Getting Started with TensorFlow

Part I : TensorFlow Graphs and Sessions

Nick Winovich

Department of Mathematics

Purdue University

July 2018

Outline

Outline

Outline

The TensorFlow Software Library

TensorFlow™ is an open-source software library designed for high performance, scalable numerical computation, placing a particular emphasis on machine learning and deep neural networks.

- Source code available at: https://github.com/tensorflow/tensorflow
- TensorFlow Python API: https://www.tensorflow.org/api_docs/python
- TensorFlow Youtube Channel: www.youtube.com/channel/UC0rqucBdTuFTjJiefW5t-IQ

Python API and Installation

The TensorFlow library can be installed using the pip package manager for Python by running the following command:

```
$ pip install --upgrade [tfBinaryURL for Python 3.n]
```

The tfBinaryURLs, along with a complete set of instructions for installation, can be found on the TensorFlow API Installation Page:

https://www.tensorflow.org/install/

- Consider installing TensorFlow in a virtual environment, especially if the operating system you are using has core software with Python as a dependency (e.g. GNU/Linux).

Tensors

“A Tensor is a symbolic handle to one of the outputs of an Operation. It does not hold the values of that operation’s output, but instead provides a means of computing those values in a TensorFlow tf.Session.” *(TensorFlow API r1.8)*

Roles and Properties of Tensors:

- Used to connect operations and establish the dependencies and dataflow associated with executing a given computation
- Provide a way of referring to the outputs of operations
- Do not store concrete values, but are aware of the data type and information regarding the expected shape of the result

Operations

"An Operation is a node in a TensorFlow Graph that takes zero or more Tensor objects as input, and produces zero or more Tensor objects as output. Objects of type Operation are created by calling a Python op constructor (such as tf.matmul) or tf.Graph.create_op." *(TensorFlow API r1.8)*

Roles and Properties of Operations:

- Specify computations connecting input and output tensors
- Can be used to produce tensors (e.g. by randomly sampling from a distribution or simply assigning a constant value)
- Define optimization procedures and training summaries

TensorFlow Graphs

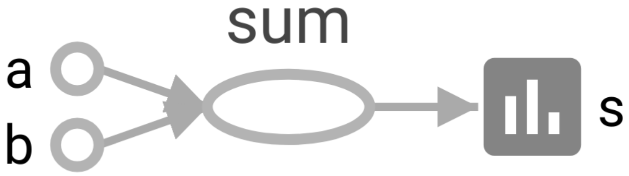
"TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices."

"In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation." *(TensorFlow API r1.8)*

- Graphs represent the overall dataflow of a model
- Specify dependencies/connections between computations

Example TensorFlow Graph

```
import tensorflow as tf

# Define constants ‘a’ and ‘b’
a = tf.constant(1.0, dtype=tf.float32, name="a")
b = tf.constant(2.0, dtype=tf.float32, name="b")

# Compute sum ‘a+b’
s = tf.add(a, b, name="sum")
print(s)  # Tensor("sum:0", shape=(), dtype=float32)
```



TensorFlow Sessions

“**The tf.Session.run method is the main mechanism for running a tf.Operation or evaluating a tf.Tensor**. You can pass one or more tf.Operation or tf.Tensor objects to tf.Session.run, and TensorFlow will execute the operations that are needed to compute the result.”

“tf.Session.run requires you to specify a list of fetches, which determine the return values, and may be a tf.Operation, a tf.Tensor, or a tensor-like type such as tf.Variable. **These fetches determine what subgraph of the overall tf.Graph must be executed** to produce the result: this is the subgraph that contains all operations named in the fetch list, plus all operations whose outputs are used to compute the value of the fetches.” *(TensorFlow API r1.8)*

Example TensorFlow Session

```
import tensorflow as tf

# Define constants ‘a’ and ‘b’
a = tf.constant(1.0, dtype=tf.float32, name="a")
b = tf.constant(2.0, dtype=tf.float32, name="b")

# Compute sum ‘a+b’
s = tf.add(a, b, name="sum")
print(s)  # Tensor("sum:0", shape=(), dtype=float32)

# Initialize session
with tf.Session() as sess:

    # Execute graph
    result = sess.run(s)
    print(result)  # 3.0
```

- The tensor `"s"` knows the shape and datatype of the result
- The actual value is computed when `sess.run(s)` is called

Naming Conventions in TensorFlow

Why does TensorFlow add `":0"` to the end of names?

