UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Machine Learning With TensorFlow®**
- Introduction, Python - pros and cons
- Python modules, DL packages and scientific blocks
- Working with the shell, IPyton and the editor
- Installing the environment with core packages
- Writing “Hello World” HW1 (10pts)

- **Tensorflow and TensorBoard basics (cont.)**
- Ecosystem, Competition, Users
- Linear algebra recap
- Data types in Numpy and Tensorflow
- Basic operations in Tensorflow
- Graph models and structures with Tensorboard

- **TensorFlow operations**
- Sessions, graphs, variables, placeholders
- Overloaded operators
- Using Aliases

- **TF 2.0 vs. 1.X comparison**
- Name scopes HW2 (10pts)

- **Data Mining and Machine Learning concepts**
- Basic Deep Learning Models, k-Means
- Linear and Logistic Regression
- Softmax classification

- **Neural Networks 1/2**
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

TensorFlow 2.0 vs. 1.X

- Using Colab:

```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version
except Exception:
  pass
```

```
Currently selected TF version: 1.x
Available versions:
* 1.x
* 2.x
```

```
[ ] try:
      # %tensorflow_version only exists in Colab.
      %tensorflow_version
    except Exception:
      pass

    # Load the TensorBoard notebook extension.
    %load_ext tensorboard
```

```
selected TF version: 2.x
versions:
```

```
[ ] try:
      # %tensorflow_version only exists in Colab.
      %tensorflow_version 2.x
    except Exception:
      pass
```

```
TensorFlow 2.x selected.
```

TensorFlow 2.0 vs. 1.X

```
[ ]  # A simple TensorBoard test 2 v1.15:

[1]  %load_ext tensorboard
     import tensorflow as tf
```

```
The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via
the %tensorflow_version 1.x magic: more info.
```

```
pip list
```

```
sympy                         1.1.1
tables                        3.4.4
tabulate                      0.8.6
tblib                         1.6.0
tensor2tensor                 1.14.1
tensorboard                   1.15.0
tensorboardcolab              0.0.22
tensorflow                    1.15.0
tensorflow-datasets           2.0.0
```

TensorFlow 2.0 vs. 1.X

```
[3] try:
      # %tensorflow_version only exists in Colab.
      %tensorflow_version 2.x
    except Exception:
      pass

    # Load the TensorBoard notebook extension.
    %load_ext tensorboard
```

```
TensorFlow 2.x selected.
```

```
pip list
```

```
tensor2tensor                 1.14.1
tensorboard                   2.1.0
tensorboardcolab              0.0.22
tensorflow                    2.1.0
tensorflow-addons             0.6.0
```

TensorFlow 2.0 vs. 1.X

- API Cleanup
- Refactor your code into smaller functions
- Use Keras layers and models to manage variables
- Combine tf.data.Datasets and @tf.function
- tf.metrics aggregates data and tf.summary logs them
- No more globals
- Eager Execution
- Functions, not sessions
- Use tf.config.experimental_run_functions_eagerly() when debugging
- Advantage of AutoGraph with Python control flow

Basic operations

Class exercise:

Create a graph that produces a 3x4 matrix, where
an op is added to the default graph and you:

- *Add a node A as a matrix of shape [3,2] with some constants*
- *Add a node B as a matrix opened to take variables of type float32 (use placeholder)*
- *Add an operation of their product to the graph producing an output matrix of shape [3,4]*
- *Print the resulting matrix on the screen*

Basic operations

Class exercise 2/2 solution:

```python
import tensorflow as tf

# ----------------
# Create a graph that produces a 3x4 matrix, where you:
#  - Add nodes A and B as matrixes with some floating-point constants
#  - Add an operation of their product to the graph producing an output matrix
#    of shape [3,4]
#  - Print the resulting matrix on the screen

matrix1 = tf.constant([[3.,3.],[4,5],[6,8]]) #3x2

# Create a placeholder so that the product with matrix1 will produces a 3x4 matrix:
matrix2 = tf.placeholder(tf.float32)

# Create a matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value is their product:
product = tf.matmul(matrix1, matrix2)

# The output of the op is returned in 'result' as a numpy `ndarray` object.
with tf.Session() as sess:
    result = sess.run(product, feed_dict={matrix2: ([[2.,6,9,2.],[8,2,2,4]])}) #3x4
    print(result)

# Result:
# [[30. 24. 33. 18.]
# [48. 34. 46. 28.]
# [76. 52. 70. 44.]]
```

TensorFlow 2.0 vs. 1.X

- API Cleanup: what is it?
  - Some of the APIs in v.2.0 and up have changed such as:
    - tf namespace was cleaned up by moving unused packages in more specific subpackages (tf.math, rtc.)
    - tf.app, tf.flags, and tf.logging are removed
    - tf.contrib has some projects taken away from it
    - tf.summary, tf.keras.metrix and tf.keras.optimizers have been replaced with their v.2.0

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Refactor your code into smaller functions: what is it?
  - In TF 1.X selected tensors were evaluated via session.run()
  - In TF 2.0, users need to refactor their code into smaller functions
  - It's not necessary to decorate each of these smaller functions with tf.function
  - Use tf.function to decorate high-level computations:

    Example: one step of training or the forward pass of your model

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Use Keras layers and models to manage variables: what is it?
  - Keras models and layers offer the variables and trainable_variables properties, which recursively gather up all dependent variables
  - This simplifies local variables management to where they are being used
  - Comparison:

conventional

```
14 def dense(x, W, b):
15   return tf.nn.sigmoid(tf.matmul(x, W) + b)
16
17 @tf.function
18 def multilayer_perceptron(x, w0, b0, w1, b1, w2, b2 ...):
19   x = dense(x, w0, b0)
20   x = dense(x, w1, b1)
21   x = dense(x, w2, b2)
22   ...
23
24 # You still have to manage w_i and b_i, and their shapes are defined
25 # far away from the code.
```

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Use Keras layers and models to manage variables: what is it?
  - Keras models and layers offer the variables and trainable_variables properties, which recursively gather up all dependent variables
  - This simplifies local variables management to where they are being used
  - Comparison:

with Keras

```
# Each layer can be called, with a signature equivalent to linear(x)
layers = [tf.keras.layers.Dense(hidden_size, activation=tf.nn.sigmoid) ...
          for _ in range(n)]
perceptron = tf.keras.Sequential(layers)

# layers[3].trainable_variables => returns [w3, b3]
# perceptron.trainable_variables => returns [w0, b0, ...]
```

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Combine tf.data.Datasets and @tf.function:
  - One can use regular Python iteration to iterate over training data (that fits in memory)
  - However, the tf.data.Dataset is the more elegant way to stream training data from disk.
  - Datasets are iterables (not iterators), and work just like other Python iterables in Eager mode
  - Designers can fully utilize dataset async prefetching/streaming features by wrapping their code in tf.function(), which replaces Python iteration with the equivalent graph operations using AutoGraph

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Combine tf.data.Datasets and @tf.function:
  - Comparison:

conventional

```
@tf.function
def train(model, dataset, optimizer):
  for x, y in dataset:
    with tf.GradientTape() as tape:
      # training=True is only needed if there are layers with different
      # behavior during training versus inference (e.g. Dropout).
      prediction = model(x, training=True)
      loss = loss_fn(prediction, y)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

  - When using Keras .fit() API, the designer won't have to worry about dataset iteration:

with Keras

```
model.compile(optimizer=optimizer, loss=loss_fn)
model.fit(dataset)
```

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- tf.metrics aggregates data and tf.summary logs them:
  - Use tf.summary.(scalar|histogram|...) to log summaries
  - Unlike TF 1.x, the summaries are emitted directly to the writer; there is no separate "merge" op and no separate add_summary() call, which means that the step value must be provided at the callsite:

```
49 summary_writer = tf.summary.create_file_writer('/tmp/summaries')
50 with summary_writer.as_default():
51   tf.summary.scalar('loss', 0.1, step=42)
```

  - To aggregate data before logging them as summaries, use tf.metrics (tf.keras.metrics) - accumulating values while returning a cumulative result when calling .result()
  - Point TensorBoard at the summary log directory in order to visualize the generated summaries:

```
53 tensorboard --logdir /tmp/summaries
```

... more on that later

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- No more globals: what is it?
  - TF 1.X relied heavily on global namespaces*
  - Variable scopes, global collections, helper methods like tf.get_global_step(), tf.global_variables_initializer() allowed for recovering variables that the user lost track of (i.e. the Python variable that was pointing to it)
  - All these methods were confusing and were removed in v.2.0
  - Conclusion: Keep track of your variables! … otherwise they will be garbage collected.

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Eager Execution: what is it?
  - Eager execution changes the traditional TensorFlow operations: tf.Tensor objects reference the values instead of symbolic handles to nodes in a computational graph
  - It immediately evaluates and returns the values to Python, viewable through 'print'
  - Eager execution is enabled like this:
    - A) tf.compat.v1.enable_eager_execution() and must be performed before any program execution
    - B) within a Python function using: tf.contrib.eager.py_func()
    - C) testing through:

```
In [7]: tf.executing_eagerly()
Out[7]: True
```

Source: https://www.tensorflow.org/guide/eager

TensorFlow 2.0 vs. 1.X

- Eager Execution: what is it?
  - Fast debugging with immediate run time errors
  - In TensorFlow 1.13 and earlier eager execution mode is not present. Instead the code exists of two phases:

    1. Construction phase – a graph is constructed

    2. Execution phase – a session is run to execute the graph

TensorFlow 2.0 vs. 1.X

- Eager Execution: what is it?
  - Example:
    - Eager execution disabled (default for up to TF v.1.15):

```
In [3]: tf.executing_eagerly()
Out[3]: False

In [4]: a = tf.constant([[34, 56], [87, 12]])

In [5]: print(a)
Tensor("Const:0", shape=(2, 2), dtype=int32)
```

    - Eager execution enabled (default for TF v.2.0 and up):

```
In [10]: tf.compat.v1.enable_eager_execution()

In [11]: tf.executing_eagerly()
Out[11]: True

In [12]: a = tf.constant([[34, 56], [87, 12]])

In [13]: print(a)
tf.Tensor(
[[34 56]
 [87 12]], shape=(2, 2), dtype=int32)
```

try it in class…

Source: https://www.tensorflow.org/guide/eager

TensorFlow 2.0 vs. 1.X

- Normal execution by default:

Example 1a:

In [1]:
```
import tensorflow as tf
import numpy as np

#define the inputs
a = tf.constant(np.array([1., 2., 3.]))
b = tf.constant(np.array([4., 5., 6.]))

c = tf.tensordot(a, b, 1) #computational graph created.

session = tf.Session() #create a session
output = session.run(c) #run the session
print(output)
```

```
32.0
```

TensorFlow 2.0 vs. 1.X

- Eager execution by default:
  - In TensorFlow version 1.15 the eager execution mode is present in the contrib package and needs to be activated.

Example 1b:

In [1]:
```
import tensorflow as tf
tf.enable_eager_execution() #activate eager execution which is present in the contrib package

#define the variables
a = tf.contrib.eager.Variable(1)
b = tf.contrib.eager.Variable(2)

#execute directly
#since eager execution mode is activate there is no need for session
for iteration in range(5):
    a.assign(a + b)
    b.assign(b * 2)

print(a.numpy())
```

```
63
```

TensorFlow 2.0 vs. 1.X

- Eager execution by default
  - In TensorFlow version 2.0 eager execution is enabled by default.
  - Operations are evaluated immediately
  - Code interacts with TensorFlow line by line without the need to define graphs and sessions

Example:

```
In [2]: b = tf.constant(np.array([4., 5., 6.]))
        type(b)

Out[2]: tensorflow.python.framework.ops.EagerTensor
```

Enabled by default

TensorFlow 2.0 vs. 1.X

- Eager execution by default
  - TensorFlow version 2.0 eager execution

Example 2a:

In [1]:
```
import tensorflow as tf
import numpy as np

#define the inputs
a = tf.constant(np.array([1., 2., 3.])) #eager execution mode enabled by default
b = tf.constant(np.array([4., 5., 6.]))

c = tf.tensordot(a, b, 1) #calculate the dot product

#since eager execution mode is enabled by default there is no need to run session
print(c.numpy())
```

```
32.0
```

TensorFlow 2.0 vs. 1.X

- Eager execution by default
  - TensorFlow version 2.0 eager execution

Example 2b:

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

#define the variables
a = tf.Variable(1)
b = tf.Variable(2)

#execute directly with eager execution
for iteration in range(5):
    a.assign(a + b)
    b.assign(b * 2)

print(a.numpy())
```

```
63
```

TensorFlow 2.0 vs. 1.X

- Functions, not sessions: what is it?
  - A session.run() call is like a function call that specifies the inputs and the function to be called. As a result we get back a set of outputs. Example:

```
# TensorFlow 1.X
outputs = session.run(f(placeholder), feed_dict={placeholder: input})

# TensorFlow 2.0
outputs = f(input)
```

  - In TF 2.0, we can decorate a Python function using tf.function() and TF runs it as a single graph!
  - This way TF 2.0 gains all the benefits of a graph mode:
    - Performance: The function can be optimized (node pruning, kernel fusion, etc.)
    - Portability: The function can be exported/reimported (SavedModel 2.0 RFC), allowing users to reuse and share modular TensorFlow functions.

Source: https://www.tensorflow.org/guide/effective_tf2

- Sh

```
[ ] # A simple TensorBoard test 2 v1.15:
```

```
[ ] %load_ext tensorboard
    import tensorflow as tf
```

```
The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the
%tensorflow_version 1.x magic: more info.
```

```
[ ] #pip list
```

```
[ ] # Let's define our input nodes:
    a = tf.constant(5, name="input_a")
    b = tf.constant(3, name="input_b")

    # Defining the next two nodes in our graph:
    c = tf.multiply(a,b, name="mul_c")
    d = tf.add(a,b, name="add_d")

    # This last line defines the final node in our graph:
    e = tf.add(c,d, name="add_e")

    # To run we have to add the two extra lines or run them in the shell:
    sess = tf.compat.v1.Session()
    sess.run(e)

    # To create the graph:
    sess.graph.as_graph_def()
    file_writer = tf.compat.v1.summary.FileWriter('./', sess.graph)

    # We clean up before we exit:
    file_writer.close()
    sess.close()
```

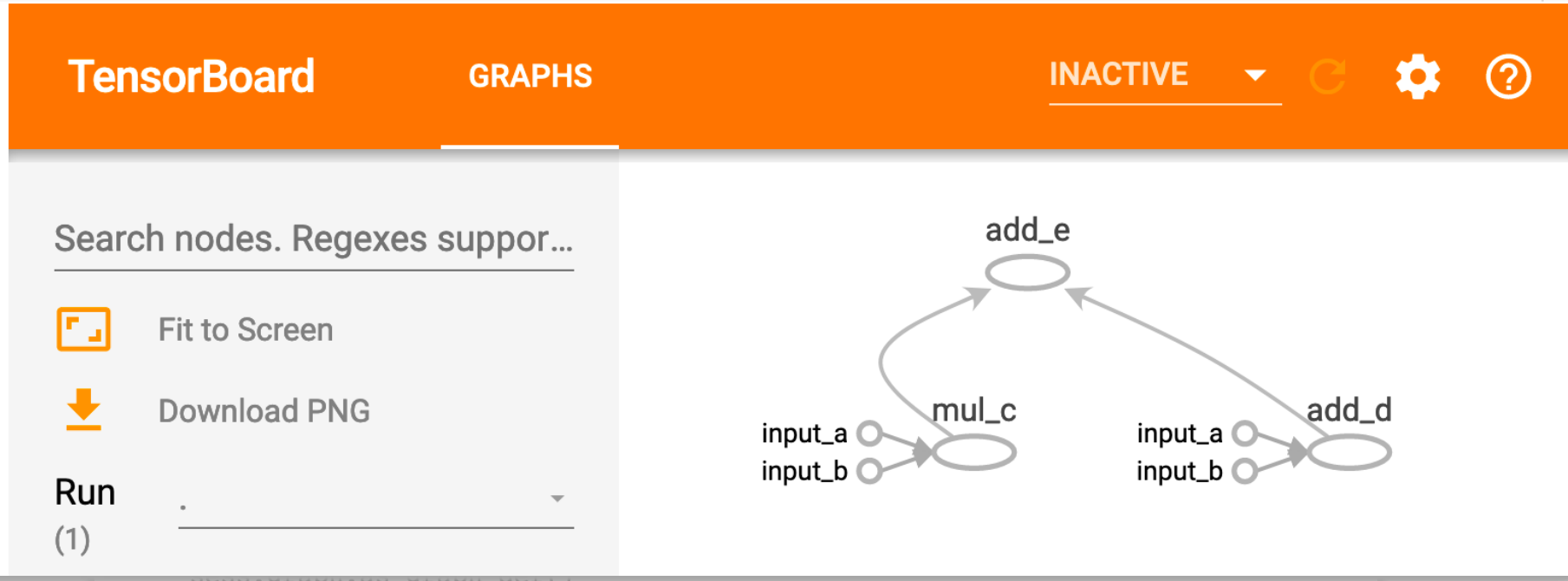
```
[ ]  # A simple TensorBoard test 2 v1.15:
```

```
[ ]  %load_ext tensorboard
     import tensorflow as tf
```

```
[ ]  ls
```

```
events.out.tfevents.1581997922.a9707ab00550  sample_data/
```

```
[ ]  %tensorboard --logdir ./ --event_file events.out.tfevents.1581997922.a9707ab00550
```




```
file_writer = tf.compat.v1.summary.FileWriter('./', sess.graph)

# We clean up before we exit:
file_writer.close()
sess.close()
```

TensorFlow 2.0 vs. 1.X

recall..

