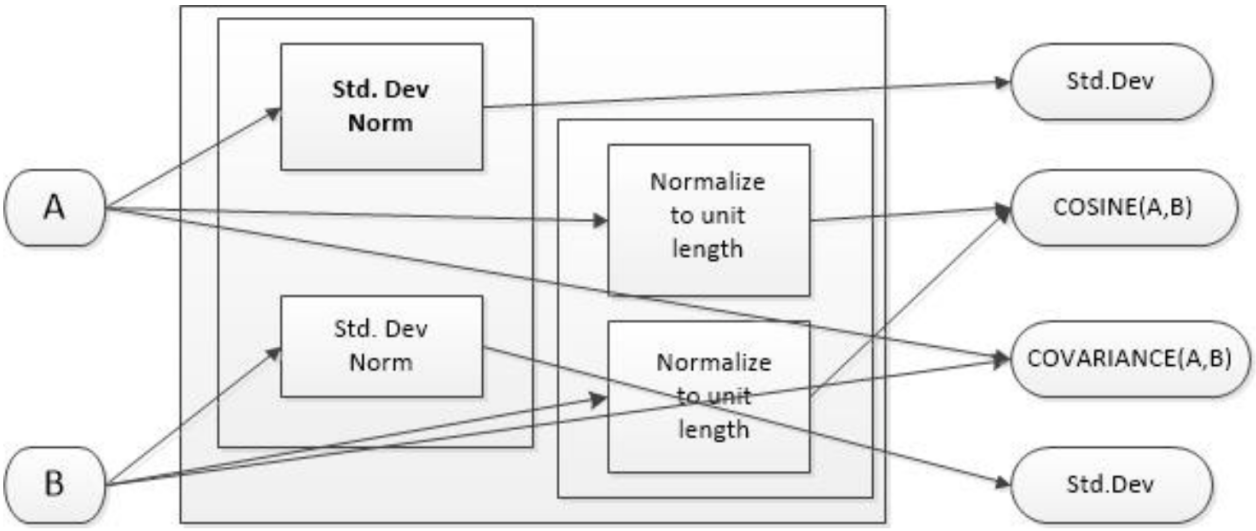



**Tensorboard Summaries**




## Problem 2.

Please construct a graph that will accept as inputs two vectors of equal length (tensors of dimension 1) and perform the operations on those vectors as depicted in the drawing bellow. Organize your variables and operations in nested namespaces as suggested by the nested boxes in the same graph. Organize your program in such a way that it repeats calculations in the graphs for 8 vectors of different lengths and element values. Collect and display in the TensorBoard as summaries the results on the right. (25%)