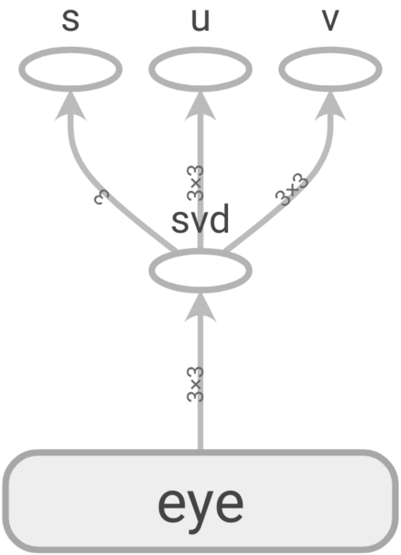
The suffix `":0"` indicates that the tensor corresponds to the first (i.e. Python index 0) output of the operation which produces it. Subsequent outputs are named sequentially (`":1"`, `":2"`, etc.).

For example, the addition operation in the previous example has been assigned the name `"sum"`; accordingly, the symbolic tensor `"s"` is referred to on the graph as `"sum:0"`, indicating that it is the first output of the `"sum"` operation.

Naming Conventions in TensorFlow

```python
import tensorflow as tf

# Define graph for computing singular value decomposition
A = tf.eye(3)
s, u, v = tf.linalg.svd(A, name="svd")

print(s)  # Tensor("svd:0", shape=(3,), dtype=float32)
print(u)  # Tensor("svd:1", shape=(3, 3), dtype=float32)
print(v)  # Tensor("svd:2", shape=(3, 3), dtype=float32)

# Initialize session
with tf.Session() as sess:

    # Execute graph to compute singular values and vectors
    s_val, u_val, v_val = sess.run([s,u,v])

    print(s_val)  # [1. 1. 1.]
    print(u_val)  # [[1. 0. 0.], [0. 1. 0.], [0. 0. 1.]]
    print(v_val)  # [[1. 0. 0.], [0. 1. 0.], [0. 0. 1.]]
```

Example: TensorFlow Graph for SVD



Symbolic Tensors and Python Variable Scope

Why not just use `"s"` instead of `"s_val"`?

Reusing the Python variable `"s"` will override the reference to the tensor object in the graph; so if the tensor `"s"` needs to be evaluated later on, we would have to resort to using `tf.get_default_graph.get_tensor_by_name("name")` ...

Note:

An easy way to avoid making this mistake is to use functions, classes, and methods which define variables with local scopes.

Placeholders and Feed Dictionaries

"TensorFlow's feed mechanism lets you inject data into any Tensor in a computation graph. A Python computation can thus feed data directly into the graph."

"Supply feed data through the feed_dict argument to a run() or eval() call that initiates computation."

"While you can replace any Tensor with feed data, including variables and constants, the best practice is to use a tf.placeholder node." *(TensorFlow API r1.8)*

- Placeholders are used to 'hold the place' of data in the graph
- Data is fed into these nodes using 'feed dictionaries'

Placeholders Example

```python
import tensorflow as tf

# Define a placeholder for input values
x = tf.placeholder(tf.float32, [None], name="x")

# Define a constant value used to shift input values
shift = tf.constant(10.0, dtype=tf.float32, name="shift")

# Define operation to compute shifted values
y = tf.add(x, shift, name="y")

# Initialize TensorFlow session
with tf.Session() as sess:

    # Specify values to feed into placeholder ‘x’
    fd = { x : [1.,2.,3.] }

    # Run operation ‘tf.add’
    y_vals = sess.run(y, feed_dict=fd)
    print(y_vals)  # [11. 12. 13.]
```

Placeholders Example

Why is the shape of the placeholder "x" set to "[None]"?

The use of "None" indicates that the size of a given dimension is not specified. This is particularly common to use for the first dimension of the input which is typically reserved for the batch size. In this case, the individual inputs are scalars (with "shape=()") so the full shape of the placeholder is "[None]".

For an RGB image with resolution 64x64, we may instead use a placeholder with shape "[None,64,64,3]"; specifying the sizes of the other dimensions is typically necessary in order for TensorFlow to determine the correct shapes of weight matrices.

Variables

"A TensorFlow variable is the best way to represent shared, persistent state manipulated by your program."

"A tf.Variable represents a tensor whose value can be changed by running ops on it."