- tf.metrics aggregates data and tf.summary logs them:
  - Use tf.summary.(scalar|histogram|...) to log summaries
  - Unlike TF 1.x, the summaries are emitted directly to the writer; there is no separate "merge" op and no separate add_summary() call, which means that the step value must be provided at the callsite:

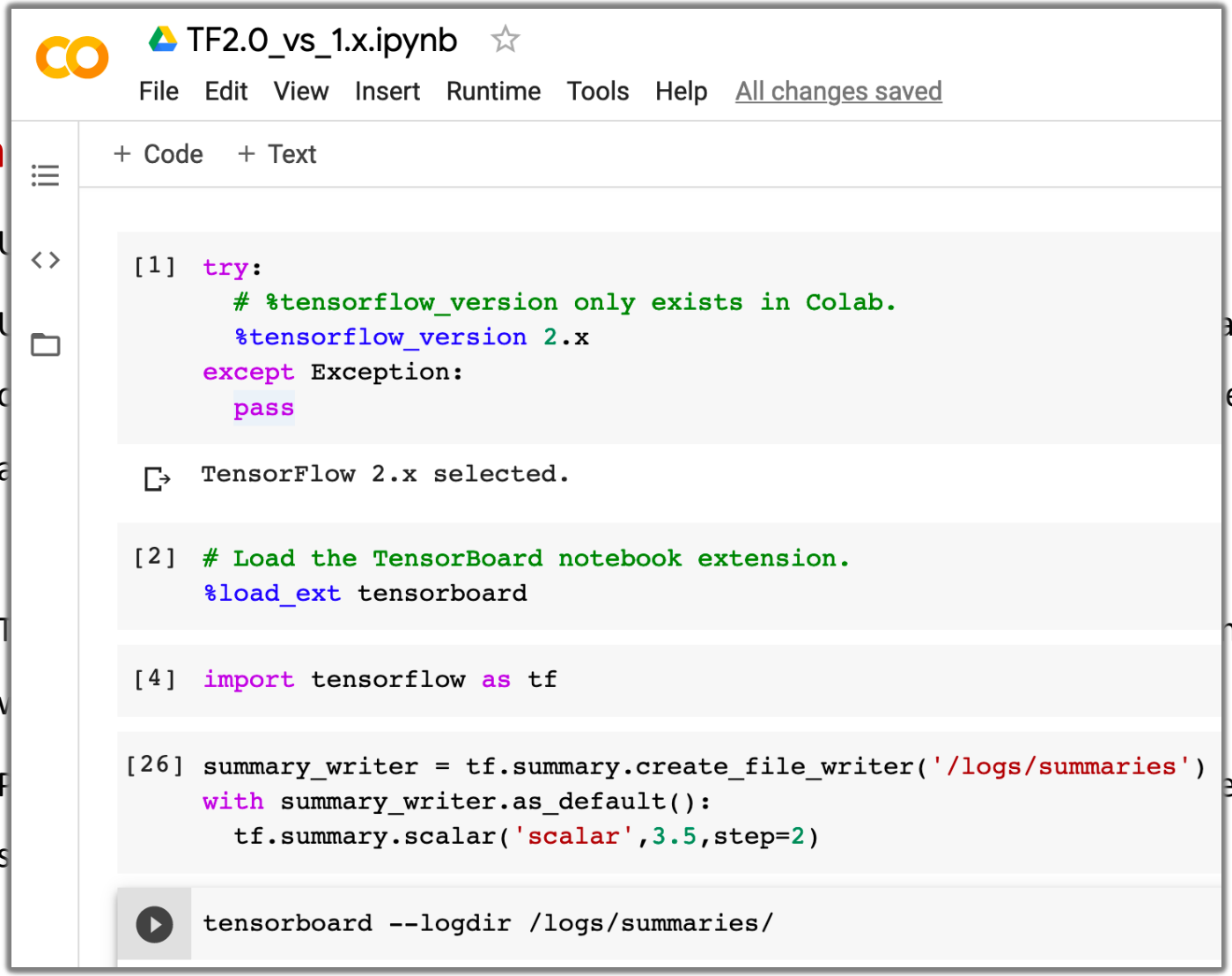
```
49 summary_writer = tf.summary.create_file_writer('/tmp/summaries')
50 with summary_writer.as_default():
51   tf.summary.scalar('loss', 0.1, step=42)
```

  - To aggregate data before logging them as summaries, use tf.metrics - accumulating values while returning a cumulative result when calling .result()
  - Point TensorBoard at the summary log directory in order to visualize the generated summaries:

```
53 tensorboard --logdir /tmp/summaries
```

TensorFlow 2.0 vs. 1.X

- tf.m
  - U
  - U
  - T
  - P



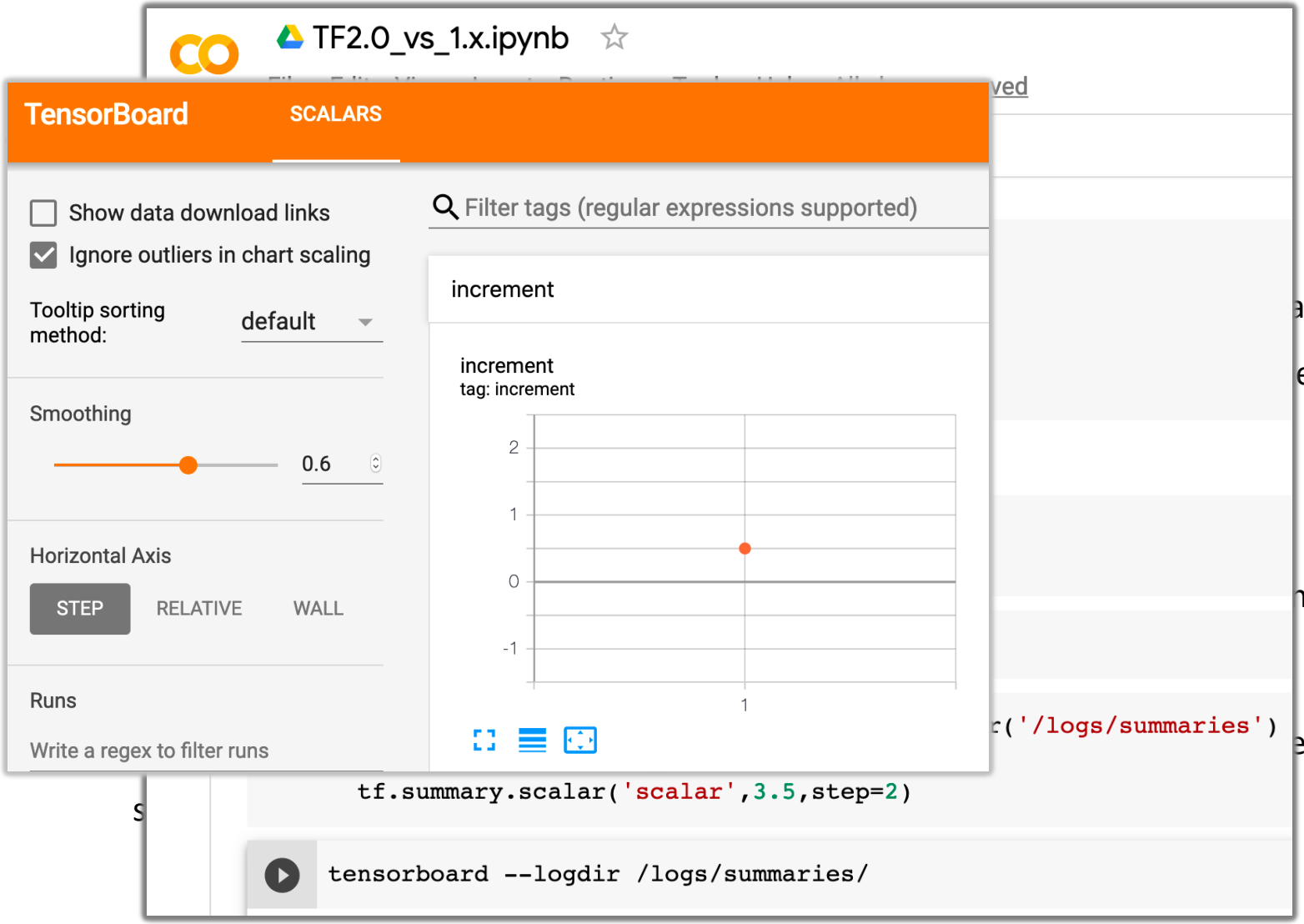
ate "merge"

e provided

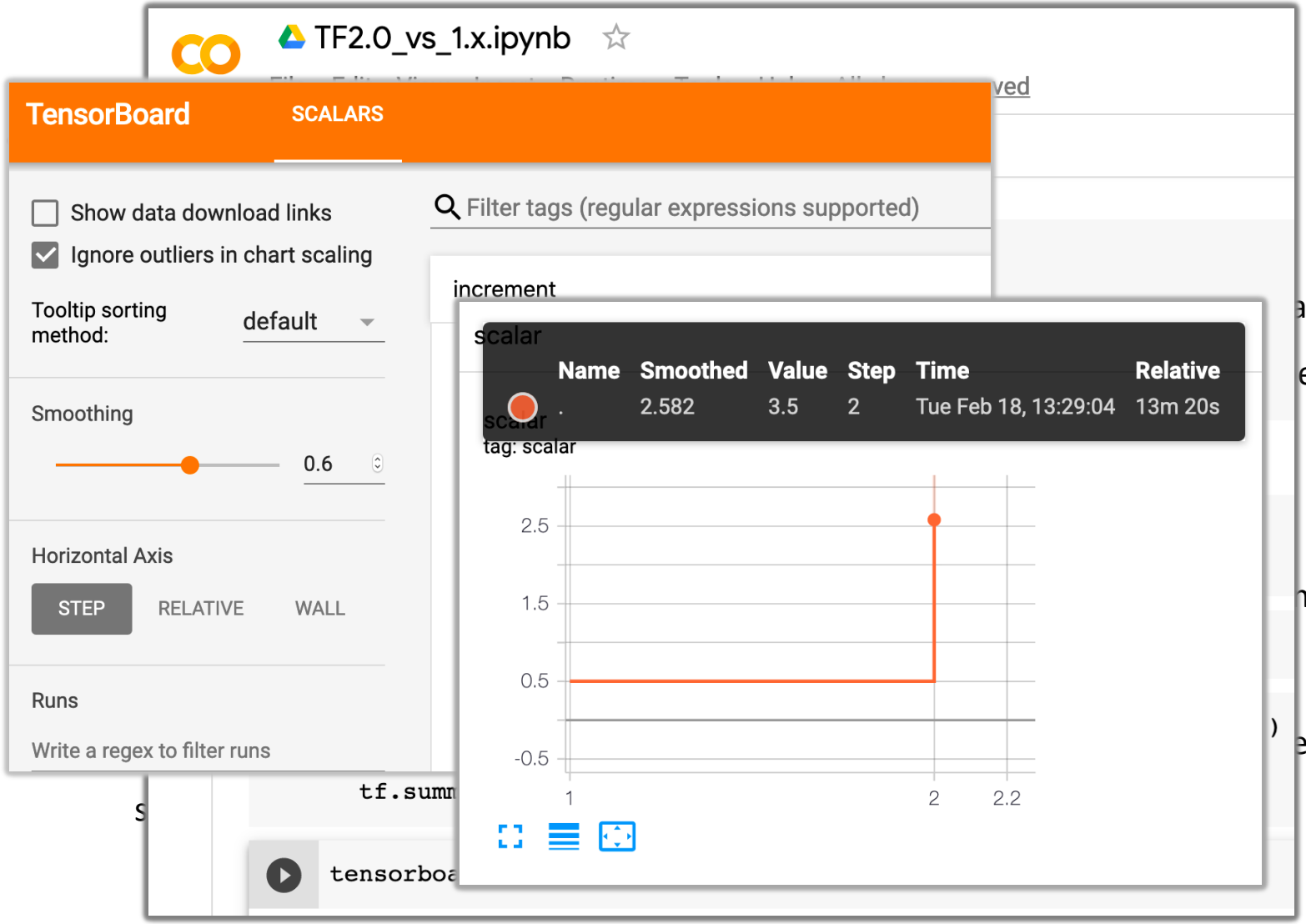
ng values

ed

TensorFlow 2.0 vs. 1.X



TensorFlow 2.0 vs. 1.X



ate "merge"

e provided

ng values

ed

TensorFlow 2.0 vs. 1.X

- Use tf.config.experimental_run_functions_eagerly() when debugging:
  - In TF 2.0, Eager execution lets you run the code step-by-step to inspect shapes, data types and values
  - Some APIs (tf.function, tf.keras, etc.) are designed to use Graph execution, for performance and portability
  - When debugging, use tf.config.experimental_run_functions_eagerly(True) to use Eager execution inside this code:

```
56 @tf.function
57 def f(x):
58   if x > 0:
59     import pdb
60     pdb.set_trace()
61     x = x + 1
62   return x
```

  - Then call like this:

```
In [1]: tf.config.experimental_run_functions_eagerly(True)
In [2]: f(tf.constant(1))
```

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Use tf.config.experimental_run_functions_eagerly() when debugging:
  - In TF 2.0, Eager execution lets you run the code step-by-step to inspect shapes, data types and values
  - Some APIs ion, for performance and portab
  - When debu ue) to use Eager execution i

```
>>> f()
-> x = x + 1
(Pdb) l
  6      @tf.function
  7      def f(x):
  8        if x > 0:
  9          import pdb
 10          pdb.set_trace()
 11  ->      x = x + 1
 12        return x
 13
 14      tf.config.experimental_run_functions_eagerly(True)
 15      f(tf.constant(1))
[EOF]
```

  - Then call like this:

```
In [1]: tf.config.experimental_run_functions_eagerly(True)
In [2]: f(tf.constant(1))
```

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Advantage of AutoGraph with Python control flow:
  - AutoGraph provides a way to convert data-dependent control flow into graph-mode equivalents like tf.cond and tf.while_loop
  - One common place where data-dependent control flow appears is in sequence models. tf.keras.layers.RNN wraps an RNN cell, allowing you to either statically or dynamically unroll the recurrence
  - For a more detailed overview of AutoGraph's features go to:
    https://www.tensorflow.org/guide/effective_tf2

Source: https://www.tensorflow.org/guide/effective_tf2

TensorFlow 2.0 vs. 1.X

- Tf.function and Autograph
  - Graphs and sessions makes TensorFlow more efficient and faster.
  - Autograph is in the contrib package and is used to convert a function into TensorFlow funct.