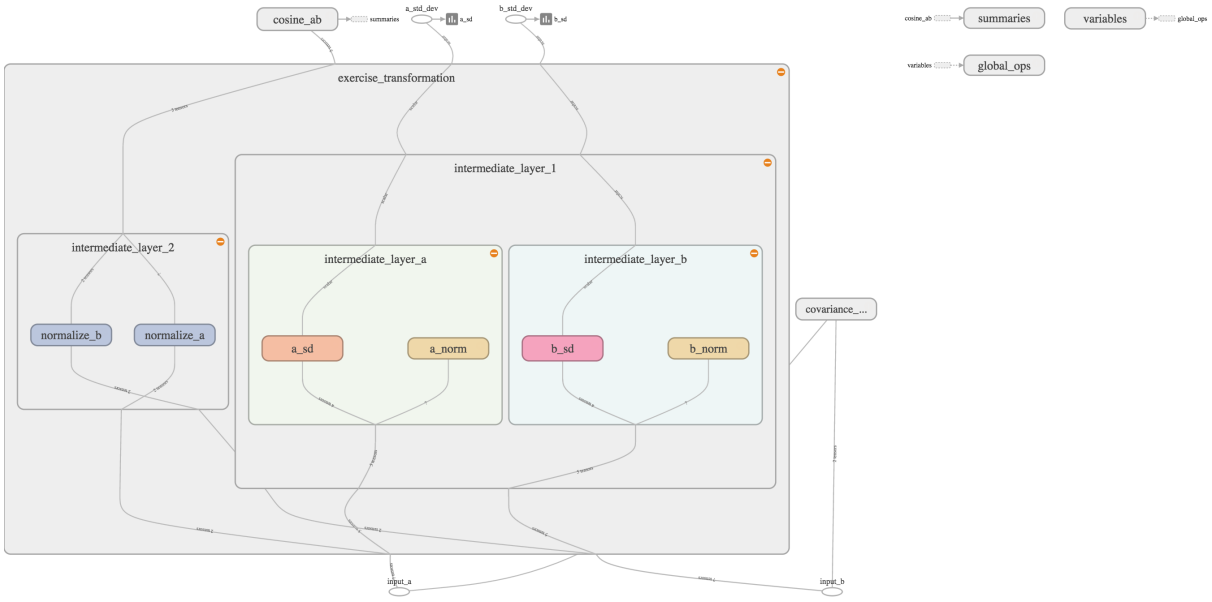

**Python script**

In [ ]:
```
import tensorflow as tf
import numpy as np

LOG_FILE = 'logs/p2'

# Explicitly create a Graph object
graph = tf.Graph()

with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, name="global_step")

        # Increments the above `global_step` Variable, should be run whenev
        # the graph is run
        increment_step = global_step.assign_add(1)

    a = tf.placeholder(tf.float32,
                       shape=[None],
                       name="input_a")
    b = tf.placeholder(tf.float32,
                       shape=[None],
                       name="input_b")

    # Primary transformation Operations
    with tf.name_scope("exercise_transformation"):
        # Separate input layer
        with tf.name_scope("intermediate_layer_1"):
            # Create input placeholder- takes in a Vector
            with tf.name_scope("intermediate_layer_a"):
                a_moments = tf.nn.moments(a, axes=[0], name="a_sd")
                a_norm = tf.norm(a, name="a_norm")

            with tf.name_scope("intermediate_layer_b"):
                b_moments = tf.nn.moments(b, axes=[0], name="b_sd")
                b_norm = tf.norm(b, name="b_norm")

        # Separate middle layer
        with tf.name_scope("intermediate_layer_2"):
            a_normalized = tf.nn.l2_normalize(a, dim=0, name="normalize_a")
            b_normalized = tf.nn.l2_normalize(b, dim=0, name="normalize_b")

    # Separate output layer
    with tf.name_scope("cosine_ab"):
        b_normalized_T = tf.transpose([b_normalized])
        cosine_similarity = tf.matmul([a_normalized],
                                      b_normalized_T)

    a_sd = tf.sqrt(a_moments[1], name="a_std_dev")
    b_sd = tf.sqrt(b_moments[1], name="b_std_dev")

    with tf.name_scope("covariance_ab"):
        a_mean = tf.cast(tf.reduce_mean(a), tf.float32)
        b_mean = tf.cast(tf.reduce_mean(b), tf.float32)
        a_delta = a - a_mean
```