"Unlike tf.Tensor objects, a tf.Variable exists outside the context of a single session.run call." *(TensorFlow API r1.8)*

- Used to define and store network parameters
- Specify the current state of the model
- Used to save models in "checkpoints"

Example: Using Variables for Model Parameters

```
import tensorflow as tf

# Define placeholders for input and ouput values
x = tf.placeholder(tf.float32, [None], name="x")
y = tf.placeholder(tf.float32, [None], name="y")

# Define trainable variable for slope
m = tf.get_variable("slope", dtype=tf.float32, shape=())

# Define trainable variable for intercept
b = tf.get_variable("intercept", dtype=tf.float32, shape=())

# Define prediction using slope and intercept variables
prediction = tf.add(tf.multiply(m,x),b)

# Define loss function for predictions w.r.t. true outputs
loss = tf.losses.mean_squared_error(y, prediction)
```

Eager Execution

"TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later."

```
import tensorflow as tf

# Enable eager execution
tf.enable_eager_execution()

# Define constants ‘a’ and ‘b’
a = tf.constant(1.0, dtype=tf.float32, name="a")
b = tf.constant(2.0, dtype=tf.float32, name="b")

# Compute sum ‘a+b’
s = tf.add(a, b, name="sum")
print(s)  # tf.Tensor(3.0, shape=(), dtype=float32)
```

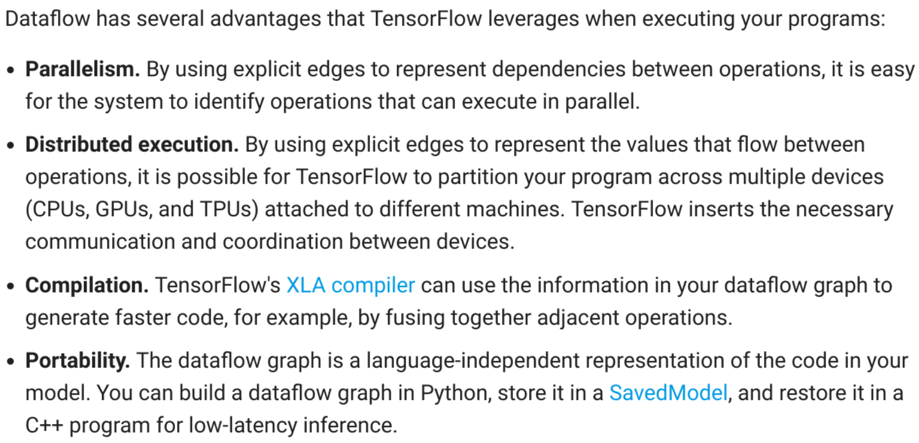
Eager Execution

"While eager execution makes development and debugging more interactive, TensorFlow graph execution has advantages for distributed training, performance optimizations, and production deployment. However, writing graph code can feel different than writing regular Python code and more difficult to debug."

"The tf.contrib.eager module contains symbols available to both eager and graph execution environments and is useful for writing code to work with graphs: tfe = tf.contrib.eager" *(TensorFlow API r1.8)*

- Good for debugging, but has some disadvantages
- The `tfe` module aims to combine eager mode and graphs

Why Dataflow Graphs?



Dataflow has several advantages that TensorFlow leverages when executing your programs:

- **Parallelism.** By using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel.
- **Distributed execution.** By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.
- **Compilation.** TensorFlow's XLA compiler can use the information in your dataflow graph to generate faster code, for example, by fusing together adjacent operations.
- **Portability.** The dataflow graph is a language-independent representation of the code in your model. You can build a dataflow graph in Python, store it in a SavedModel, and restore it in a C++ program for low-latency inference.

https://www.tensorflow.org/programmers_guide/graphs

Outline

The tf.layers Module

TensorFlow offers several predefined network layers in the tf.layers module. These layers help streamline the process of creating all of the variables, tensors, and operations necessary for implementing many of the most commonly used neural network layers.