Example 1c (TensorFlow 1.15):

In [1]:
```
import tensorflow as tf
tf.enable_eager_execution() #activate eager execution which is present in contrib package
from tensorflow.contrib import autograph #import autograph from contrib package to use graph mode

#function to find the next number in fibonacci series
def fib1(a):
    x, y = 0, 1
    for i in range(a-1):
        x, y = y, x + y
    return y

tf_fib1 = autograph.to_graph(fib1) #using autograph we coonvert the function to a tensorflow function

graph = tf.Graph() #create a graph
with graph.as_default():
    #within the graph we call the fib1 function so it runs in graph mode
    answer = tf_fib1(tf.constant(8)) #only created the operations required to perform the computation and adds them to the graph

with tf.Session(graph=graph) as sess: #start a session to evaluate the result in tensor and print
    print(answer.eval())
```

```
21
```

TensorFlow 2.0 vs. 1.X

- Tf.function and Autograph

Example 1d (TensorFlow 1.15):

In [1]:
```
import tensorflow as tf
tf.enable_eager_execution() #activate eager execution mode
from tensorflow.contrib import autograph #import autograph from contrib package

#define the inputs
a = 2.0
b = 8.0

#function to find the geometric mean
def geomean(a, b):
    gmean = tf.sqrt(a * b)
    return gmean

tf_geomean = autograph.to_graph(geomean) #autograph converts the function to a tensflow function

graph = tf.Graph() #create a graph
with graph.as_default():
    #call the geomean fucntion so it runs in graph mode
    answer = tf_geomean(a, b) #only creates the operations to perform the computation and add them to graph

with tf.Session(graph=graph) as sess:#start a session to evaluate the result and print
    print(answer.eval())
```

```
4.0
```

TensorFlow 2.0 vs. 1.X

- Tf.function and Autograph
  - In TensorFlow 2.0, a function can be decorated using tf.function, which compiles and runs it as a graph
  - Using this all benefits of graph mode can be gained such as performance, optimization, portability, etc.
  - Unlike sessions there is no need to initialize variables
  - Autograph is available in tf.function, this makes it easy to run tensorflow computations
  - When a python function is initiated with tf.function, it automatically converts the python code to tensorflow graph code
  - The code is then compiled into a graph and executed when the function is called

TensorFlow 2.0 vs. 1.X

- Tf.function and Autograph
  - TensorFlow 2.0 tf.function

Example 2c:

In [2]:
```
import tensorflow as tf

@tf.function #using tf.function, the function is compiled and executed as a graph
def fib1(a): #function to find the next term in fibonacci series
    x, y = 0, 1
    for i in range(a-1):
        x, y = y, x + y
    return y

result = fib1(tf.constant(8)) #compiles the code into a graph and executes it
tf.print(result)
```

```
21
```

TensorFlow 2.0 vs. 1.X

- Tf.function and Autograph
  - TensorFlow 2.0 tf.function

Example 2d:

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

#define the inputs
a = 2.0
b = 8.0

@tf.function #decorate the function using tf.function, which compiles and executes it as a graph
def geomean(a, b): #function to find the geometric mean
    gmean = tf.sqrt(a * b)
    return gmean

gmean = geomean(a, b) #compiles the code into a graph and executes it
tf.print(gmean)
```

```
4
```

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of Optimizer to minimize data loss:
    - TensorFlow can use a gradient descent algorithm to minimize a loss function in a certain way. In TensorFlow we can access a GradientDescentOptimizer as part of tf.train.
    - In TensorFlow version 1.14.0 we use this command to minimize data loss:

      optimizer = tf.train.GradientDescentOptimizer

      train = optimizer.minimize(error) → here we are minimizing is the error.
    - In TensorFlow version 2.0.0 Keras library initializes the optimizer to minimize the loss. We use below command:

      train = tf.keras.optimizers.SGD(learning_rate=0.1)

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of Optimizer:

Example 1e: TensorFlow 1.14

```
In [4]: #Y= bx+c

In [5]: np.random.rand(2)
Out[5]: array([0.48502309, 0.37827261])

In [6]: b = tf.Variable(0.48)
        c = tf.Variable(0.37)

In [7]: #Cost Function

        error = 0

        for x,y in zip(x_data,y_label):

            y_pred = b*x + c

            error += (y-y_pred)

In [8]: # In V1.14.0 Below function is used to declare optimization to minimze data loss #
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
        train = optimizer.minimize(error)
```

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of Optimizer:

Example 2e: TensorFlow 2.0

```
In [11]: #Y= mx+b

In [22]: np.random.rand(2)

Out[22]: array([0.62112104, 0.0966464 ])

In [24]: m = tf.Variable(0.62)
         b = tf.Variable(0.09)

In [25]: # cost function
         loss = 0
         for x,y in zip(x_data,y_label):
             y_pred = m*x+b
             loss += (y-y_pred)**2

In [26]:  #In V2.0.0 we use keras library to initalize optimizer
         tf.function
         from tensorflow import keras
         train = tf.keras.optimizers.SGD(learning_rate=0.1)
         def loss(y, y_pred):
             return tf.reduce_mean(tf.square(y-y_pred))
```

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of global variables:
    - In TensorFlow before execution all global variables are required to be initialized
    - Everything except of variables do not require initialization (constants and placeholders)
    - But every used variable (even if it is a constant) must be initialized
    - In TensorFlow V1.14.0 global variables are initialized as below alias:
      init = tf.global_variables_initializer()
    - In TensorFlow version V2.0.0 all global variables are declared as:
      tf.initialize_all_variables

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of global variables:

    Example 1f: Tensorflow 1.14

```
In [9]: # In V1.14.0 variables are initialized using below command #
        init = tf.global_variables_initializer()
```

all global variables are initialized before execution

```
In [10]: with tf.Session() as sess:
             sess.run(init)

             epochs = 10

             for i in range(epochs):
                 sess.run(train)

             final_slope, final_intercept = sess.run([b,c])
```

TensorFlow 2.0 vs. 1.X

- Flexibility in Linear model
  - Initialization of global variables:

    Example 2f: Tensorflow 2.0

```
In [21]: # Below command used for variable initailization throws error in V2.0.0
         init = tf.global_variables_initializer()
```

This line in v.1.14.0 will throw an ERROR in v.2.0.0

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-515360f499c3> in <module>
----> 1 init = tf.global_variables_initializer()

AttributeError: module 'tensorflow' has no attribute 'global_variables_initializer'
```

```
# In V2.0.0 below command is used to initialize variable:
tf.initialize_all_variables
```

TensorFlow 2.0 vs. 1.X

- Tensorflow datasets:
  - Datasets are TensorFlow is used for machine learning
  - Datasets are used to work with NN models in a more elegant way
  - To download DataSet in V1.14.0: Below command is used:

    tf_flowers = tfds.load(name="tf_flowers", split="train")
  - To download DataSet in V2.0.0 we use this command:

    cifar10_builder.download_and_prepare()

TensorFlow 2.0 vs. 1.X

- Tensorflow datasets:
  - Downloading data set “tf_flowers” in v.1.14.0

Example 1g:

In [18]:
```
tf.compat.v1.enable_eager_execution()
```

In [19]:
```
print(tfds.list_builders())
```

```
['abstract_reasoning', 'aflw2k3d', 'amazon_us_reviews', 'bair_robot_pushing_small', 'bigearthnet', 'binarized_mnist', 'binary_alpha_digits', 'caltech101', 'caltech_birds2010', 'caltech_birds2011', 'cats_vs_dogs', 'celeb_a', 'celeb_a_hq', 'chexpert', 'cifar10', 'cifar100', 'cifar10_corrupted', 'clevr', 'cnn_dailymail', 'coco', 'coco2014', 'coil100', 'colorectal_histology', 'colorectal_histology_large', 'curated_breast_imaging_ddsm', 'cycle_gan', 'deep_weeds', 'definite_pronoun_resolution', 'diabetic_retinopathy_detection', 'downsampled_imagenet', 'dsprites', 'dtd', 'dummy_dataset_shared_generator', 'dummy_mnist', 'emnist', 'eurosat', 'fashion_mnist', 'flores', 'food101', 'gap', 'glue', 'groove', 'higgs', 'horses_or_humans', 'image_label_folder', 'imagenet2012', 'imagenet2012_corrupted', 'imdb_reviews', 'iris', 'kitti', 'kmnist', 'lfw', 'lm1b', 'lsun', 'mnist', 'mnist_corrupted', 'moving_mnist', 'multi_nli', 'nsynth', 'omniglot', 'open_images_v4', 'oxford_flowers102', 'oxford_iiit_pet', 'para_crawl', 'patch_camelyon', 'pet_finder', 'quickdraw_bitmap', 'resisc45', 'rock_paper_scissors', 'rock_you', 'scene_parse150', 'shapes3d', 'smallnorb', 'snli', 'so2sat', 'squad', 'stanford_dogs', 'stanford_online_products', 'starcraft_video', 'sun397', 'super_glue', 'svhn_cropped', 'ted_hrlr_translate', 'ted_multi_translate', 'tf_flowers', 'titanic', 'trivia_qa', 'uc_merced', 'ucf101', 'visual_domain_decathlon', 'voc2007', 'wikipedia', 'wmt14_translate', 'wmt15_translate', 'wmt16_translate', 'wmt17_translate', 'wmt18_translate', 'wmt19_translate', 'wmt_t2t_translate', 'wmt_translate', 'xnli']
```

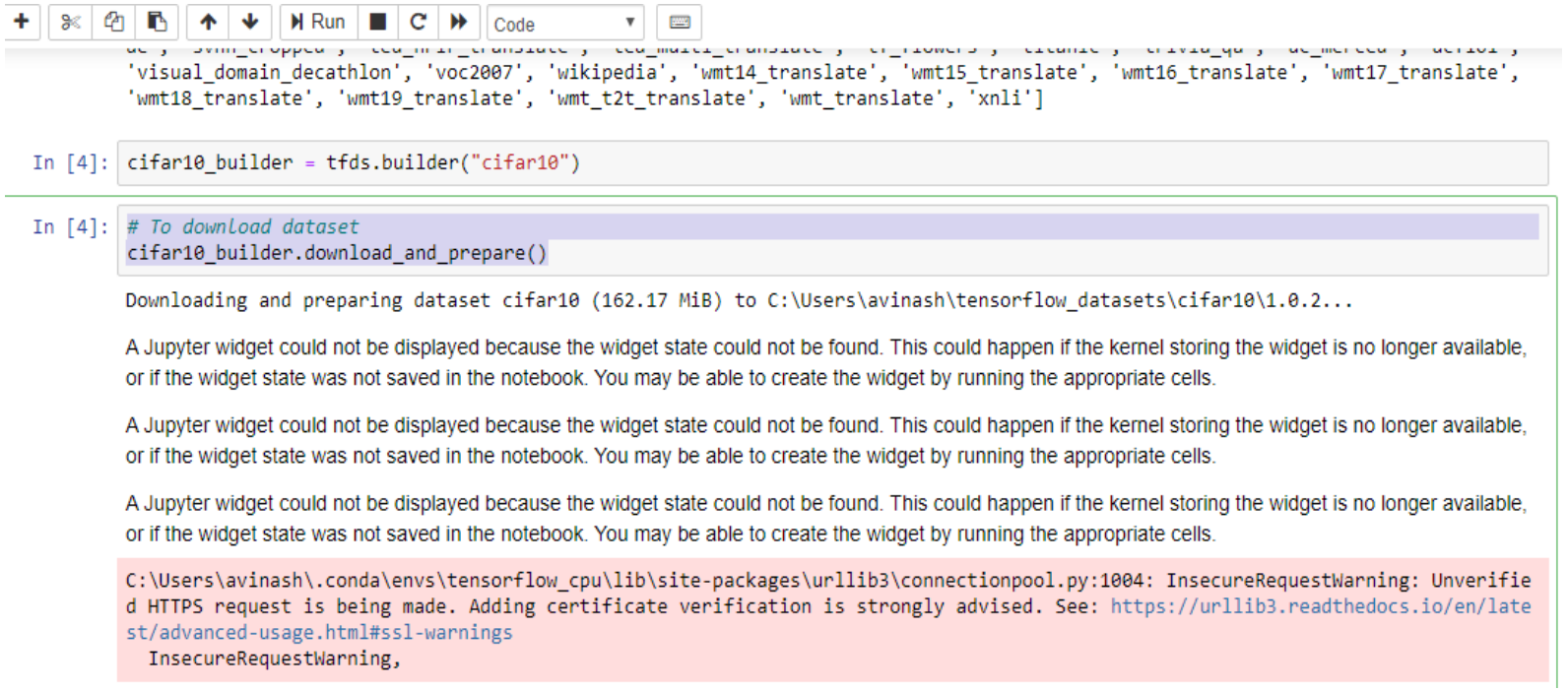
In [20]:
```
# we use below command to download data set in V1.14.0
tf_flowers = tfds.load(name="tf_flowers", split="train") #----------> split and train data set
assert isinstance(tf_flowers, tf.data.Dataset)
print(tf_flowers)
```

```
WARNING:absl:Warning: Setting shuffle_files=True because split=TRAIN and shuffle_files=None. This behavior will be deprecated on 2019-08-06, at which point shuffle_files=False will be the default for all splits.

<DatasetV1Adapter shapes: {image: (?, ?, 3), label: ()}, types: {image: tf.uint8, label: tf.int64}>
```

TensorFlow 2.0 vs. 1.X

- Tensorflow datasets:
  - Downloading data set “cifar10” in v.2.0.0

    Example 2g:



TensorFlow 2.0 vs. 1.X

- Tensorflow datasets:

To split and train data the following command is used in v.1.14.0

```
tf_flowers = tfds.load(name="tf_flowers", split="train")
assert isinstance(tf_flowers, tf.data.Dataset)
print(tf_flowers)
```

Example 1g:

In [6]:
```
# to split and train data below command is used in V1.14.0
train_ds, test_ds = tfds.load('horses_or_humans', split=['train', 'test'])
```

```
WARNING:absl:Warning: Setting shuffle_files=True because split=TRAIN and shuffle_files=None. This behavior will be deprecated o
n 2019-08-06, at which point shuffle_files=False will be the default for all splits.
```

In [7]:
```
# If we will use below split and test command which is used in V2.0.0 then it will thrownerror
ds_train, ds_test = horses_or_humans_builder.as_dataset(split=[tfds.Split.TRAIN, tfds.Split.TEST])
```

!!!

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-6556afde61b1> in <module>
----> 1 ds_train, ds_test = horses_or_humans_builder.as_dataset(split=[tfds.Split.TRAIN, tfds.Split.TEST])

AttributeError: 'dict' object has no attribute 'as_dataset'
```

TensorFlow 2.0 vs. 1.X

- Tensorflow datasets:
  - To split and train data below command is used in V2.0.0

    ds_train, ds_test = cifar10_builder.as_dataset(split=[tfds.Split.TRAIN, tfds.Split.TEST])

Example 2g:

In [5]:
```
# to split and train data set
ds_train, ds_test = cifar10_builder.as_dataset(split=[tfds.Split.TRAIN, tfds.Split.TEST])
```

```
WARNING:absl:Warning: Setting shuffle_files=True because split=TRAIN and shuffle_files=None. This behavior will be deprecated o
n 2019-08-06, at which point shuffle_files=False will be the default for all splits.
```

In [6]:
```
ds_train = ds_train.batch(10)
```

In [7]:
```
for features in ds_train:
    image, label = features["image"], features["label"]
```

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Is integrated with TensorFlow 2.0 and is easily called
  - It can be used to visualize the graph and other tools to understand, debug, and optimize our model *
  - Provides a built-in performance dashboard **
  - Tensorflow backend switches to tf.keras in TensorFlow 2.0.
  - tf.keras is better maintained and has better integration with TensorFlow features (eager execution, distribution support and other)

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0
    - First layer has input_shape defined as [28x28] matrix
    - Dense layer has activation function tf.nn.relu, and numbers of neurons
    - Fully connected layer is using softmax function

Example 1a:

KERAS AND TENSORBOARD

```
[ ]  1  #Keras is default API in 2.0
     2  #keras has built in datasets, we are using one such dataset 'fashion_mnist'
     3  fashion_mnist = tf.keras.datasets.fashion_mnist
     4  (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

```
[ ]  1  #Simple AI model in a sequential way
     2  import datetime
     3  model = tf.keras.Sequential([
     4                               tf.keras.layers.Flatten(input_shape=(28,28)),   #first lay
     5                               tf.keras.layers.Dense(128, activation=tf.nn.relu),  #dense
     6                               tf.keras.layers.Dense(10, activation=tf.nn.softmax) #fully
```

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

    Example 1a:

```
11  model.compile(optimizer = 'adam',                      #using adam optimizer
12                loss = 'sparse_categorical_crossentropy',  #specify loss which means here we a
13                metrics = ['accuracy'])
14
15  log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
16  tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
17
18  model.fit(train_images, train_labels, epochs = 5, callbacks = [tensorboard_callback])
19
20  %tensorboard --logdir logs/fit
```