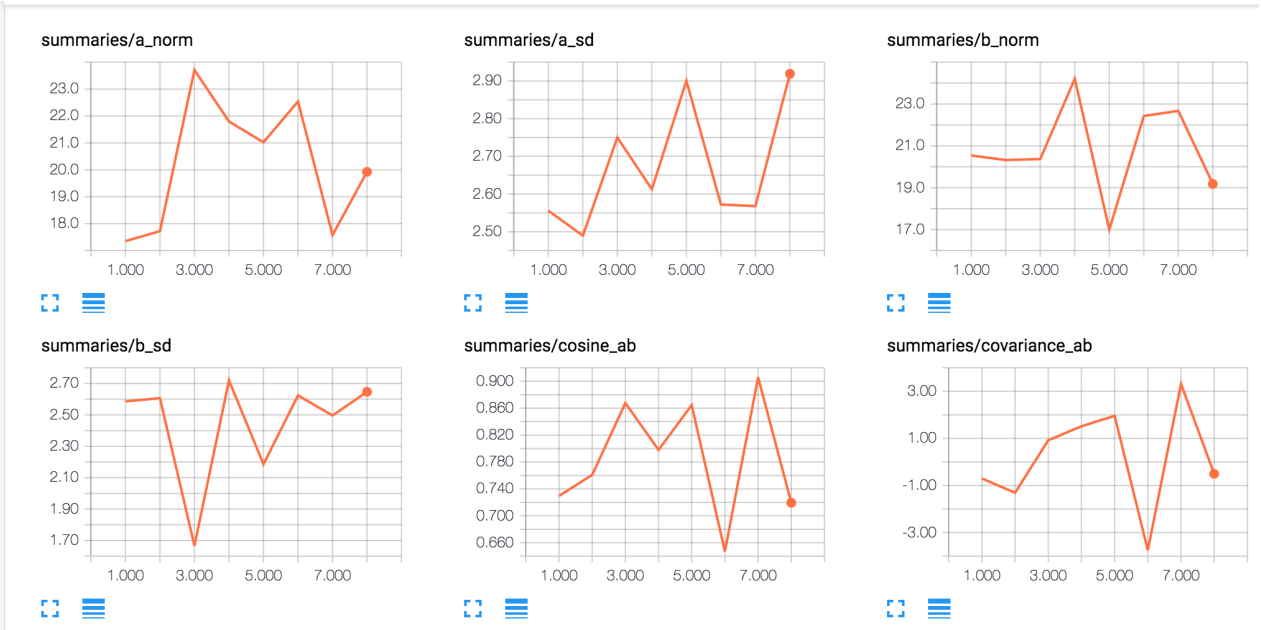
```
        b_delta = b - b_mean
        covariance = tf.reduce_mean(tf.multiply(a_delta, b_delta))

    # Summary Operations
    with tf.name_scope("summaries"):
        # Creates summary for output node
        tf.summary.scalar("a_sd", a_sd)
        tf.summary.scalar("b_sd", b_sd)
        tf.summary.scalar("a_norm", a_norm)
        tf.summary.scalar("b_norm", b_norm)
        tf.summary.scalar("cosine_ab", cosine_similarity[0][0])
        tf.summary.scalar("covariance_ab", covariance)

    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.global_variables_initializer()
        # Collect all summary Ops in graph
        merged_summaries = tf.summary.merge_all()

# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)

# Open a SummaryWriter to save summaries
writer = tf.summary.FileWriter(LOG_FILE, graph)

# Initialize Variables
sess.run(init)


def run_graph(input_tensor1, input_tensor2):
    """
    Helper function; runs the graph with given input tensor and saves summar
    """
    feed_dict = {a: input_tensor1, b: input_tensor2}
    # a_sd_val, b_sd_val, a_norm_val, b_norm_val, cosine_similarity_val, cov
    #     [a_sd, b_sd, a_norm, b_norm, cosine_similarity, covariance, merged
    # print("a_sd: {0}, b_sd: {1}, a_norm: {2}, b_norm: {3}, cosine: {4}, co
    # format(a_sd_val, b_sd_val, a_norm_val, b_norm_val,
    # cosine_similarity_val, covariance_val))
    summary, step, a_mean_val, b_mean_val, covariance_val = sess.run(
        [merged_summaries, increment_step, a_mean, b_mean, covariance], feed
    writer.add_summary(summary, step)
    print("a_mean: {0}, b_mean: {1}, cov: {2}".format(
        a_mean_val, b_mean_val, covariance_val))


#run_graph([3.0, 5.0, 355.0, 3.0], [22.0, 111.0, 3.0, 10.0])
#run_graph([3, 1, 3, 3],[3, 1, 3, 3])

def run_graph_with_random_vectors(iterations=8, seed=1234):
    np.random.seed(seed)
    for i in range(iterations):
        v_len = np.random.randint(10, 20)
        print("Vector length: {0}".format(v_len))
        x, y = [], []
        for j in range(v_len):
```

```
            x.append(np.random.randint(1, 10))
            y.append(np.random.randint(1, 10))
        print("x: {0}".format(x))
        print("y: {0}".format(y))
        run_graph(x, y)


run_graph_with_random_vectors()


# Writes the summaries to disk
writer.flush()

# Flushes the summaries to disk and closes the SummaryWriter
writer.close()

# Close the session
sess.close()

# To start TensorBoard after running this file, execute the following commar
# $ tensorboard --logdir='./improved_graph'
```

### TensorBoard Graph




### TensorBoard Summaries




## Problem 3.

Fetch Iris Dataset from https://archive.ics.uci.edu/ml/datasets/Iris (https://archive.ics.uci.edu/ml/datasets/Iris) and make attached Python script, softmax_irises.py work. You might have to upgrade the script to TF 1.x API. Generate TensorBoard graph of the process and use scalar summary to presenting variation of the loss function during the training process. Report the results of the evaluation process. (25%)