In addition, these layers offer a way to easily specify the use of:

- activation functions
- bias terms
- weight regularization
- weight constraints

The tf.layers Module

Commonly Used Layers:

- tf.layers.dense
- tf.layers.conv2d
- tf.layers.conv2d_transpose
- tf.layers.dropout
- tf.layers.flatten
- tf.layers.max_pooling2d
- tf.layers.average_pooling2d
- tf.layers.batch_normalization

The full list of predefined network layers available in tf.layers, along with descriptions of the options/arguments for each layer, can be found on the official TensorFlow Python API documentation page:

https://www.tensorflow.org/api_docs/python/tf/layers

(RNN cells are also available in tf.contrib.layers.rnn)

Example: Dense Network

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None,1], name="x")
y = tf.placeholder(tf.float32, [None,1], name="y")

def dense_network(x, name=None):
    h = tf.layers.dense(x, 10, activation=tf.nn.relu)
    h = tf.layers.dense(h, 10, activation=tf.nn.relu)
    h = tf.layers.dense(h, 1, activation=None, name=name)
    return h

prediction = dense_network(x, name="prediction")
```

- Dense layers assume inputs have the form [batch_size, shape]
- Always consider the activation function used for the final layer
- The final prediction must have the correct shape (i.e. `y.shape`)

Adding Dropout

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None,1], name="x")
y = tf.placeholder(tf.float32, [None,1], name="y")

def dense_network(x, name=None):
    h = tf.layers.dense(x, 100, activation=tf.nn.relu)
    h = tf.layers.dropout(h, rate=0.1, training=True)
    h = tf.layers.dense(h, 100, activation=tf.nn.relu)
    h = tf.layers.dense(h, 1, activation=None, name=name)
    return h

prediction = dense_network(x, name="prediction")
```

- `"rate"` = probability of dropping/removing a unit (i.i.d.)
- Having `"training"` always set to `"True"` is problematic ...

Using a Training Placeholder

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, [None,1], name="x")
y = tf.placeholder(tf.float32, [None,1], name="y")
train = tf.placeholder(tf.bool, name="train")

def dense_network(x, training=False, name=None):
    h = tf.layers.dense(x, 100, activation=tf.nn.relu)
    h = tf.layers.dropout(h, rate=0.1, training=training)
    h = tf.layers.dense(h, 100, activation=tf.nn.relu)
    h = tf.layers.dense(h, 1, activation=None, name=name)
    return h

prediction = dense_network(x, training=train, name="prediction")
```

- Now we can feed the values `"True"` or `"False"` into the placeholder `"train"` to indicate whether the network is being trained or just being used for inference during a given run call

Defining Layers with Custom Default Arguments

```
# Define custom dense/fully-connected layer with dropout
def dense(x, n_out, activation=tf.nn.relu,
          drop_rate=0.01, name=None, training=True):

    # Define variable initializers
    wt_init = tf.random_normal_initializer(stddev=0.05)
    bi_init = tf.random_normal_initializer(mean=0.1, stddev=0.3)

    # Define dense layer
    y = layers.dense(x, n_out,
                     activation=activation,
                     use_bias=True,
                     kernel_initializer=wt_init,
                     bias_initializer=bi_init,
                     name=name)

    # Define dropout
    y = layers.dropout(y, rate=drop_rate, training=training)
    return y
```

Outline

Defining Loss Functions

TensorFlow has a number of predefined loss functions defined:

https://www.tensorflow.org/api_docs/python/tf/losses

Some common examples include:

- `tf.losses.mean_squared_error`
- `tf.losses.softmax_cross_entropy`

Custom Loss Functions: Loss functions can also be defined manually, but should be composed only of TensorFlow operations to ensure gradients are computed correctly. The shape/dimensions also need to be 'reduced' during the loss calculation to produce a scalar value; this can typically be done using e.g. `tf.reduce_sum`.

Specifying an Optimization Algorithm

Several predefined optimizers are also available, including:

- `tf.train.GradientDescentOptimizer`
- `tf.train.AdagradOptimizer`
- `tf.train.RMSPropOptimizer`
- `tf.train.AdamOptimizer`

Optimization Hyperparameters:

Each optimizer has its own set of arguments/parameters which often need to be adjusted to achieve the optimal training performance; links to the original papers defining each of these optimizers are provided on the associated TensorFlow API pages.

Defining the Optimization Operation in the Graph

Standard Example:

```
optim = tf.train.AdamOptimizer(0.001).minimize(loss)
```

`.minimize()`

> "This method simply combines calls compute_gradients() and apply_gradients(). If you want to process the gradient before applying them call compute_gradients() and apply_gradients() explicitly instead of using this function."
> *(TensorFlow API r1.8)*

- For example, gradients can be "clipped" using the L2-norm via: `compute_gradients`([loss]), `tf.clip_by_norm`([gradients]), and `apply_gradients`([gradient/variable pairs]).

Random Initialization of Variables

The values of Variables are typically adjusted according to the network's optimization procedure, but first must be initialized:

```
# Define variable initializer
init = tf.global_variables_initializer()

with tf.Session() as sess:

    # Execute variable initialization
    sess.run(init)
```

If no explicit variable initializer is specified, the default is to use:

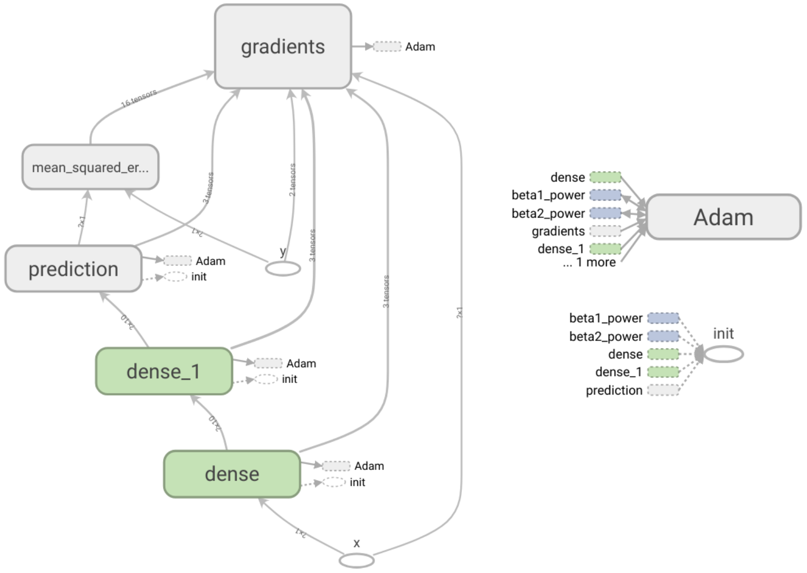
`tf.glorot_uniform_initializer`

which is defined in "*Understanding the difficulty of training deep feedforward neural networks*" by Glorot and Bengio.

Example: Training a Dense Network to Model sin(x)

```
# Using network/placeholders from "Network Design with tf.layers"
prediction = dense_network(x, name="prediction")

# Define loss function
loss = tf.losses.mean_squared_error(y, prediction)

# Define optimization operation
optim = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)

# Define variable initializer
init = tf.global_variables_initializer()

with tf.Session() as sess:
    # Initialize variables and begin training
    sess.run(init)
    for n in range(0,10000):
        # Create artificial data from sin(x) with batch size 100
        x_batch = np.pi/2 * np.random.normal(size=[100, 1])
        y_batch = np.sin(x_batch)
        fd = {x: x_batch, y: y_batch}
        sess.run(optim, feed_dict=fd)
```

Example: TensorFlow Graph for Dense Network



Note on Batch Normalization

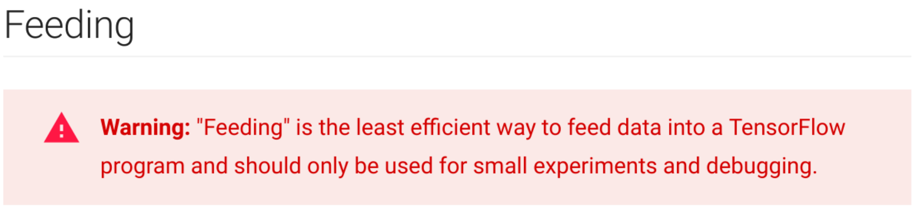
While most tf.layers can simply be added into the network directly without further modifications to the underlying dataflow graph, batch normalization requires “update operations” to be added to the list of dependencies for the training/optimization operation:

```
# Retrieve update ops for batch normalization
update_ops = tf.GraphKeys.UPDATE_OPS