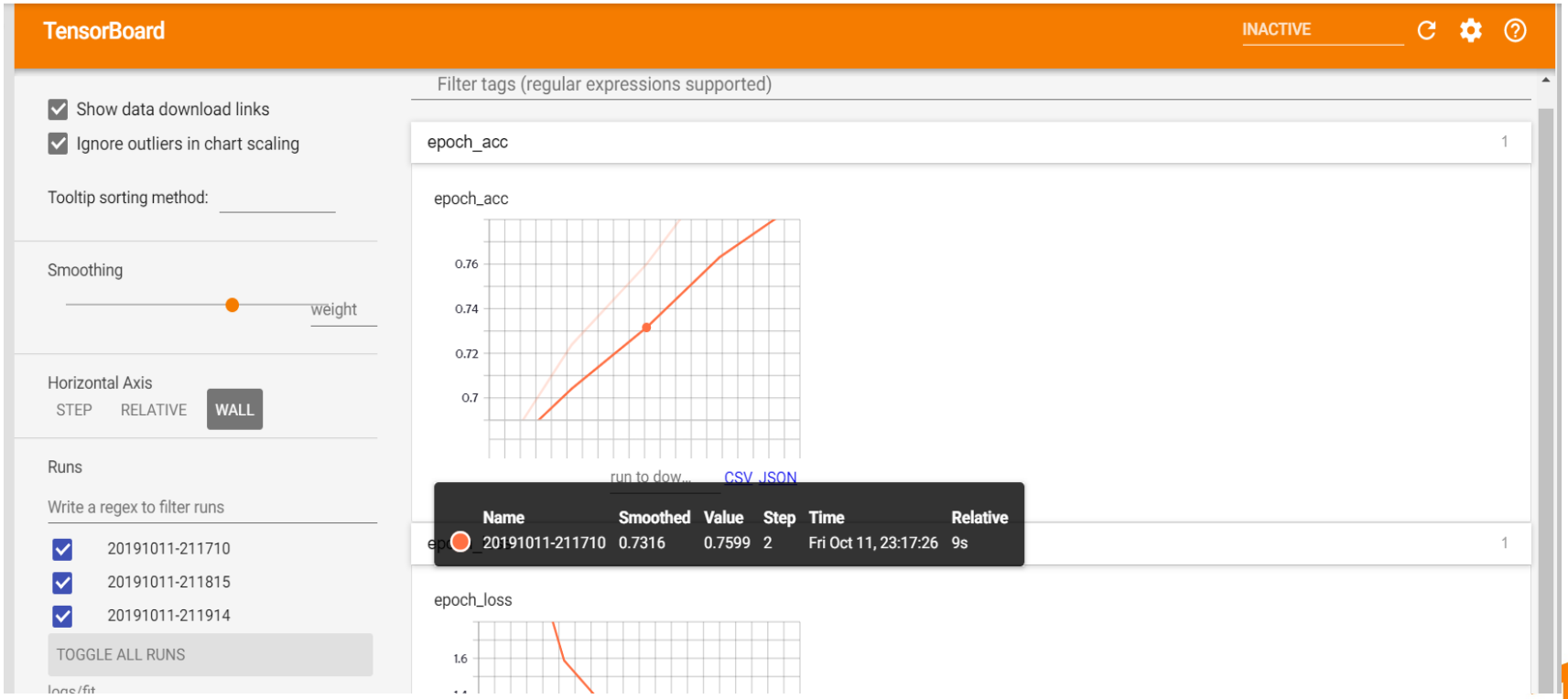
```
Epoch 1/5
60000/60000 [==============================] - 5s 90us/sample - loss: 3.2293 - accuracy: 0.6823
Epoch 2/5
60000/60000 [==============================] - 6s 92us/sample - loss: 0.7243 - accuracy: 0.7272
Epoch 3/5
60000/60000 [==============================] - 5s 91us/sample - loss: 0.6108 - accuracy: 0.7724
Epoch 4/5
60000/60000 [==============================] - 5s 91us/sample - loss: 0.5569 - accuracy: 0.8019
Epoch 5/5
60000/60000 [==============================] - 5s 91us/sample - loss: 0.5250 - accuracy: 0.8126
```

Accuracy : 81.26%

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

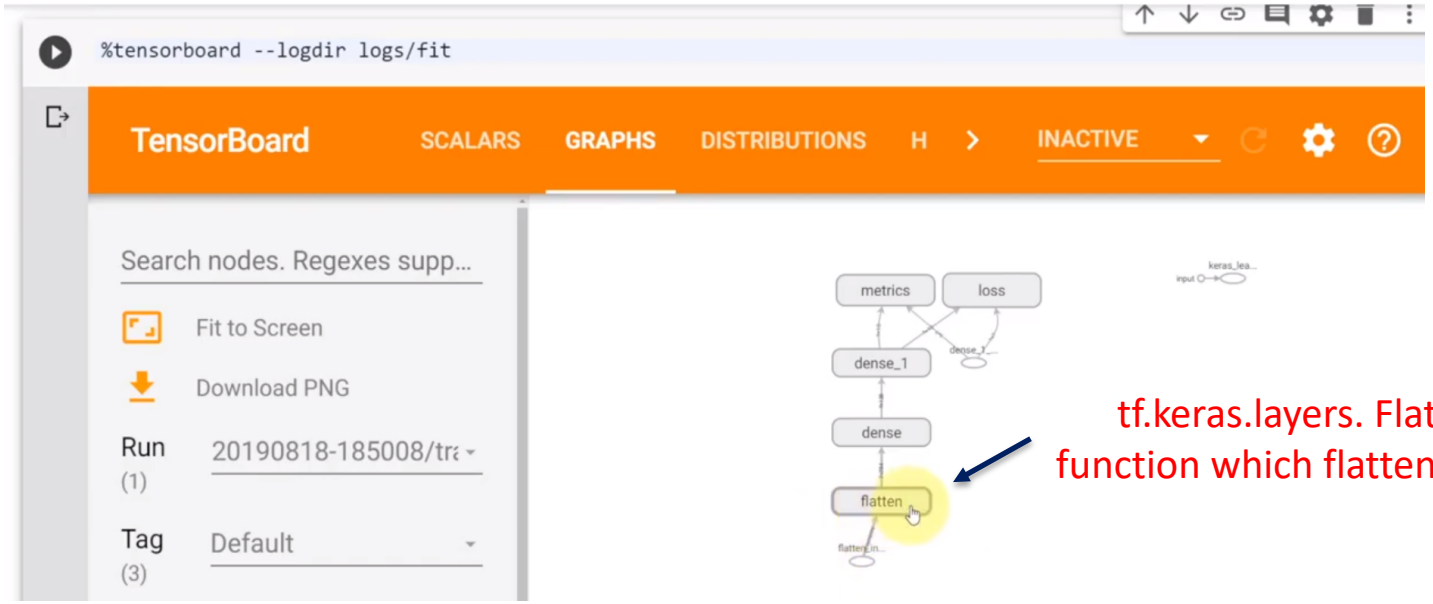
    Example 1a:



TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

    Example 1a:



tf.keras.layers. Flatten is a function which flattens the input

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 1.x

Example 1b:

```
[ ]  1  #Simple AI model in a sequential way
     2  import datetime
     3  from keras.callbacks import TensorBoard
     4  fashion_mnist = tf.keras.datasets.fashion_mnist
     5  (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
     6  model = tf.keras.Sequential([
     7                               tf.keras.layers.Flatten(input_shape=(28,28)),   #first layer
     8                               tf.keras.layers.Dense(128, activation=tf.nn.relu),  #dense la
     9                               tf.keras.layers.Dense(10, activation=tf.nn.softmax) #fully co
    10                               # model.add(Flatten(input_shape(28,28))),
    11                               # model.add(Dense(128, activation=relu)),
    12                               # model.add(Dense(10, activation=softmax))
    13  ])
    14  model.compile(optimizer = 'adam',                      #using adam optimizer
    15                loss = 'sparse_categorical_crossentropy',  #specify loss which means here w
    16                metrics = ['accuracy'])
    17
    18  log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    19  tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
    20
    21  model.fit(train_images, train_labels, epochs = 5, callbacks = [tensorboard_callback])
    22
    23  %load_ext tensorboard
    24  %tensorboard --logdir {logs_base_dir}
    25  %tensorboard --logdir logs/fit
```

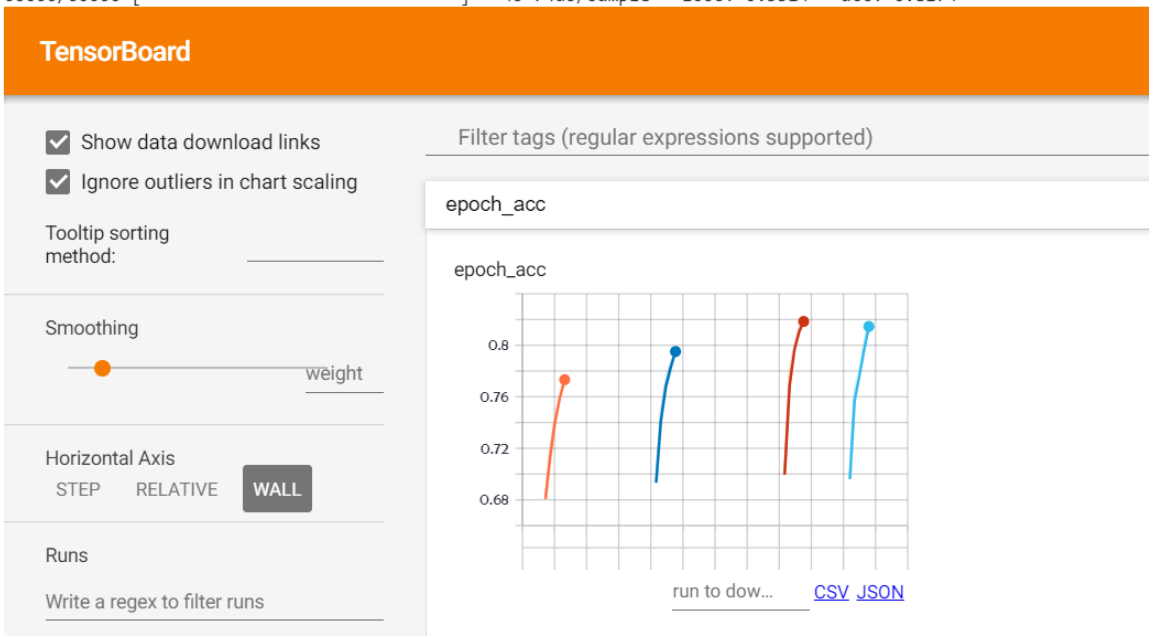
Tensorboard needs to be imported in 1.x and is not inbuilt as in 2.0

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 1.x

    Example 1b:

```
] Epoch 3/5
↳ 60000/60000 [==============================] - 4s 74us/sample - loss: 0.6420 - acc: 0.7791
  Epoch 4/5
  60000/60000 [==============================] - 4s 73us/sample - loss: 0.5749 - acc: 0.7999
  Epoch 5/5
  60000/60000 [==============================] - 4s 74us/sample - loss: 0.5314 - acc: 0.8174
```



TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

Example 2a:

```
[ ]
import tensorflow as tf
#Create a simple tensorflow graph
with tf.name_scope("My_Graph"):            #add scope as a prefix to operations , making three seperate Scopes i.e Scope_1, Scope_2 and Scope_3
  with tf.name_scope("Scope_1"):
    a = tf.add(1, 2, name = "addition_1")          # We are adding two constants '1' and '2' and putting in 'a'
    b = tf.multiply(a, 3, name = "multiplying_1")  # we are using 'a' which has already been defined and a constant '3', multiply function and will be stored in 'b'
  with tf.name_scope("Scope_2"):
    c = tf.add(4, 5, name = "addition_2")            # We are adding two constants '4' and '5' and putting in 'c'
    d = tf.multiply(c, 6, name = "multiplying_2")    # we are using 'c' which has already been defined and a constant '6', multiply function and will be stored in 'd'
with tf.name_scope("Scope_3"):
  e = tf.multiply(4, 5)
  f = tf.div(c, 6, name = "division")
```

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

Example 2a:

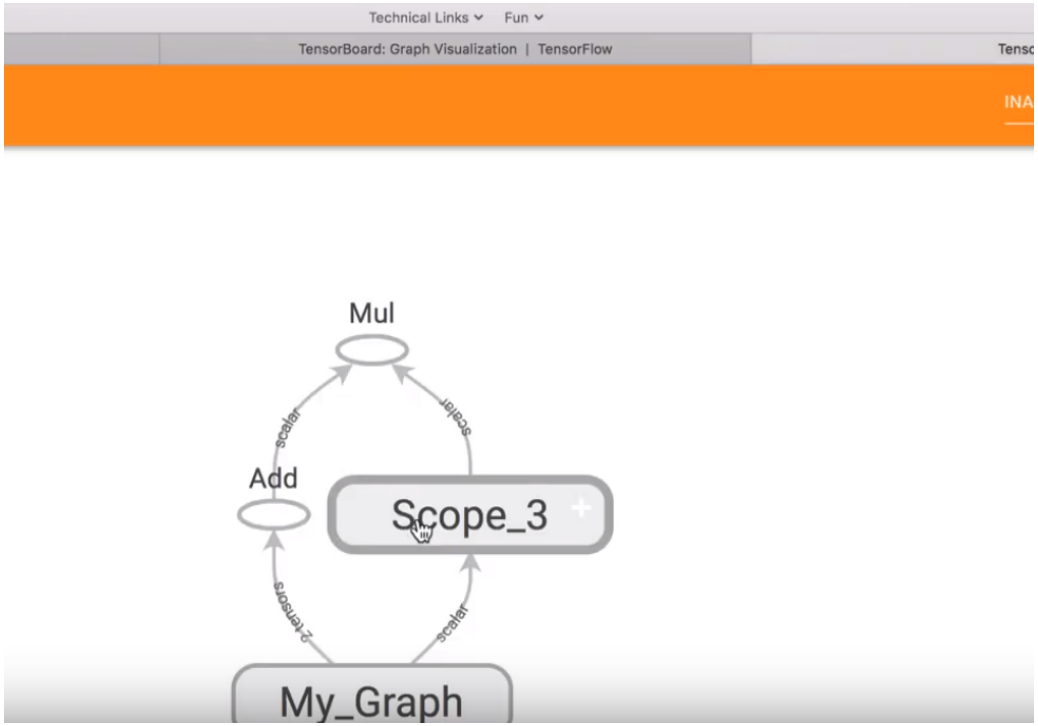
```
[ ]  1  #pip install tensorboardcolab
     2  from tensorboardcolab import *
     3  tbc = TensorBoardColab() # To create a tensorboardcolab object it will automatically create the URL
     4  writer = tbc.get_writer() # To create a FileWriter
     5  writer.add_graph(tf.get_default_graph()) # add the graph
     6  writer.flush()    #Forces to send buffered data to storage
```

```
Wait for 8 seconds...
TensorBoard link:
https://9d8efdea.ngrok.io
```