### Upgraded and improved code

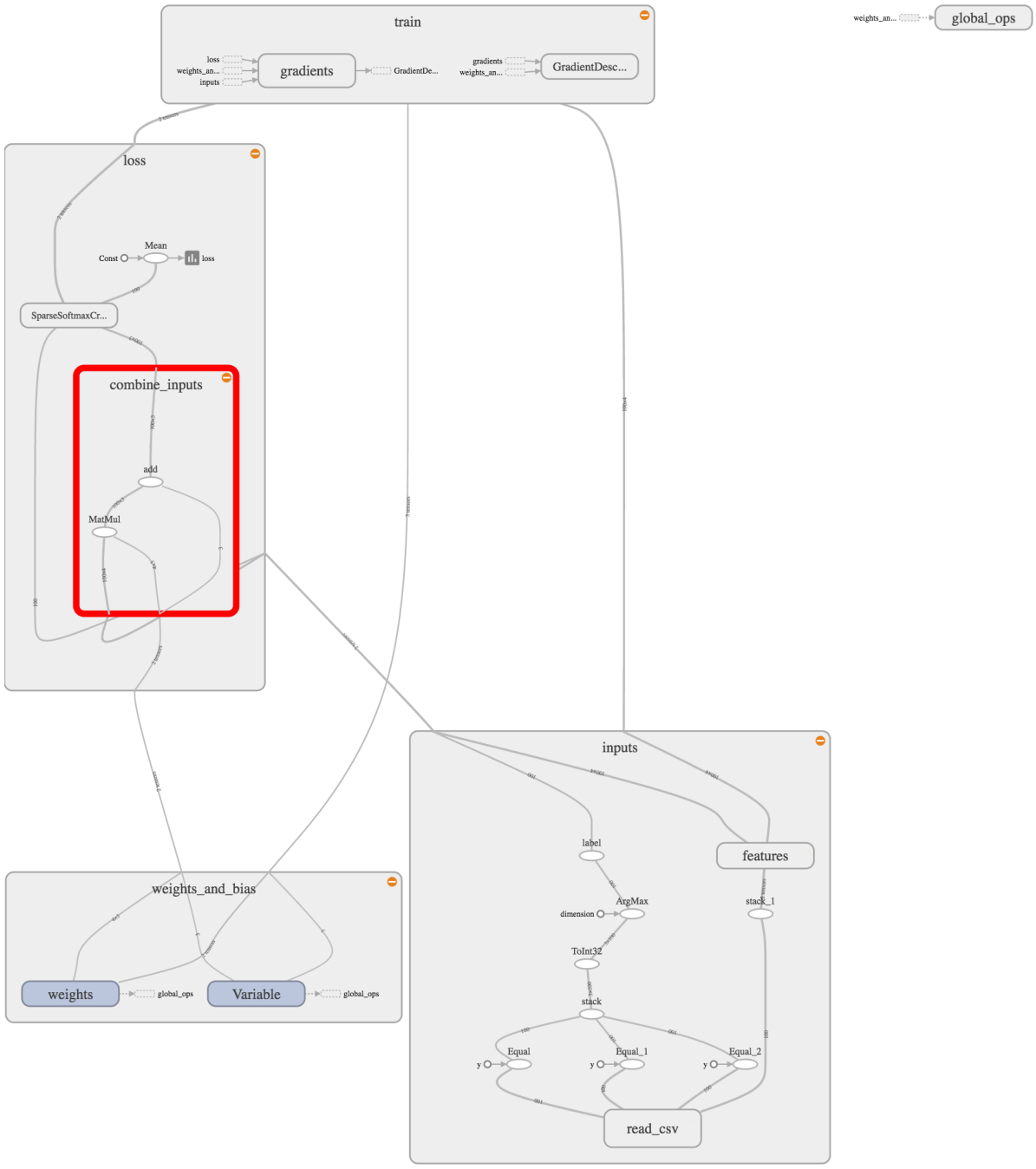
In [ ]:

```
# pylint: disable=invalid-name

# Softmax example in TF using the classical Iris dataset
# Download iris.data from https://archive.ics.uci.edu/ml/datasets/Iris

import os
import tensorflow as tf

DATA_FILE = "data/IrisDataSet.csv"
LOG_FILE = "logs/p3_iris"


def combine_inputs(X):
    with tf.name_scope("combine_inputs"):
        return tf.matmul(X, W) + b


def inference(X):
    with tf.name_scope("inference"):
        return tf.nn.softmax(combine_inputs(X))


def loss(X, Y):
    with tf.name_scope("loss"):
        return tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                logits=combine_inputs(X),
                labels=Y))


def read_csv(batch_size, file_name, record_defaults):
    with tf.name_scope("read_csv"):
        filename_queue = tf.train.string_input_producer(
            [os.path.dirname(__file__) + "/" + file_name])

        reader = tf.TextLineReader(skip_header_lines=1)
        key, value = reader.read(filename_queue)

        # decode_csv will convert a Tensor from type string (the text line)
        # a tuple of tensor columns with the specified defaults, which also
        # sets the data type for each column
        decoded = tf.decode_csv(
            value, record_defaults=record_defaults, name="decode_csv")

        # batch actually reads the file and loads "batch_size" rows in a si
        # tensor
        return tf.train.shuffle_batch(decoded,
                                      batch_size=batch_size,
                                      capacity=batch_size * 50,
                                      min_after_dequeue=batch_size,
                                      name="shuffle_batch")


def inputs():
    with tf.name_scope("inputs"):
        sepal_length, sepal_width, petal_length, petal_width, label =\
```