# Define optimizer for training
with tf.control_dependencies(tf.get_collection(update_ops)):
    optim = tf.train.AdamOptimizer(0.001).minimize(loss)
```

Outline

Outline

Feeding Data in TensorFlow



Feeding

Warning: "Feeding" is the least efficient way to feed data into a TensorFlow program and should only be used for small experiments and debugging.

https://www.tensorflow.org/api_guides/python/reading_data#Feeding

The tf.data API

"This is an improved version of the old input methods—feeding and QueueRunner—which are described below for historical purposes." *(TensorFlow API r1.8)*

- In the past, `tf.train.QueueRunner` was used to construct more efficient data pipelines; unfortunately, it was somewhat (extremely) difficult to get the queues working correctly.
- The `tf.data` API provides a much more straightforward approach for constructing efficient input pipelines.
- The two main abstractions introduced in the `tf.data` API are:

  `tf.data.Dataset` and `tf.data.Iterator`

tf.data.Dataset

“A tf.data.Dataset represents a sequence of elements, in which each element contains one or more Tensor objects. For example, in an image pipeline, an element might be a single training example, with a pair of tensors representing the image data and a label.” *(TensorFlow API r1.8)*

- Datasets consist of a list of elements of training examples
- Elements typically contain tensors for input/output tuples
- `tf.data.Datasets` provide a streamlined approach to batching, transforming, and shuffling training data

tf.data.Iterator

"A tf.data.Iterator provides the main way to extract elements from a dataset. The operation returned by Iterator.get_next() yields the next element of a Dataset when executed, and typically acts as the interface between input pipeline code and your model. " *(TensorFlow API r1.8)*

- Once a dataset is defined, an associated iterator can be created and used for fetching examples from the dataset
- Evaluating `Iterator.get_next()` returns an operation for retrieving the next element of the underlying dataset

Simplest Method: tf.data.from_tensor_slices

```
# Define iterator for dataset with mini-batch size 100
# (where x_data and y_data are arrays of input/output data)
dataset = tf.data.Dataset.from_tensor_slices((x_data,y_data))

dataset = dataset.repeat(5)      # repeat for 5 epochs
dataset = dataset.shuffle(2500) # shuffle w/ buffer size 2500
dataset = dataset.batch(100)     # create batches of 100
iterator = dataset.make_one_shot_iterator() # create iterator
```

- The `tf.data.Dataset.from_tensor_slices` method facilitates the creation of datasets directly from arrays
- Datasets can easily be repeated, shuffled, and batched using predefined methods of `tf.data.Dataset` objects
- More efficient 'fused' operations are available in `tf.contrib`

Placing Dataset Iterators in the Graph

```
# Retrieve next batch from dataset
x_batch, y_batch = iterator.get_next()

# Compute network prediction
prediction = dense_network(x_batch, name="prediction")

# Evaluate loss function
loss = tf.losses.mean_squared_error(y_batch, prediction)
```

- The output of `iterator.get_next()` will depend on the form of training examples from the underlying dataset; here we assume an example consists of a pair of input/output tensors
- Once a batch is retrieved, the graph is constructed as before

More Scalable Approach: Using *.tfrecords Files

```
# Create list of .tfrecords files
files = tf.data.Dataset.list_files("training-*.tfrecords")

# Create datasets with 4 mebibyte buffers
def tfrecord_dataset(fname):
    return tf.data.TFRecordDataset(fname,buffer_size=4*1024*1024)

# Read from multiple files in parallel
dataset = files.apply(tf.contrib.data.parallel_interleave(
    tfrecord_dataset, cycle_length=8, sloppy=True))

# Apply fused ops for shuffling, repeating, parsing, and batching
dataset = dataset.apply(
    tf.contrib.data.shuffle_and_repeat(10000))
dataset = dataset.apply(
    tf.contrib.data.map_and_batch(_parse_data, 100)

iterator = dataset.make_one_shot_iterator() # create iterator
```

∗ We will discuss this approach in more detail in Part II . . .

Outline

Saving Training Logs with tf.summary

```
# Define loss function and optimization operation
loss = tf.losses.mean_squared_error(y, prediction)
optim = tf.train.AdamOptimizer(learning_rate).minimize(loss)