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

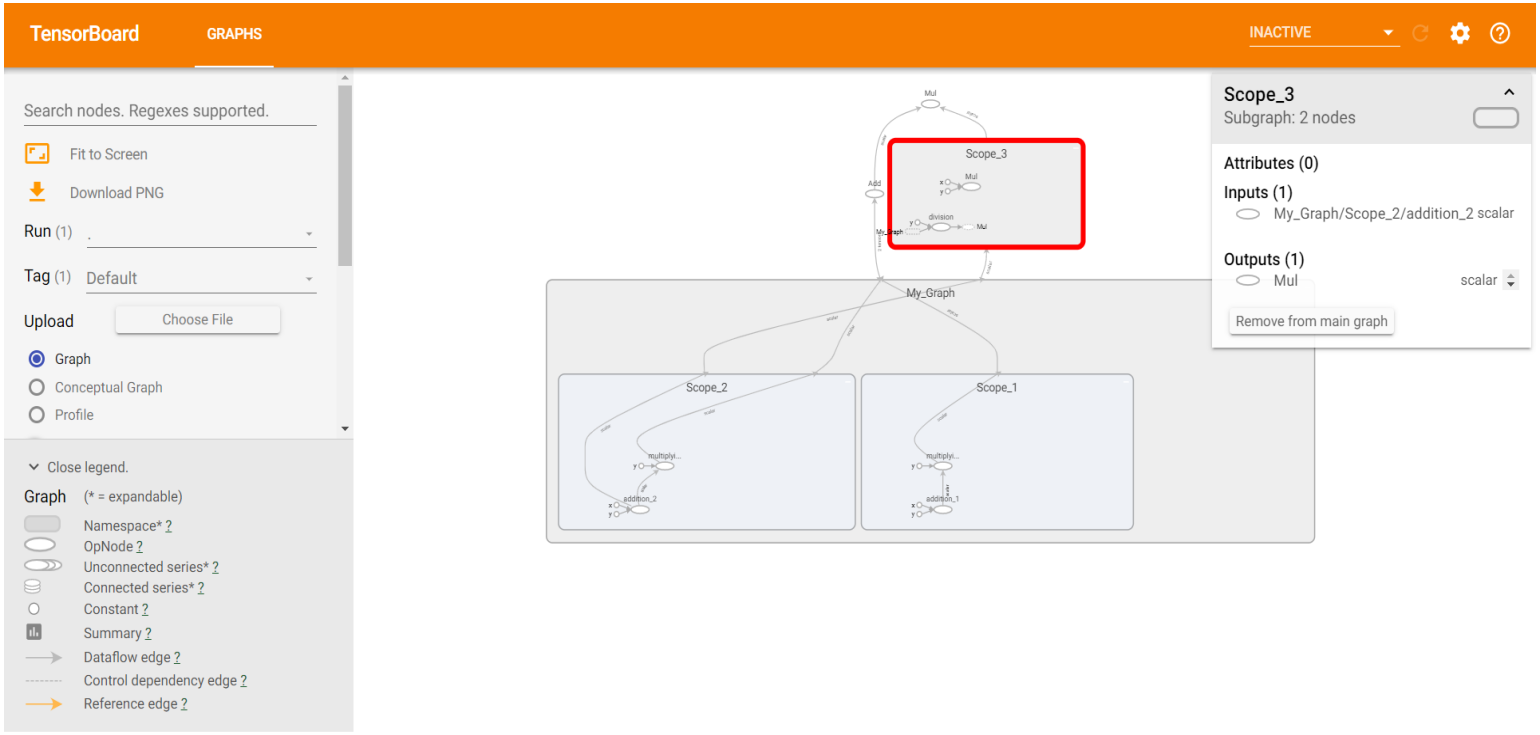
    Example 2a:



TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 2.0

Example 2a:



TensorFlow 2.0 vs. 1.X

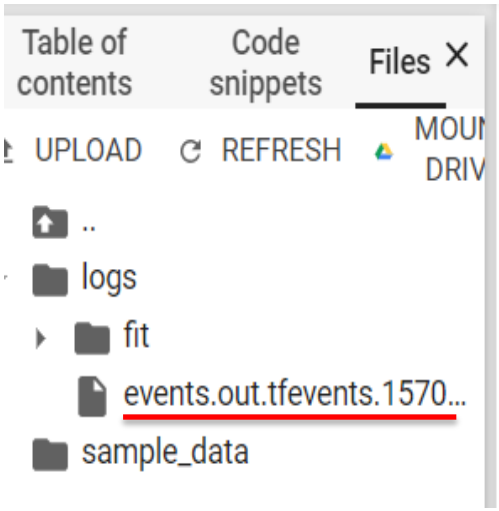
- Tensorboard
  - Tensorflow 1.x
  - Execution phase will have sessions - used to run the code and get results

Example 2b:

TensorFlow 2.0 vs. 1.X

- Tensorboard
  - Tensorflow 1.x

Example 2b:




```
    b = tf.multiply(a, 3, name = "multiplying_1")
  with tf.name_scope("Scope_2"):
    c = tf.add(4, 5, name = "addition_2")
    d = tf.multiply(c, 6, name = "multiplying_2")
with tf.name_scope("Scope_3"):
  e = tf.multiply(4, 5)
  f = tf.div(c, 6, name = "division")

g = tf.add(b, d)
h = tf.multiply(g, f)
```

TensorFlow 2.0 vs. 1.X

- Distributed Strategy
  - The tf.distribute.Strategy API is integrated in Tensorflow 2.0
  - It provides an abstraction for distributing the training across multiple processing units
  - Main goal is to allow users to enable distributed training using existing models and training code, with minimal changes
  - Which makes us develop out model first and then decide how we want to run it : over multiple GPUs or even TPUs
  - This will dramatically improve computational efficiencies

TensorFlow 2.0 vs. 1.X

- Distributed Strategy
  - Tensorflow 2.0

DISTRIBUTED STRATEGY

```
import tensorflow as tf
import datetime

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

#Distributed Strategy is an Tensorflow API which helps us develp model once and decide how to run it over multiple GPUs or TPUs
#this will dramatically improve computational efficiency
strategy = tf.distribute.MirroredStrategy()  #Define strategy : creates one replica per GPU device. Each variable in the model is mirrored across all its copies.

with strategy.scope():
  model = tf.keras.Sequential([
                        tf.keras.layers.Flatten(input_shape=(28,28)),   #first layer with input shape = 28 pixels
                        tf.keras.layers.Dense(128, activation=tf.nn.relu),  #dense layer with 128 neurons and using rectified linewar units fuction
                        tf.keras.layers.Dense(10, activation=tf.nn.softmax) #fully connected neural network with 10 classes and softmax( probablity within each class
                        # model.add(Flatten(input_shape(28,28))),
                        # model.add(Dense(128, activation=relu)),
                        # model.add(Dense(10, activation=softmax))
  ])
  model.compile(optimizer = 'adam',                     #using adam optimizer
                loss = 'sparse_categorical_crossentropy',  #specify loss which means here we are defining different catagories of class. Also can use binary_crossentropy fo
                metrics = ['accuracy'])

log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

model.fit(train_images, train_labels, epochs = 25, callbacks = [tensorboard_callback])
```

TensorFlow 2.0 vs. 1.X

- Distributed Strategy
  - Tensorflow 2.0

```
[ ] Epoch 10/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4782 - acc: 0.8413
Epoch 11/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4781 - acc: 0.8426
Epoch 12/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4781 - acc: 0.8430
Epoch 13/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4713 - acc: 0.8453
Epoch 14/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4659 - acc: 0.8460
Epoch 15/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4608 - acc: 0.8473
Epoch 16/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4657 - acc: 0.8467
Epoch 17/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4582 - acc: 0.8470
Epoch 18/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4624 - acc: 0.8484
Epoch 19/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4583 - acc: 0.8495
Epoch 20/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4551 - acc: 0.8495
Epoch 21/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4512 - acc: 0.8491
Epoch 22/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4521 - acc: 0.8490
Epoch 23/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4504 - acc: 0.8513
Epoch 24/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4542 - acc: 0.8499
Epoch 25/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4482 - acc: 0.8501
<tensorflow.python.keras.callbacks.History at 0x7f82a36ef4a8>
```

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed
  - TensorFlow v.1 module contains the complete API with it's original semantics
  - TensorFlow will upgrade the script and convert it to version 2
  - It directly determines the equivalent behaviour and renames v1.arg_max to tf.argmax
  - Variable is created with v1.get_variable

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed

Before converting Tensorflow 1.14:

- Create variable with v1.get_variable
- Assess collected data implicitly with different methods
- The v1 layers module is used to contain layer functions that are relied on v1.variable scope to define and re-use the variables
- Variable is created with v1.get_variable
- V1.losses.get_regularization_loss
- Initializing variables manually
- Use tf.data datasets for data input, these objects are integrated with tensorflow
- Accessing collections explicitly

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed
  - Tensorflow 1.14

Example 1k:

```
in_a = tf.placeholder(dtype=tf.float32, shape=(2))
in_b = tf.placeholder(dtype=tf.float32, shape=(2))

def forward(x):
  with tf.variable_scope("matmul", reuse=tf.AUTO_REUSE):
    W = tf.get_variable("W", initializer=tf.ones(shape=(2,2)),
                        regularizer=tf.contrib.layers.l2_regularizer(0.04))
    b = tf.get_variable("b", initializer=tf.zeros(shape=(2)))
    return W * x + b

out_a = forward(in_a)
out_b = forward(in_b)

reg_loss = tf.losses.get_regularization_loss(scope="matmul")

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  outs = sess.run([out_a, out_b, reg_loss],
                  feed_dict={in_a: [1, 0], in_b: [0, 1]})
```

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed

After converting to TensorFlow 2.0:

- The function will be in forward position still defines the calculation
- Forward is replaced with session.run
- For performance tf.function decorator can be added
- Without referring to any global collections the regularization are calculated
- Use tf.variable for state they are always usable from both content and this tensor's behave differently

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed
  - Tensorflow 2.0

Example 2k:

```
in_a = tf.placeholder(dtype=tf.float32, shape=(2))
in_b = tf.placeholder(dtype=tf.float32, shape=(2))

def forward(x):
  with tf.variable_scope("matmul", reuse=tf.AUTO_REUSE):
    W = tf.get_variable("W", initializer=tf.ones(shape=(2,2)),
                        regularizer=tf.contrib.layers.l2_regularizer(0.04))
    b = tf.get_variable("b", initializer=tf.zeros(shape=(2)))
    return W * x + b

W = tf.Variable(tf.ones(shape=(2,2)), name="W")
b = tf.Variable(tf.zeros(shape=(2)), name="b")

@tf.function
def forward(x):
  return W * x + b

out_a = forward([1,0])
print(out_a)
```

TensorFlow 2.0 vs. 1.X

- Tensorflow variable_scope no longer needed

  No more placeholders:

  - Use tf.data datasets for data input
  - These objects are efficient, expressive and integrate well with TensorFlow

```
tf.Tensor(
[[1. 0.]
 [1. 0.]], shape=(2, 2), dtype=float32)
```

```
out_b = forward([0,1])

regularizer = tf.keras.regularizers.l2(0.04)
reg_loss = regularizer(W)
```

TensorFlow 1.X name scopes

- Real world models can contain dozens or hundreds of nodes, as well as millions of parameters.
- In order to manage this level of complexity, TensorFlow currently offers a mechanism to help organize your graphs with:

  name scopes

- They are incredibly simple to use and provide great value when visualizing your graph with TensorBoard.

TensorFlow 1.X name scopes

- Name scopes allow you to group Operations into larger, named blocks.
- When you launch your graph with TensorBoard, each name scope will encapsulate its own Ops, making the visualization much more digestible.
- For basic name scope usage, simply add your Operations in a with tf.name_scope(<name>) block (… see next slide)

TensorFlow 1.X name scopes

- Example:

```
import tensorflow as tf

with tf.name_scope("Scope_A"):
    a = tf.add(1, 2, name="A_add")
    b = tf.mul(a, 3, name="A_mul")

with tf.name_scope("Scope_B"):
    c = tf.add(4, 5, name="B_add")
    d = tf.mul(c, 6, name="B_mul")

e = tf.add(b, d, name="output")
```

→

```
# Creating aliases:
tf.mul = tf.multiply
tf.sub = tf.subtract
```

- To see the result of these name scopes in TensorBoard, let's open up a FileWriter and write this graph to disk.

```
writer = tf.train.SummaryWriter('./name_scope_1',
graph=tf.get_default_graph())
writer.close()
```

older

```
tf.train.SummaryWriter = tf.summary.FileWriter
```

1.15

```
tf.compat.v1.summary.FileWriter
```

TensorFlow 1.X name scopes

- Because the tf.summary.FileWriter exports the graph immediately, we can simply start up TensorBoard after running the code on the previous slide.

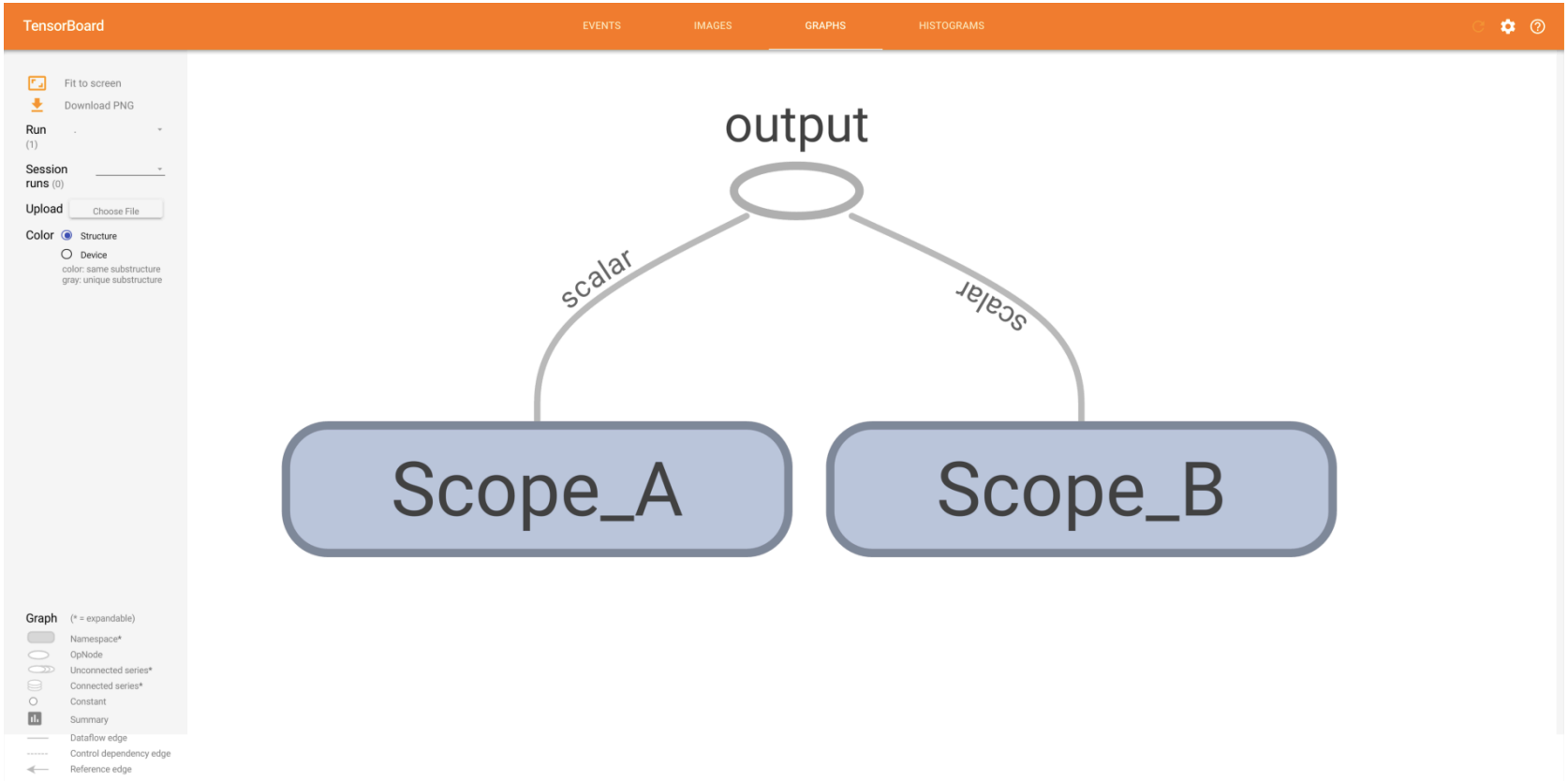
- Navigate to where you ran the previous script and start up TensorBoard:

```
$ tensorboard --logdir='./name_scope_1'
```

- This will start a TensorBoard server on your local computer at port 6006.
- Open a browser and enter localhost:6006 into the URL bar