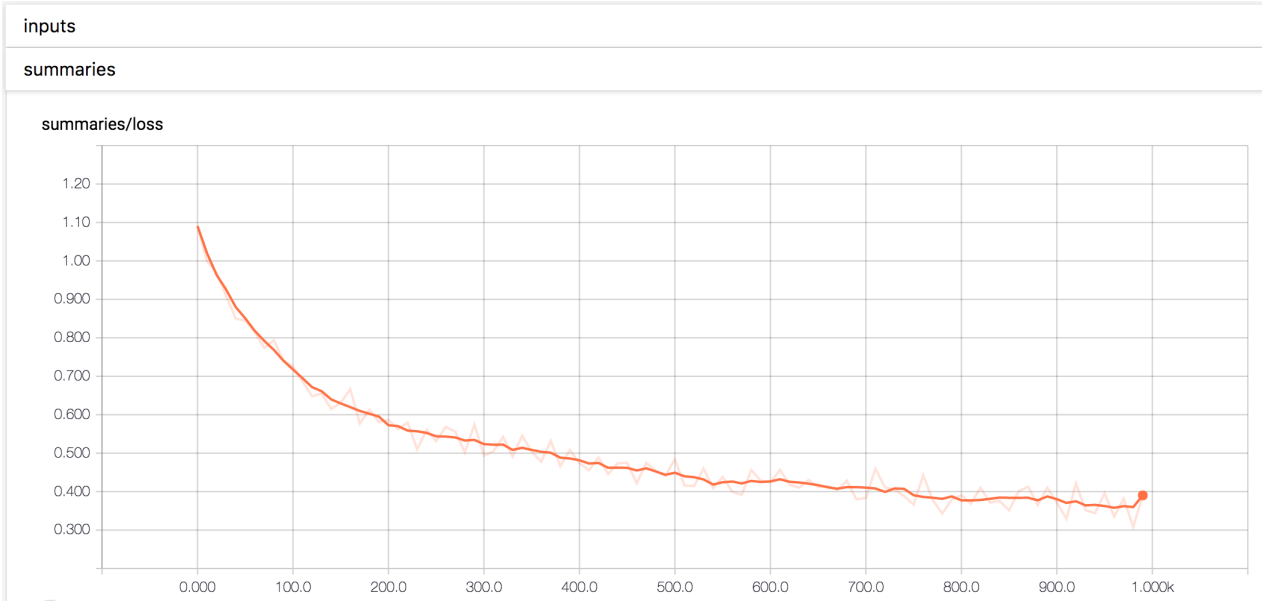
```
            read_csv(100, DATA_FILE, [[0.0], [0.0], [0.0], [0.0], [""]])

        # convert class names to a 0 based class index.
        label_number = tf.to_int32(tf.argmax(tf.to_int32(tf.stack([
            tf.equal(label, ["Iris-setosa"]),
            tf.equal(label, ["Iris-versicolor"]),
            tf.equal(label, ["Iris-virginica"])
        ])), 0), name="label")

        # Pack all the features that we care about in a single matrix;
        # We then transpose to have a matrix with one example per row and o
        # feature per column.
        features = tf.transpose(tf.stack(
            [sepal_length, sepal_width, petal_length, petal_width]), name="f

        return features, label_number


def train(total_loss):
    with tf.name_scope("train"):
        learning_rate = 0.01
        return tf.train.GradientDescentOptimizer(learning_rate, name="Gradie


def evaluate(sess, X, Y):
    with tf.name_scope("evaluate"):
        predicted = tf.cast(tf.arg_max(inference(X), 1), tf.int32)
        print("Evaluation: ", sess.run(tf.reduce_mean(
            tf.cast(tf.equal(predicted, Y), tf.float32))))


# Explicitly create a Graph object
graph = tf.Graph()

with graph.as_default():
    with tf.name_scope("weights_and_bias"):
        # this time weights form a matrix, not a column vector, one "weight
        # vector" per class.
        W = tf.Variable(tf.zeros([4, 3]), name="weights")
        # so do the biases, one per class.
        b = tf.Variable(tf.zeros([3], name="bias"))

    X, Y = inputs()
    total_loss = loss(X, Y)
    train_op = train(total_loss)

    with tf.name_scope("summaries"):
        # Creates summary for output node
        # Scalar summary for loss
        tf.summary.scalar("loss", total_loss)

    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.global_variables_initializer()
        # Collect all summary Ops in graph
        merged_summaries = tf.summary.merge_all()
```

```
# Launch the graph in a session, setup boilerplate
with tf.Session(graph=graph) as sess:
    # Open a SummaryWriter to save summaries
    writer = tf.summary.FileWriter(LOG_FILE, graph)

    sess.run(init)

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    # actual training loop
    training_steps = 1000
    for step in range(training_steps):
        sess.run([train_op])
        # for debugging and learning purposes, see how the loss gets
        # decremented thru training steps
        if step % 10 == 0:
            loss_val, summary_str = sess.run([total_loss, merged_summaries])
            writer.add_summary(summary_str, step)
            if step % 100 == 0:
                print("loss: ", loss_val)

    evaluate(sess, X, Y)

    # Writes the summaries to disk
    writer.flush()

    # Flushes the summaries to disk and closes the SummaryWriter
    writer.close()

    coord.request_stop()
    coord.join(threads)
    sess.close()
```

