Introduction to TensorFlow

Stephan Eule, Erik Schultheis

January 26, 2018

A Linear Model

Multiply with weights and add a bias:

$$y = W \cdot x + b$$

Rewritten in function notation:

$$\begin{aligned} y &= +(W \cdot x, b) \\ &= +(\cdot(W, x), b) \end{aligned}$$

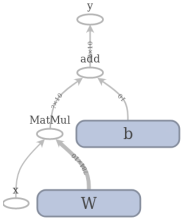

Figure: The linear model in graph form.

Operations and Data

The expression consists of operations that are applied to data:

$$y = +(\cdot(W, x), b)$$

An operation is not necessarily a function!

Why bother?

Autodifferentiation

We don't want to calculate gradients by hand! Use the graph and apply the chain rule automatically.

Why bother?

Parallelism

An Ideal World

The result of each operation depend only on its input data.
Traverse the graph backwards and find all operations whose inputs are data that is ready. These (a, b, c, d) can be evaluated in parallel. Whenever an operation finishes, look if we can start the next one.

$$f(g(a, b), h(c, d()))$$

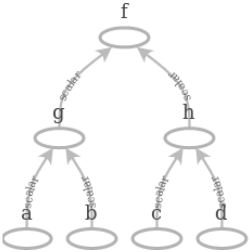

Figure: A parallelizable computation.

Why bother?

Parallelism

An Ideal World

The result of each operation depend only on its input data.
Traverse the graph backwards and find all operations whose inputs are data that is ready. These (a, b, c, d) can be evaluated in parallel. Whenever an operation finishes, look if we can start the next one.

$$f(g, h(c, d))$$

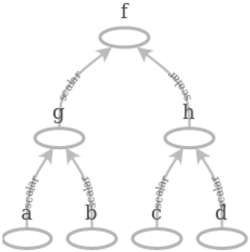

Figure: A parallelizable computation.

Why bother?

Parallelism

An Ideal World

The result of each operation depend only on its input data.
Traverse the graph backwards and find all operations whose inputs are data that is ready. These (a, b, c, d) can be evaluated in parallel. Whenever an operation finishes, look if we can start the next one.

$$f(g, h)$$

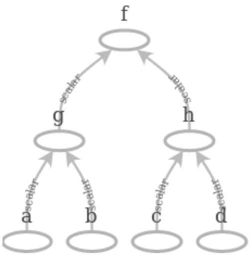

Figure: A parallelizable computation.

Why bother?

Parallelism

An Ideal World

The result of each operation depend only on its input data.
Traverse the graph backwards and find all operations whose inputs are data that is ready. These (a, b, c, d) can be evaluated in parallel. Whenever an operation finishes, look if we can start the next one.

f

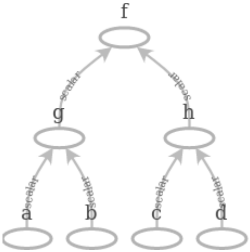

Figure: A parallelizable computation.

An Ideal World

The result of each operation depend only on its input data.
Traverse the graph backwards and find all operations whose inputs are data that is ready. These (a, b, c, d) can be evaluated in parallel. Whenever an operation finishes, look if we can start the next one.

f

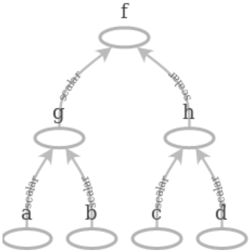

Figure: A parallelizable computation.

The Harsh Reality

We want operations to be more general than pure functions. Therefore, if one operation depends on a side effect of another (but not on data produced by it), we need to add a "fake" (dataless) edge to the graph, called a *control dependency*.

Why bother?

Efficiency

Acyclic Graphs

Calculate only what is needed. Up to now we have considered only trees, but we can use any directed acyclic graph.

$$x = f(g(), h()), \quad y = s(x(), h()), \quad z = f(t()) \tag{1}$$

Why bother?

Efficiency

Acyclic Graphs

Calculate only what is needed. Up to now we have considered only trees, but we can use any directed acyclic graph.

$$x = f(g(), h()), \quad y = s(x(), h()), \quad z = f(t()) \tag{1}$$

Calculation

To calculate y just do the same as before. We will automatically skip useless operations and compute things only once:

$$x = f(g, h), \quad y = s(x, h), \quad z = f(t()) \tag{2}$$

Why bother?

Efficiency

Acyclic Graphs

Calculate only what is needed. Up to now we have considered only trees, but we can use any directed acyclic graph.

$$x = f(g(), h()), \quad y = s(x(), h()), \quad z = f(t()) \tag{1}$$

Calculation

To calculate y just do the same as before. We will automatically skip useless operations and compute things only once:

$$x = f, \quad y = s(x, h), \quad z = f(t()) \tag{2}$$

Why bother?