TensorFlow 1.X name scopes

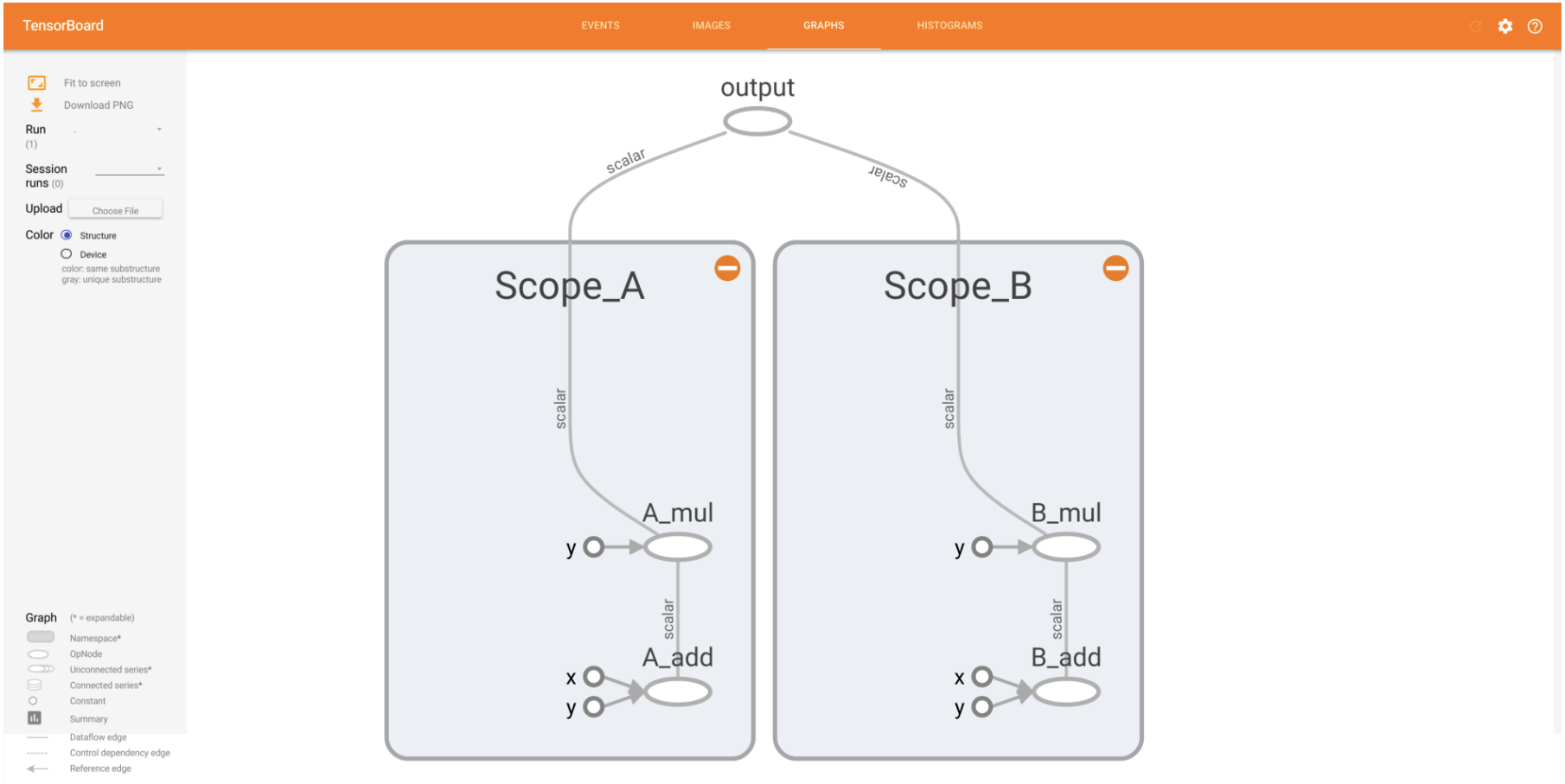
- Navigate to the "Graph" tab, and you'll see something similar to this:



TensorFlow 1.X name scopes

- You'll notice that the add and mul Operations we added to the graph aren't immediately visible- instead, we see their enclosing name scopes.
- You can expand the name scope boxes by clicking on the plus + icon in their upper right corner.
- Inside of each scope, you'll see the individual Operations you've added to the graph

TensorFlow 1.X name scopes



You can nest name scopes within other name scopes

```
graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(tf.float32, shape=[], name="input_a")
    in_2 = tf.placeholder(tf.float32, shape=[], name="input_b")
    const = tf.constant(3, dtype=tf.float32, name="static_val-
ue")

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
            A_mul = tf.mul(in_1, const)
            A_out = tf.sub(A_mul, in_1)

        with tf.name_scope("B"):
            B_mul = tf.mul(in_2, const)
            B_out = tf.sub(B_mul, in_2)

        with tf.name_scope("C"):
            C_div = tf.div(A_out, B_out)
            C_out = tf.add(C_div, const)

        with tf.name_scope("D"):
            D_div = tf.div(B_out, A_out)
            D_out = tf.add(D_div, const)

    out = tf.maximum(C_out, D_out)

writer = tf.train.SummaryWriter('./name_scope_2', graph=graph)
writer.close()
```

`tf.summary.SummaryWriter()`

TensorFlow 1.X name scopes

- To mix things up, this code explicitly creates a tf.Graph object instead of using the default graph.

- Let's look at the code and focus on the name scopes to see exactly how it's structured:

```
graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(...)
    in_2 = tf.placeholder(...)
    const = tf.constant(...)

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
```

TensorFlow 1.X name scopes

```
        # Takes in_1, outputs some value
        ...

    with tf.name_scope("B"):
        # Takes in_2, outputs some value
        ...

    with tf.name_scope("C"):
        # Takes the output of A and B, outputs some value
        ...

    with tf.name_scope("D"):
        # Takes the output of A and B, outputs some value
        ...

# Takes the output of C and D
out = tf.maximum(...)
```

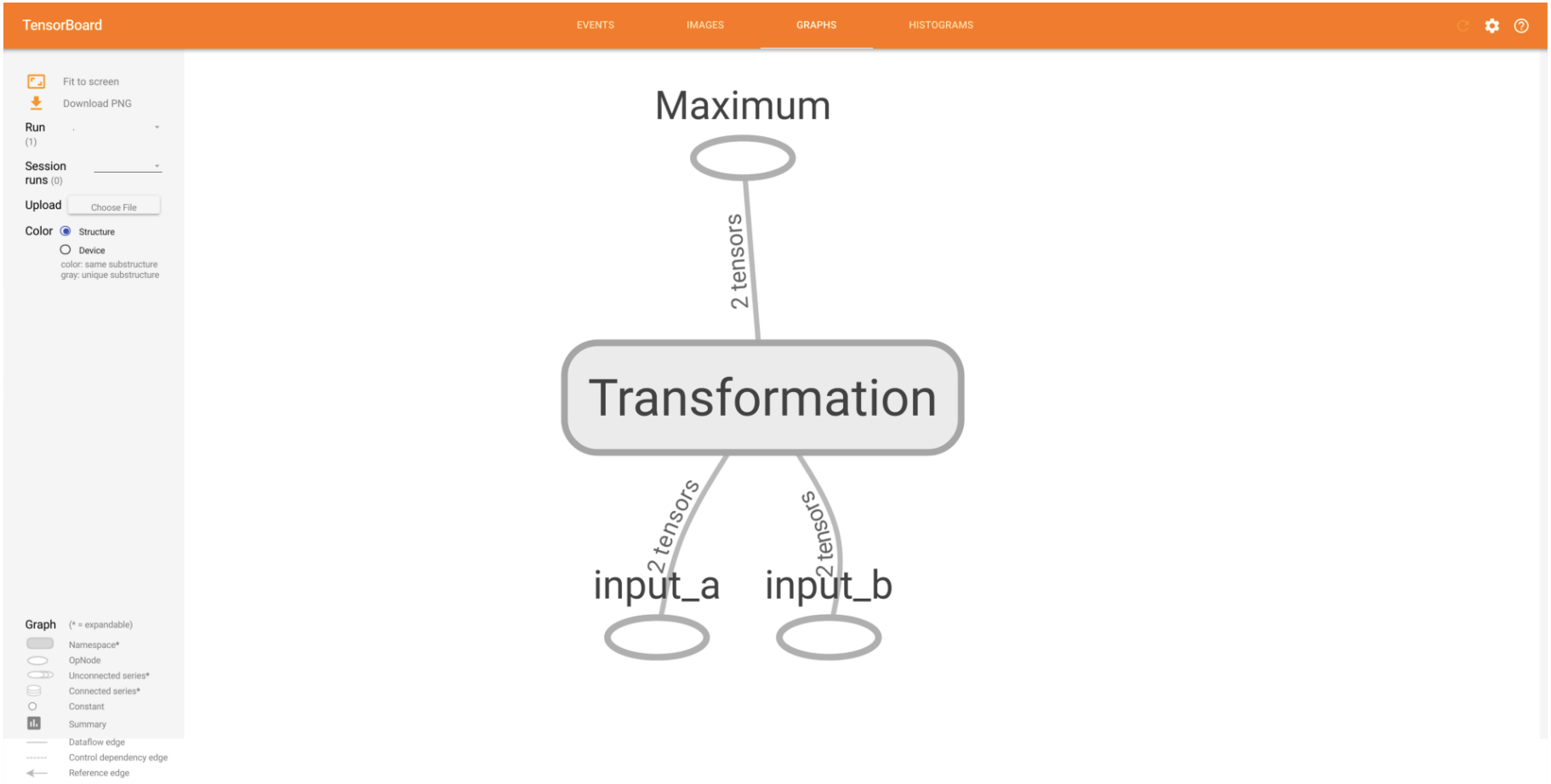
TensorFlow 1.X name scopes

- This model has:
  - two scalar placeholder nodes as input
  - a TensorFlow constant
  - a middle chunk called "Transformation", and
  - a final output node that uses tf.maximum() as its Operation.

- We can see this high-level overview inside of TensorBoard:

```
# Start up TensorBoard in a terminal, loading in our previous graph
$ tensorboard --logdir='./name_scope_2'
```

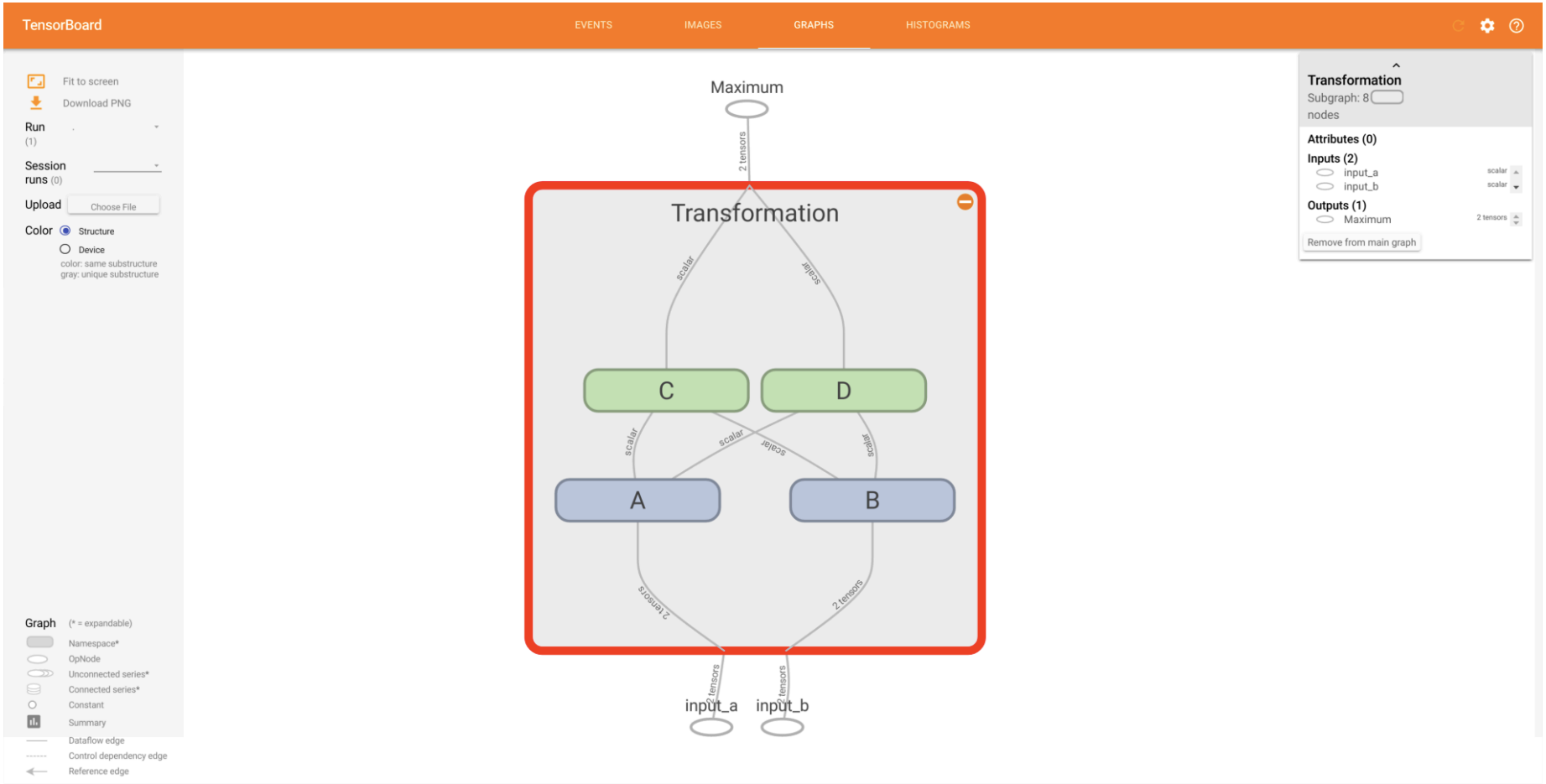
TensorFlow 1.X name scopes



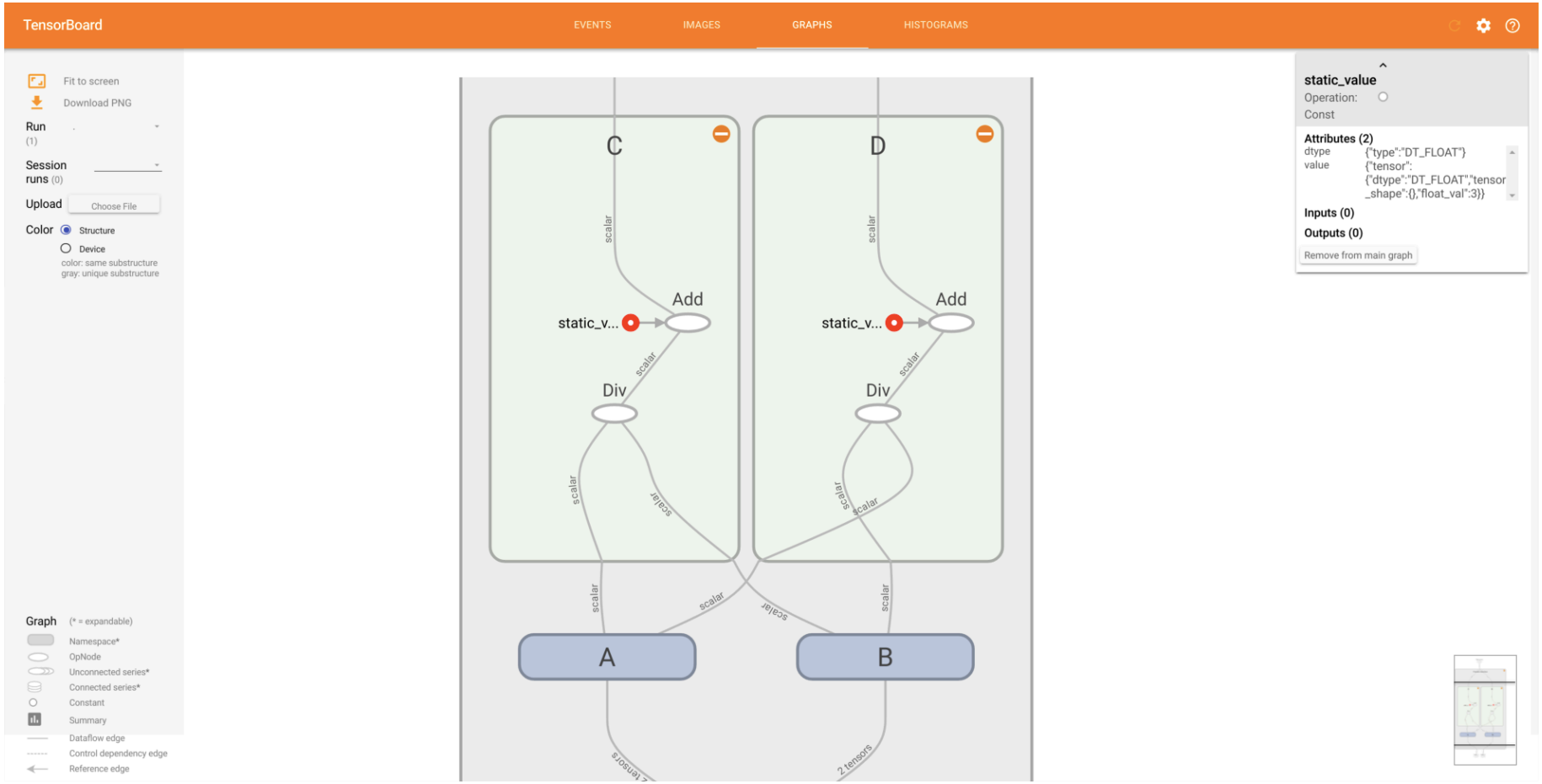
TensorFlow 1.X name scopes

- Inside of the Transformation name scope are four more name scopes arranged in two "layers".
- The first layer is comprised of scopes "A" and "B", which pass their output values into the next layer of "C" and "D".
- The final node then uses the outputs from this last layer as its input.
- If you expand the Transformation name scope in TensorBoard, you'll get a look the one shown on next slide