**TensorBoard Graph**




**TensorBoard Scalar summary for loss function**




**Evaluation**

loss: 0.371925
Evaluation: 0.99

## Problem 4.

Analyze all relevant and non-obvious individual steps in the script, softwmax_irises.py by examining their inputs and outputs. When convenient, use existing Iris Dataset. When convenient, you are welcome to provide your own inputs. Please examine and describe actions of functions and operations within those functions: combine_inputs(), line 13 inference(), line 17 read_csv(), line 25 decode_csv() line 34 train.shuffle_batch(), line 37 inputs(), line 43 label_number = tf.to_int32(…), line 49 features = tf.transpose(..), line 57 evaluate(), line 67 predicted = tf.cast(tf.arg_max(inference(X), 1).., line 69 tf.reduce_mean(tf.cast(tf.equal(predicted,Y),.,line 71 threads = tf.train.start_queue_runners(sess=sess, coord=coord).., line 85

Please describe the effect of every function or command by providing an illustrative input and output set of values and well as a brief narrative. Please rely on TensorFlow API as much as possible. (%25)

## combine_inputs() method

This method combine all features and makes a (?, 3) shape matrix. Note 3 is the number of labels and ? is the batch size, 100 by default.

```
def combine_inputs(X):
    with tf.name_scope("combine_inputs"):
        return_val = tf.matmul(X, W) + b
        if DEBUG:
            return_val = tf.Print(return_val,
            [return_val, tf.shape(return_val)], "combine_inputs = ",
```

```
 summarize=10)
        return return_val
```

**Sample inputs (with two records):**
b shape [3]= [0.0066666659 -0.0033333334 -0.0033333334]
W shape [4 3]= [[0.031333331 -0.015666667 -0.015666667][0.020999998 -0.010500001 -0.010500001][0.009 6666655 -0.0048333332 -0.0048333332][0.0013333332]...]
X shape [2 4]= [[5.1 3.8 1.5 0.3][5.4 3.9 1.7 0.4]]

**Sample output:**
combine_inputs shape [2 3]= [[0.26116663 -0.13058333 -0.13058333][0.2747333 -0.13736668 -0.13736668] ]

## inference() method

This method computes softmax activations.