# Define merged summary operation
loss_sum = tf.summary.scalar("loss", loss)
sum_op = tf.summary.merge_all()

with tf.Session() as sess:
    sess.run(init)
    graph = tf.get_default_graph()
    writer = tf.summary.FileWriter("logs", graph=graph)
    for n in range(0,10000):
        _, summary = sess.run([optim, sum_op])
        writer.add_summary(summary, n)
```

- `tf.summary.scalar` is used to define scalar summaries
- `tf.summary.merge_all()` defines a merged summary op
- `tf.summary.FileWriter` is used to write summary files

Using TensorBoard to Visualize Summaries

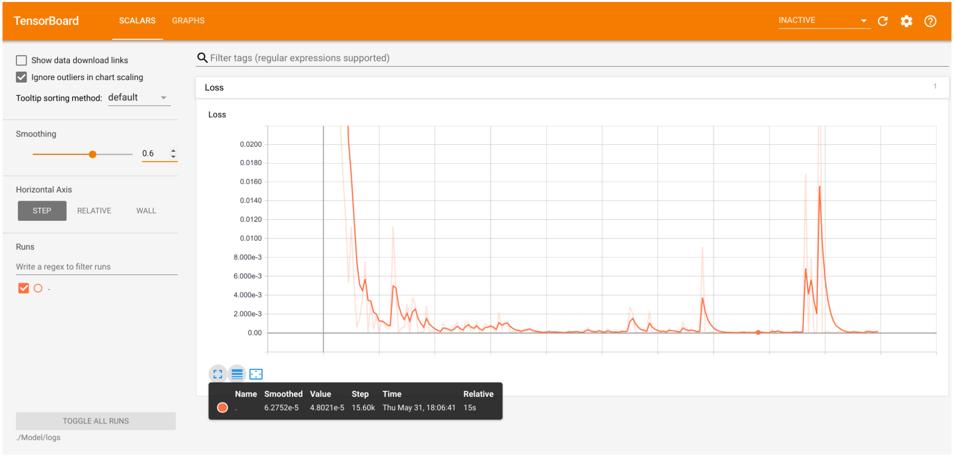
The scalar summaries, along with visual representations of the underlying dataflow graph, can be viewed by issuing the command:

```
$ tensorboard --logdir logs
```

and navigating to the address `localhost:6006` in a web browser.

- If `localhost` does not work, you can try `127.0.0.1:6006`
- The default port can also be changed using the `--port` flag
- "`with tf.name_scope(...)`" blocks can be used to organize graphs and improve/consolidate visualization in TensorBoard

Using TensorBoard to Visualize Summaries



Show data download links
[✓] Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing: 0.6

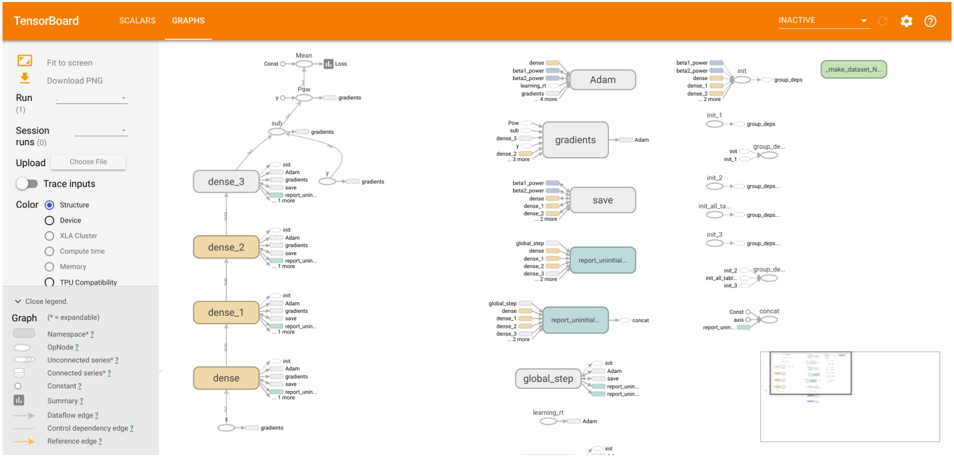
Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs
Write a regex to filter runs
[✓] .

./Model/logs

```
Name  Smoothed    Value       Step    Time                      Relative
.     6.2752e-5   4.8021e-5   15.60k  Thu May 31, 18:06:41      15s
```

Using TensorBoard to Visualize Summaries



Run (1)

Session runs (0)

Upload

Trace inputs

Color [✓ Structure] Device XLA Cluster Compute time Memory TPU Compatibility

Close legend.

Graph (* = expandable)

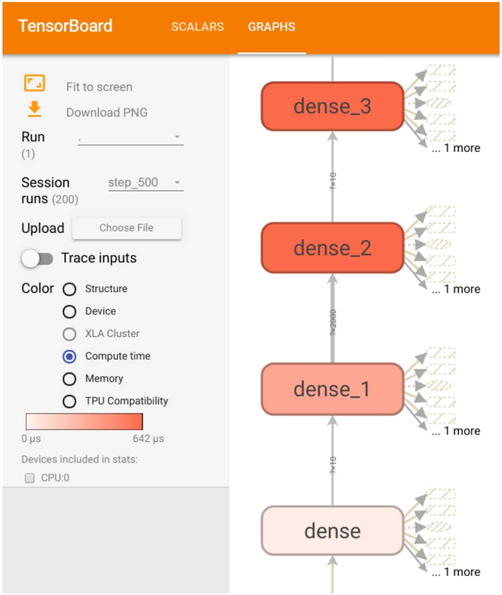
▭	Namespace* ?
⬭	OpNode ?
⬭	Unconnected series* ?
☰	Connected series* ?
○	Constant ?
📊	Summary ?
→	Dataflow edge ?
⇢	Control dependency edge ?
→	Reference edge ?

Using Metadata to Find Computational Bottlenecks