TensorFlow 1.X name scopes

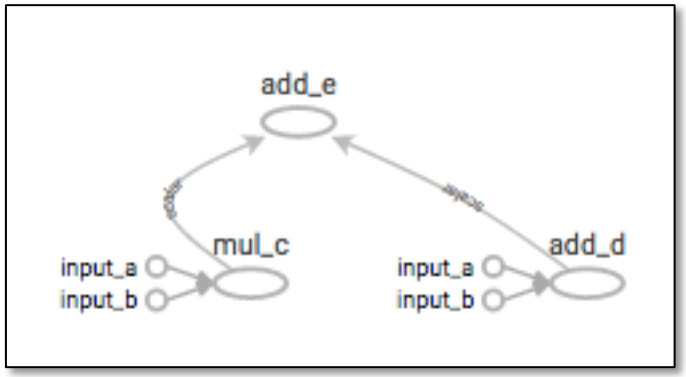


TensorFlow 1.X name scopes



TensorFlow 1.X

- Below is how we can reconstruct our graph from Lecture 2:

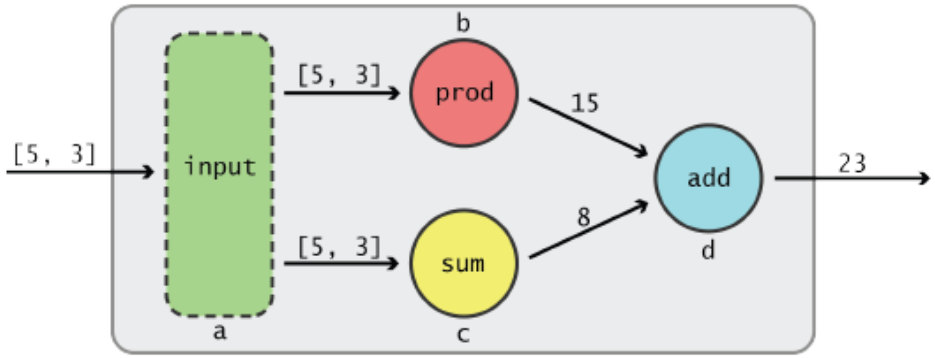


basic_graph.py

```python
# Import the tensorflow library, and reference it as `tf`
import tensorflow as tf

# Build our graph nodes, starting from the inputs
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
c = tf.multiply(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
e = tf.add(c,d, name="add_e")

# Open up a TensorFlow Session
sess = tf.Session()

# Execute our output node, using our Session
sess.run(e)

# Open a TensorFlow FileWriter to write our graph to disk
writer = tf.summary.FileWriter('./my_graph', sess.graph)

# Close our FileWriter and Session objects
writer.close()
sess.close()
```

TensorFlow 1.X

- Here is a new implementation with a new graph:




```
 3 # First we need to import TensorFlow:
 4 import tensorflow as tf
 5
 6 # Defining our single input node:
 7 a = tf.constant([5,3], name="input_a")
 8
 9 # Defining nobe 'b':
10 b = tf.reduce_prod(a, name="prod_b")
11
12 # Defining the next two nodes in our graph:
13 c = tf.reduce_sum(a, name="sum_c")
14
15 # This last line defines the final node in our graph:
16 d = tf.add(b,c, name="add_d")
17
18 # To run we have to add the two extra lines or run them in the shell:
19 sess = tf.Session()
20 sess.run(d)
```

TensorFlow 1.X

- We made few main changes here:

1. We replaced the separate nodes a and b with a consolidated input node (now just a).

2. We passed in a list of numbers, which tf.constant is able to convert to a 1-D Tensor

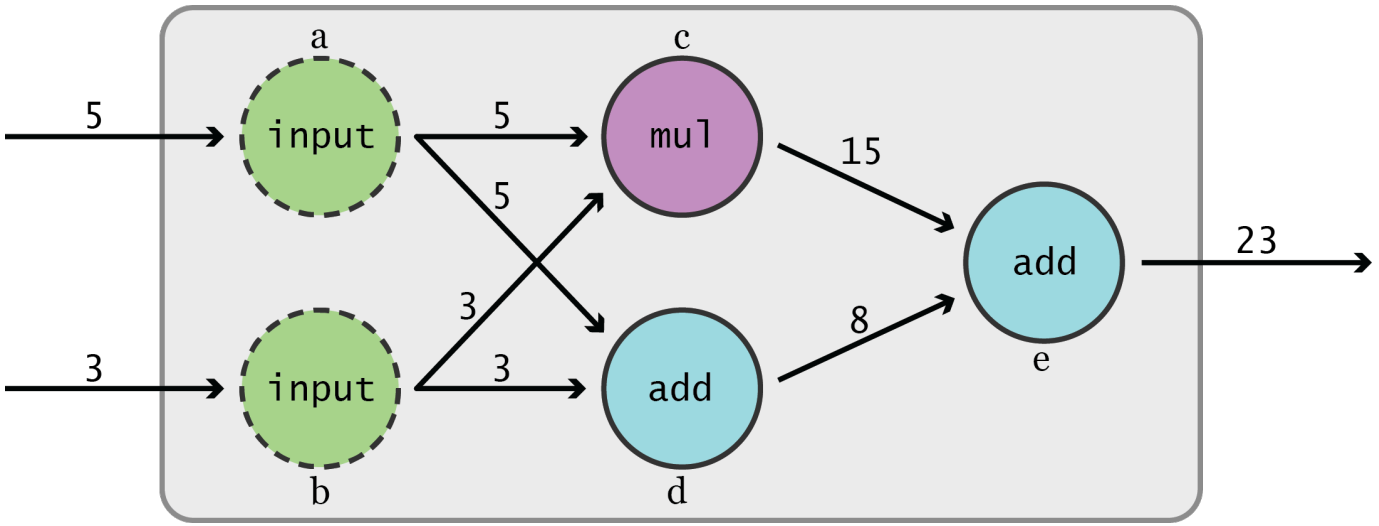
3. Our multiplication and addition Operations, which used to take in scalar values, are now tf.reduce_prod() and tf.reduce_sum()

4. These functions, when just given a Tensor as input, take all of its values and either multiply or sum them up, respectively

TensorFlow name scopes

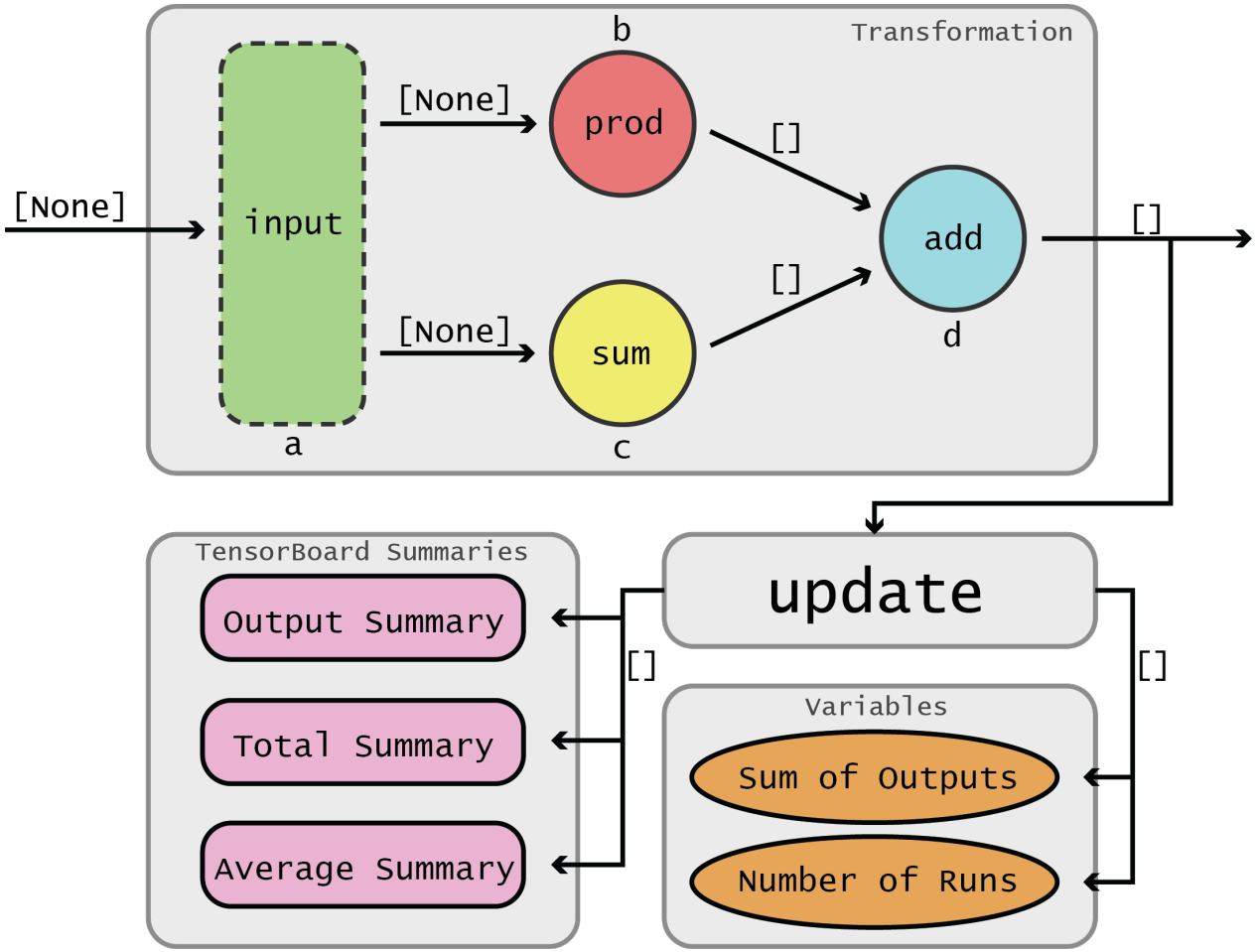
- New exercise:

  Consider the following graph:



TensorFlow name scopes

- New exercise:



TensorFlow 1.X name scopes

- Exercise: *building the graph*

```
import tensorflow as tf
```

```
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, traina-
ble=False, name="global_step")

        # Variable that keeps track of the sum of all output
values over time:
        total_output = tf.Variable(0.0, dtype=tf.float32,
trainable=False, name="total_output")
```

TensorFlow 1.X name scopes

- Exercise: *building the graph*

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...

    with tf.name_scope("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"):
            output = tf.add(b, c, name="output")
```

TensorFlow 1.X name scopes

- Exercise: *building the graph*

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...
    with tf.name_scope("transformation"):
        ...

    with tf.name_scope("update"):
        # Increments the total_output Variable by the latest
output
        update_total = total_output.assign_add(output)

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

TensorFlow 1.X name scopes

- Exercise: *building the graph*

notice the change in newer TF versions (not 2.x)

```
with graph.as_default():
        ...
    with tf.name_scope("update"):
        ...

    with tf.name_scope("summaries"):
        avg = tf.div(update_total, tf.cast(increment_step,
tf.float32), name="average")

        # Creates summaries for output node
        tf.scalar_summary(b'Output', output, name="output_sum-
mary")
        tf.scalar_summary(b'Sum of outputs over time', up-
date_total, name="total_summary")
        tf.scalar_summary(b'Average of outputs over time',
avg, name="average_summary")
```

```
# Summary Operations
with tf.name_scope("summaries"):
    tf.summary.scalar('output summary', output)
    tf.summary.scalar('product of inputs', b)
    tf.summary.scalar('sum of inputs', c)
```

TensorFlow 1.X name scopes

- Exercise: *building the graph*

```
with graph.as_default():
        ...
    with tf.name_scope("summaries"):
        ...
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.initialize_all_variables()
        # Merge all summaries into one Operation
        merged_summaries = tf.merge_all_summaries()
```

```
# Collect all summary Ops in graph
merged_summaries = tf.summary.merge_all()
```

notice the change in newer TF versions (not 2.x)

TensorFlow 1.X name scopes

- Exercise: *running the graph*
  - Let's open up a Session and launch the Graph we just made.
  - We can also open up a tf.summary.FileWriter, which we'll use to save our summaries.
  - Here, we list ./improved_graph as our destination folder for summaries to be saved:

```
sess = tf.Session(graph=graph)
writer = tf.train.SummaryWriter('./improved_graph', graph)
```

  - With a Session started, let's initialize our Variables before doing anything else:

```
sess.run(init)
```

```
writer = tf.summary.FileWriter('./improved_graph', graph)
```

notice the change in newer TF versions (not 2.x)

TensorFlow 1.X name scopes

- Exercise: *running the graph*
  - To run our graph, let's create a helper function, run_graph() so that we don't have to keep typing the same thing over and over again.
  - What we'd like is to pass in our input vector to the function, which will run the graph and save our summaries:

```
def run_graph(input_tensor):
    """ Helper function; runs the graph with given input tensor and
        saves summaries """
    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)
```

TensorFlow 1.X name scopes

- Exercise: *running the graph*
  - To run our graph, let's create a helper function, run_graph() so that we don't have to keep typing the same thing over and over again.
  - What we'd like is to pass in our input vector to the function, which will run the graph and save our summaries:

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

TensorFlow 1.X name scopes

- Exercise: *running the graph*
  - Call run_graph several times with vectors of various lengths like this:

```
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])
```

  - Write the summaries to disk with the FileWriter.flush() method:

```
writer.flush()
```

  - Let's close both FileWriter and Session , now that we're done:

```
writer.close()
sess.close()
```

TensorFlow 1.X name scopes

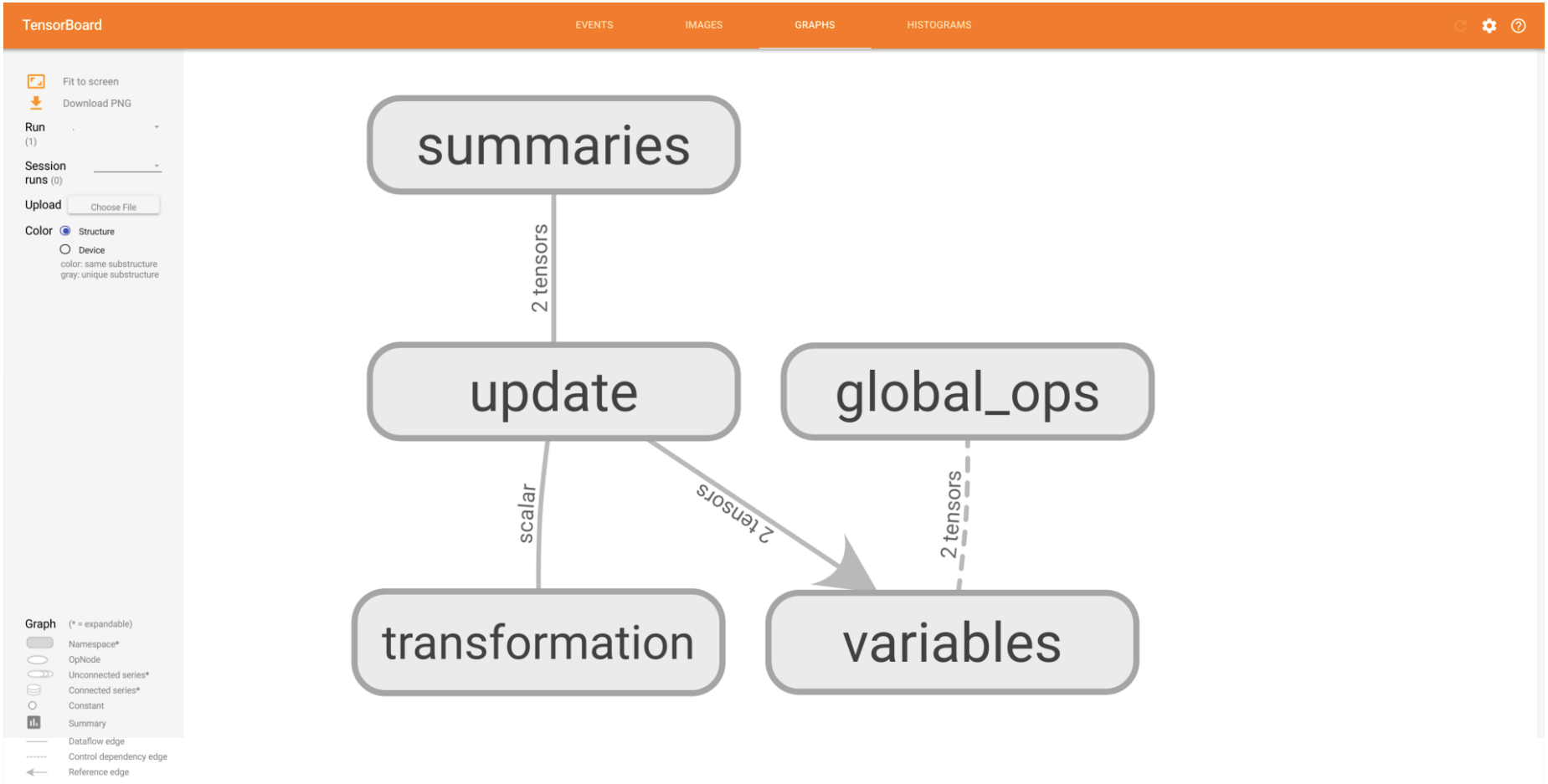
- Exercise: *running the graph*
  - Let's open up TensorBoard and see what we've got.
  - Open up a terminal shell, navigate to the directory where you ran this code (make sure the "improved_graph" directory is located there), and run the following:

```
$ tensorboard --logdir="improved_graph"
```

  - this starts up a TensorBoard server on port 6006, hosting the data stored in "improved_graph"
  - Type in "localhost:6006" into your web browser and let's see what we've got!
  - Let's first check out the "Graph" tab: (on next slide)

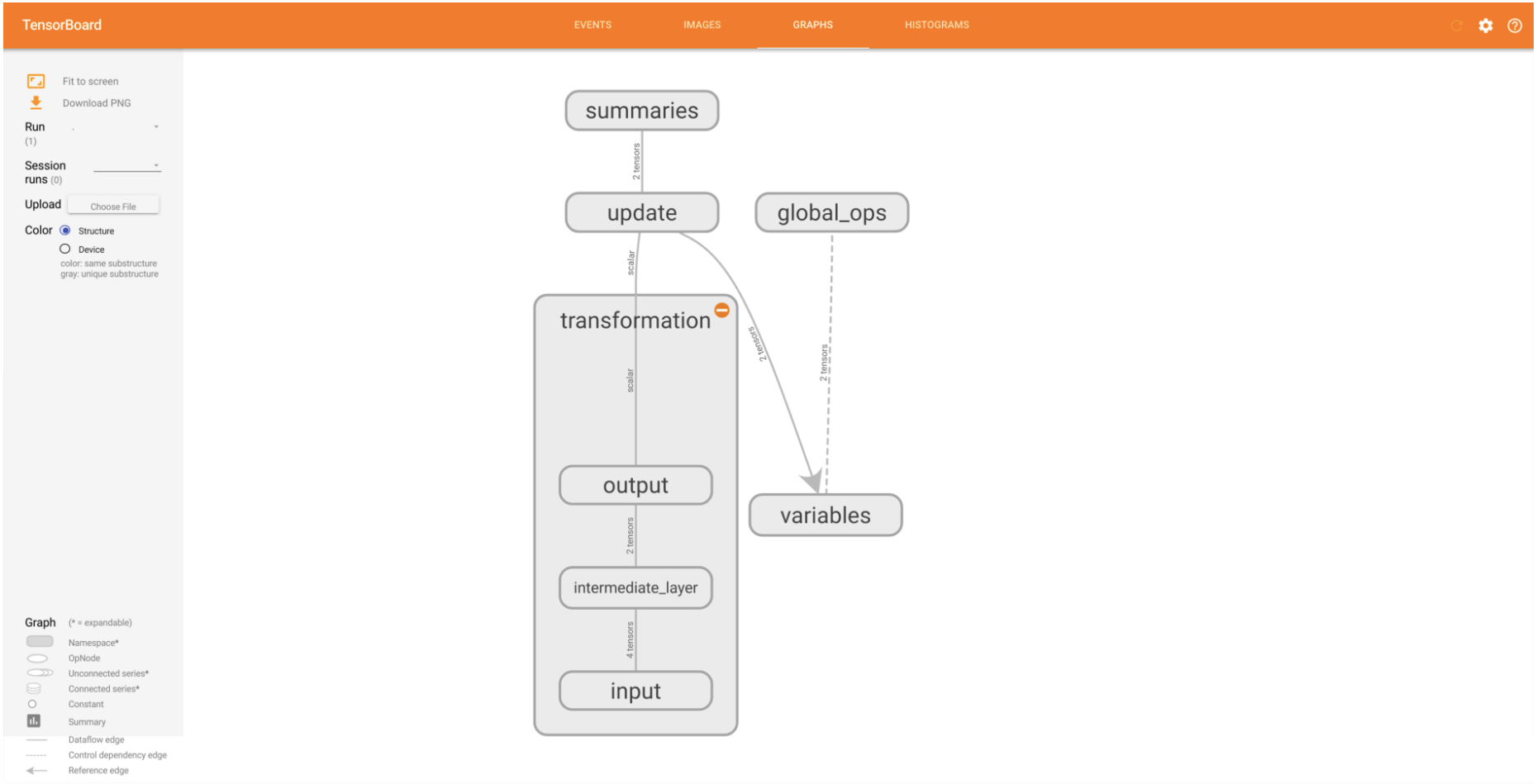
TensorFlow 1.X name scopes

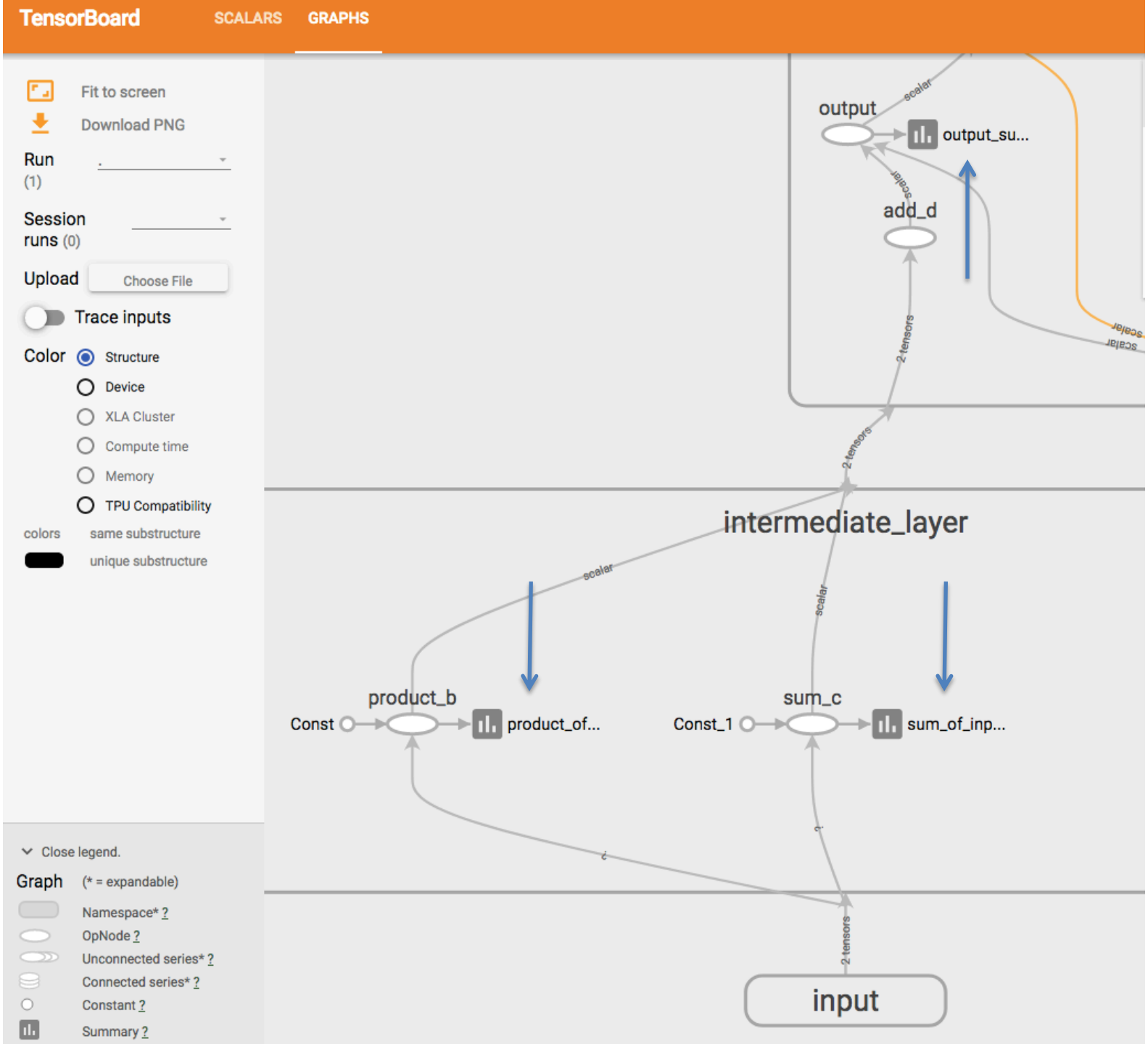
- Exercise: *running the graph*



TensorFlow 1.X name scopes

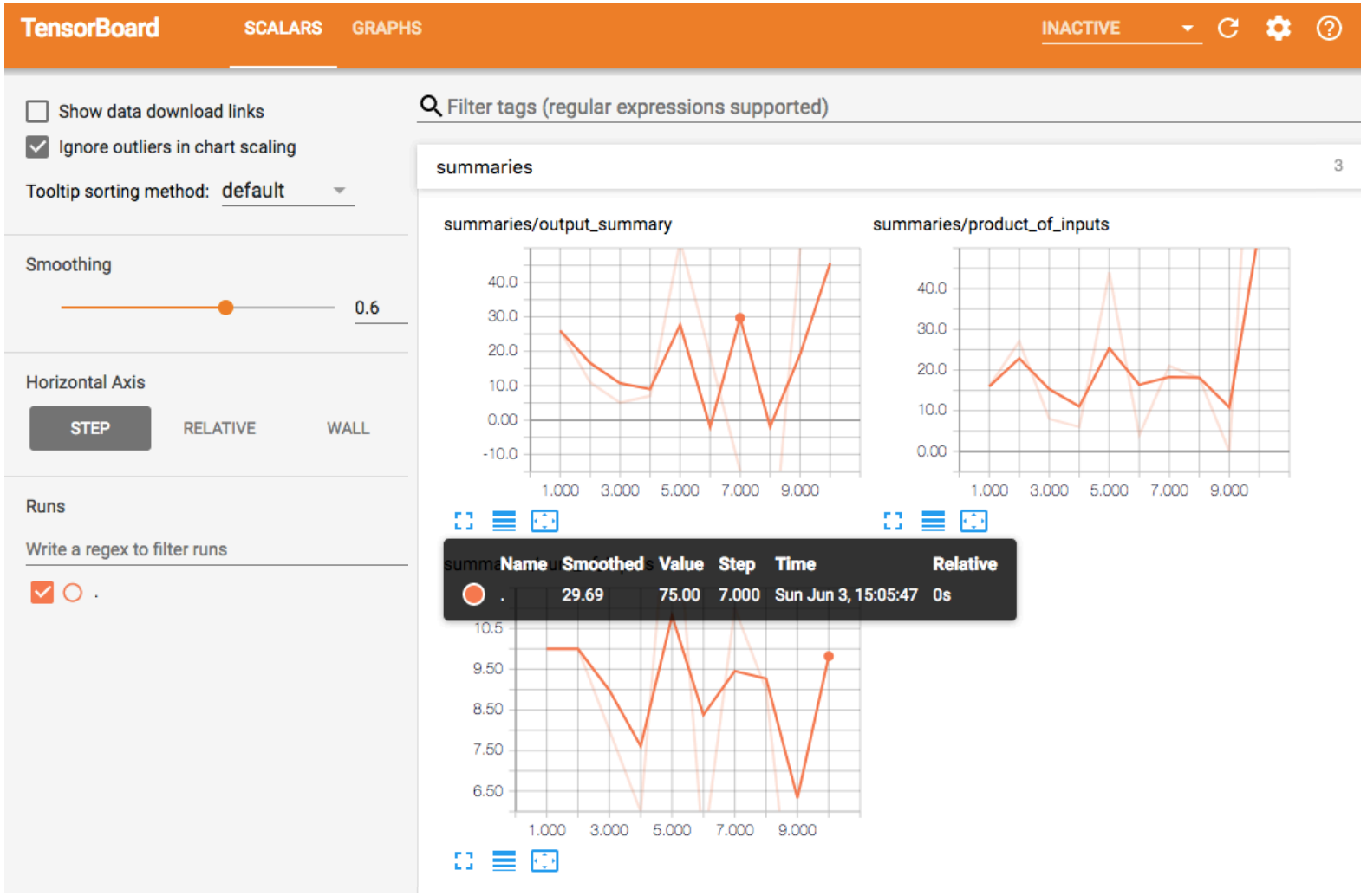
- Exercise: *running the graph*




TensorBoard
SCALARS
GRAPHS
Fit to screen
Download PNG
Run (1)
Session runs (0)
Upload
Choose File
Trace inputs
Color
Structure
Device
XLA Cluster
Compute time
Memory
TPU Compatibility
colors
same substructure
unique substructure
Close legend.
Graph (* = expandable)
Namespace* ?
OpNode ?
Unconnected series* ?
Connected series* ?
Constant ?
Summary ?
output
output_su...
add_d
scalar
2 tensors
intermediate_layer
product_b
Const
product_of...
sum_c
Const_1
sum_of_inp...
input

TensorFlow 1.X name scopes

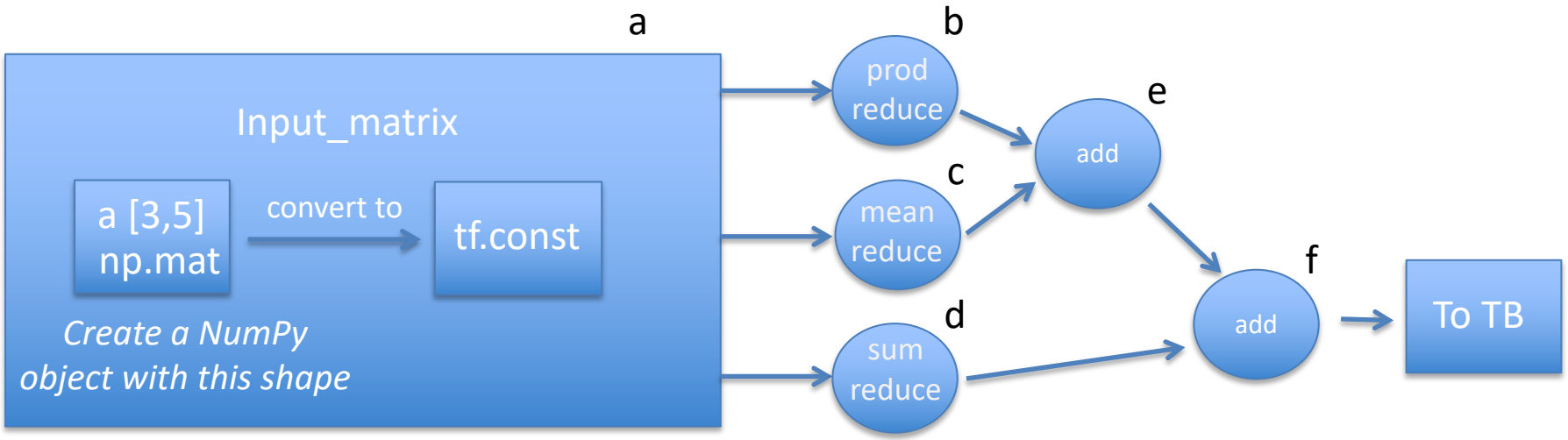
- Exercise: *running the graph*



TensorFlow

HW2:

recreate the graph and visualize it in Tensorboard



TensorFlow

- HW2 *solution*: *recreate the graph and visualize it in Tensorboard*

solution