Efficiency

Acyclic Graphs

Calculate only what is needed. Up to now we have considered only trees, but we can use any directed acyclic graph.

$$x = f(g(), h()), \quad y = s(x(), h()), \quad z = f(t()) \tag{1}$$

Calculation

To calculate y just do the same as before. We will automatically skip useless operations and compute things only once:

$$x = f, \quad y = s, \quad z = f(t()) \tag{2}$$

Why bother?

TensorFlow is based on the idea of using a computation graph!

Basic Data Structures

Overview

Graph

The `tf.Graph` class manages a computation graph.

Operation

The `tf.Operation` class represents an operation.

Data

The `tf.Tensor` class represents blobs of data that are inputs and outputs of operations.

Executation

The `tf.Session` class manages the execution of computations and external resources that operations can interact with.

Graphs

`tf.Graph`

Graph objects

`tf.Graph` objects manage the computational graph. Tensors and operations are identified by a unique name. It also provides *context managers* and some metadata; we'll look at that later.

Graphs
`tf.Graph`

Graph objects

`tf.Graph` objects manage the computational graph. Tensors and operations are identified by a unique name. It also provides *context managers* and some metadata; we'll look at that later.

The Default Graph

TensorFlow always has a graph as the *default graph* (even if you don't create any graph). New operations are added to the current default graph.

Operations and Tensors

`tf.Operation`

Operations

An operation maps inputs to (possibly multiple) outputs. It can also have side effects (eg. reading from memory, file system). An operation can have `attributes` (i.e. parameters that cannot be dynamically set) and be associated to a device (e.g. a specific CPU or GPU).

Operations and Tensors

`tf.Operation`

Operations

An operation maps inputs to (possibly multiple) outputs. It can also have side effects (eg. reading from memory, file system). An operation can have `attributes` (i.e. parameters that cannot be dynamically set) and be associated to a device (e.g. a specific CPU or GPU).

Order of Execution

An operation can be executed once its inputs (including control dependencies) are available. Apart from that operations are not further synchronized. An operation is run only once, we assume that the outputs are fixed once the inputs are ready. For different executions of the computation, the operation may produce different outputs for the same inputs (e.g. random ops).

Operations and Tensors

tf.Operation

Stephan Eule, Erik Schultheis

Creating Operations

Operations are usually added to the graph using constructor functions. These functions convert the inputs to tensors and validate them as far as possible. They do not return the operation, but its output, which is usually what you need. Some examples are

`tf.add`
`tf.multiply`
`tf.subtract`
`tf.divide`
`tf.matmul`
`tf.constant`

Also, the usual math functions like trigonometrics, sqrt, exp etc.

Operations and Tensors

Example: `tf.constant`

`tf.constant`

The `tf.constant` function does not return an operation but a tensor. Explicitly access its operation by the `op` attribute. This operation has zero inputs and one output. The `value` of the constant is fixed at creation time and part of the op definition.

```
>>> a = tf.constant(5)
>>> print(a)
Tensor("Const:0", shape=(), dtype=int32)
>>> print(a.op)
>>> a.op.graph == tf.get_default_graph()
True
```

Operations and Tensors

Example: `tf.constant`

The op definition

```
name: "Const"
op: "Const"
attr {
  key: "dtype"
  value { type: DT_INT32 }
}
attr {
  key: "value"
  value {
    tensor {
      dtype: DT_INT32
      tensor_shape {}
      int_val: 5
    }
  }
}
```

Operations and Tensors

`tf.Tensor`

Typed Multidimensional Array

A `tf.Tensor` `T` is a multidimensional array with a `fixed data type`. They are the outputs of operations and their `name` contains a number to mark the output index. A tensor has no `value` until the graph is executed!

TensorShape

A tensor has an associated `(static) shape` (`tf.TensorShape`). It can be partially defined and is available as `T.shape`. Upon execution, each tensor has a second (dynamic) shape which always is completely specified.

Overloaded Operators

For most python operators (e.g. +, -, *, /, **) the special methods of `tf.Tensor` are overloaded.

Operations and Tensors

Example: Arithmetic

`tf.add`

Inputs to operations have to be tensors, so any python `value` you supply to tf.add is transformed into a tensor (tf.constant).

```
>>> a = tf.add(3, 5)
<tf.Tensor 'Add:0' shape=() dtype=int32>
```

Operations and Tensors

Example: Arithmetic

`tf.add`

Inputs to operations have to be tensors, so any python `value` you supply to tf.add is transformed into a tensor (tf.constant).

```
>>> a = tf.add(3, 5)
<tf.Tensor 'Add:0' shape=() dtype=int32>
```

The result is the tensor, not the `value`!