**Sample Input (with two records)**
X shape [2 4] = [[5.1 3.3 1.7 0.5][5.1 3.7 1.5 0.4]]
**Sample output:**
inference shape [2 3] = [[0.425876 0.28706202 0.28706202][0.42834732 0.28582633 0.28582633]]

## read_csv() method

Reads the input csv file in given batch size (100).

## decode_csv() line 34

Convert CSV records to tensors. Each column maps to one tensor.

## train.shuffle_batch(), line 37

Creates batches by randomly shuffling tensors.

## inputs(), line 43. Code with debug prints

```
def inputs():
    with tf.name_scope("inputs"):
        sepal_length, sepal_width, petal_length, petal_width, label
=\
            read_csv(100, DATA_FILE, [[0.0], [0.0], [0.0], [0.0],
[""]])
```

```
            # convert class names to a 0 based class index.
            label_number = tf.to_int32(tf.argmax(tf.to_int32(tf.stack([
                tf.equal(label, ["Iris-setosa"]),
                tf.equal(label, ["Iris-versicolor"]),
                tf.equal(label, ["Iris-virginica"])
            ])), 0), name="label")

            # Pack all the features that we care about in a single matri
    x;
            # We then transpose to have a matrix with one example per ro
    w and one
            # feature per column.
            features = tf.transpose(tf.stack(
                [sepal_length, sepal_width, petal_length, petal_width]),
     name="features")

            if DEBUG:
                sepal_length = tf.Print(sepal_length, [sepal_length], "s
    epal_length = ")
                sepal_width = tf.Print(sepal_width, [sepal_width], "sepa
    l_width = ")
                petal_length = tf.Print(petal_length, [petal_length], "p
    etal_length = ")
                petal_width = tf.Print(petal_width, [petal_width], "peta
    l_width = ")
                label_number = tf.Print(label_number, [label, label_numb
    er], "label_number = ")
                features = tf.Print(features, [features], "features = ",
     summarize=5)
            return features, label_number
```

sepal_length = [7.1 5.8 6.3...]
sepal_width = [3 2.8 2.9...]
petal_width = [2.1 2.4 1.8...]
petal_length = [5.9 5.1 5.6...]
label_number = [Iris-virginica Iris-virginica Iris-virginica...][2 2 2...]
features = [[7.1 3 5.9 2.1][5.8]...]

input() method reads the csv file 100 record batches. Then it converts labels to 0 based index.

---

For example: label = [Iris-virginica Iris-virginica Iris-versicolor...] will be converted to label_number = [2 2 1...]. This creates a vector of shape [100]

---

Then it creates a matrix of features stacks them like. This results in a matrix of [100 4]: features = [[7.1 3 5.9 2.1][5.8...]...]

## label_number = tf.to_int32(…), line 49

Convert labels into 0 based index:

label = [Iris-virginica Iris-virginica Iris-versicolor...] will be converted to label_number = [2 2 1...]. This creates a vector of shape [100]

## features = tf.transpose(..), line 57

Creates a matrix of features stacks them like. This results in a matrix of [100 4]: features = [[7.1 3 5.9 2.1][5.8...]...]

## evaluate(), line 67

Calculates the predicted label. Then measures the accuracy compared to actual value.
**Sample output (with two records):**
b shape[3]= [0.0066666659 -0.0033333334 -0.0033333334]
W shape[4 3] = [[0.034333333 -0.017166667 -0.017166667][0.022999998 -0.011500001 -0.011500001][0.010 333333 -0.005166667 -0.005166667][0.0019999999]...]
X shape[2 4] = [[4.8 3 1.4 0.1][4.6 3.6 1 0.2]]
inference = [[0.42300561 0.28849721 0.28849721][0.42410427 0.28794786 0.28794786]][2 3]
**Sample output:**
predicted shape[2] = [0 0]
Evaluation: 1.0

## predicted = tf.cast(tf.arg_max(inference(X), 1).., line 69

Calculate the predicte value in 0 based index.
ex:
predicted shape[2] = [0 0]

## tf.reduce_mean(tf.cast(tf.equal(predicted,Y),.,line 71

Compare actual to predicted and calculate how accurate the model

## threads = tf.train.start_queue_runners(sess=sess, coord=coord).., line 85

Starts all queue runners collected in the graph.

This is a companion method to add_queue_runner(). It just starts threads for all queue runners collected in the graph. It returns the list of all threads.