```
# Record meta data (e.g. memory usage, compute time, etc.)
run_opts = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
run_meta = tf.RunMetadata()

_, summary = sess.run([optim, sum_op],
                      options=run_opts, run_metadata=run_meta)

writer.add_run_metadata(run_meta, n)
writer.add_summary(summary, n)
```

- `sess.run` can be passed options for storing metadata
- Metadata is written using `writer.add_run_metadata`

Using Metadata to Find Computational Bottlenecks



TensorBoard SCALARS GRAPHS

Fit to screen

Download PNG

Run (1)

Session runs (200): step_500

Upload: Choose File

Trace inputs

Color: Structure, Device, XLA Cluster, [✓ Compute time], Memory, TPU Compatibility

0 μs 642 μs

Devices included in stats:

CPU:0

Outline

Class Definitions of Models

As models become more complex, it is advisable to organize your code in blocks using Python classes. This provides a more natural approach to model representation by defining the core components of models in terms of properties and methods.

Properties/Attributes:

- `tf.graph`
- `tf.session`
- `tf.data.Dataset`
- `tf.train.global_step`
- `tf.train.Saver`
- training hyperparameters

Methods:

- `set_session()`
- `build_model()`
- `predict()`
- `train()`
- `evaluate()`

Basics of Python Classes

```
# Define a new class named "MyClass"
class MyClass(object):

    # Define a constructor for the class
    def __init__(self, val):
        self.val = val

    # Define a property "val"
    @property
    def val(self):
        return self._val

    # Define a setter for "val"
    @val.setter
    def val(self, val):
        self._val = val

    # Define a method "add_to_val"
    def add_to_val(self, a):
        self.val += a
```

Basics of Python Classes

```python
# Create instance of class
obj = MyClass(1.0)

# Call "add_to_val" method
obj.add_to_val(2.0)

# Retrieve "val" using default getter
print(obj.val)
```

- Inheritance from "`object`" is used for Python 2 compatability
- The "`__init__`" method is called by e.g. "`MyClass(1.0)`"
- The `self.` prefix is passed to all methods in the class
- Using the "`@property`" decorator is not strictly necessary
- Setters are helpful for working with constraints on properties

The __init__ Method

The `__init__` method can be used for general setup tasks, such as:

- Specifying training hyperparameters
- Calling methods to build the underlying graph/model
- Initializing or restoring variables from a checkpoint
- Specifying the current training step (e.g. global step tensor)
- Checking that training data files exist in the filesystem
- Saving copies of current configuration files for records

Methods: build_model()

A `build_model` method can be defined to construct the underlying graph for the proposed model; in particular it may define:

- placeholders and variables
- network layers and regularization
- model predictions and loss functions
- optimization and summary operations

Methods: train(), predict(), and evaluate()

Following the coding style used in the `tf.estimator` module, the `train`, `predict`, and `evaluate` methods are defined as follows:

`train()`

“Trains a model given training data input_fn.”

`predict()`

“Yields predictions for given features.”

`evaluate()`

“Evaluates the model given evaluation data input_fn.” *(TensorFlow API r1.8)*

Defining main() Function for Executing Code

```
# Initialize and train model
def main():

    # Initialize model
    model = Model()

    # Create session for training
    with tf.Session() as sess:

        # Set model session
        model.set_session(sess)

        # Build model graph
        model.build_model()

        # Train model
        model.train()

# Run main() function when called directly
if __name__ == '__main__':
    main()
```

Additional Examples and Explanations

Additional examples can be found on GitHub at:

https://github.com/nw2190/TensorFlow_Examples

Explanations of the code provided above are also available at:

https://www.math.purdue.edu/~nwinovic/tensorflow.html