Operations and Tensors

Example: Arithmetic

`tf.add`

Inputs to operations have to be tensors, so any python `value` you supply to tf.add is transformed into a tensor (tf.constant).

```
>>> a = tf.add(3, 5)
<tf.Tensor 'Add:0' shape=() dtype=int32>
```

The result is the tensor, not the `value`!

`tf.mul`

Arithmetic only works if the inputs have the same type.

```
>>> x = tf.constant(3)
>>> y = tf.constant(3.5)
>>> m = tf.multiply(x, y)
TypeError
```

Similar to NumPy

Tensorflow data types are almost the same as numpy's. A non-exhaustive list:

- signed integers `tf.int8`, ..., `tf.int64`
- unsigned integers `tf.uint8`, ..., `tf.uint64`
- floating point `tf.float32`, `tf.float64`
- complex numbers `tf.complex64`
- boolean `tf.bool`
- byte strings `tf.string`

Data Types

Basics

Stephan Eule, Erik Schultheis

References

The data type of a tensor can also be a reference (e.g. `tf.float32_ref`). In that case the contents of the tensor can be written to.

Quantized and half-precision

For speed (typically during inference) tensorflow also offers half precision (float16) and quantized data types.

Default Data Types

When converting python `values` to tensors they will be converted to 32 bit types (`tf.float32` for floats and `tf.int32` for integers). On most GPUs `float32` are *much* faster than their 64 bit counterparts.

Data Types

Casting

Strictness

TensorFlow is stricter about data types than numpy. You cannot combine tensors of different data type in arithmetic.

```
>>> tf.add(tf.placeholder(tf.int32), tf.placeholder(tf.float32))
Error
```

tf.cast

The tf.cast operation creates a new tensor with a given data type and the "same" value as the input.

```
>>> a = tf.placeholder(tf.int32)
>>> b = tf.placeholder(tf.float32)
>>> tf.add(tf.cast(a, tf.float32), b)
<Tensor ...>
```

Sessions

`tf.Session`

Execution

Run the graph given some *fetches* (= Tensors whose `value` you want to calculate and retrieve) and an optional *feed* dict. The `feed_dict` parameter allows any (feedable) Tensor to be given a fixed `value` so that the graph will not be traversed further.

Resources

A session object also manages resources (e.g. allocated memory). These can be temporary (`tensors` during a single run) or persistent (the values of `tf.Variables`). Therefore it is imperative to close a session after its use to free these resources again.

Sessions

`tf.Session`

Fetching Values

Replaces tensor objects with their `values` in any nested structure of dicts, lists and tuples.

Sessions

tf.Session

Fetching Values

Replaces tensor objects with their values in any nested structure of dicts, lists and tuples.

```
>>> session = tf.Session()
>>> a = tf.constant(5)
>>> b = tf.constant(8)
>>> session.run([a, b])
[5, 8]
>>> session.run({"a": a, "b": (b,)})
{"a": 5, "b": (8,)}
```

Sessions

`tf.Session`

Fetching Operations

Operations can also be part of the fetch structure. This causes them to be run. Since they have no `value` they always return None.

Sessions

`tf.Session`

Fetching Operations

Operations can also be part of the fetch structure. This causes them to be run. Since they have no `value` they always return `None`.

```
>>> session = tf.Session()
>>> x = tf.add(5, 13)
>>> session.run((x, x.op))
(13, None)
```

Sessions

tf.Session

Feeding

Feeding a tensor means that tf assumes that its `value` is readily available, so no operation has to be invoked to calculate it.

```
>>> p = tf.Print(5, [5])
>>> session.run(p) # prints 5
5
>>> session.run(p, feed_dict={p: 6}) # prints nothing
6
```

Sessions

tf.InteractiveSession

Direct evaluation

To get a single tensor or run a single operation it is possible to call `run` (for operations) or `eval` (for tensors).

```
>>> a = tf.constant(5)
>>> a.eval(session)
5
```

Sessions

tf.InteractiveSession

Direct evaluation

To get a single tensor or run a single operation it is possible to call `run` (for operations) or `eval` (for tensors).

```
>>> a = tf.constant(5)
>>> a.eval(session)
5
```

Interactive Session

If an `InteractiveSession` is used, it will be the *default session* so there is no need to specify it.

```
>>> session = tf.InteractiveSession()
>>> a = tf.constant(5)
>>> a.eval()
5
```

Your Turn

Task

Transform the following computation into a tensorflow graph. We want to have x and y as tensors.

```
x = 1 + 3
s = 2*x**2 + 5
y = x + np.sqrt(s)
```

Your Turn

Task

Transform the following computation into a tensorflow graph. We want to have x and y as tensors.

```
x = 1 + 3
s = 2*x**2 + 5
y = x + np.sqrt(s)
```

A Solution

Ensure matching data types when passing in tensors! `Python constants` will be automatically cast.

```
tf.InteractiveSession()
x = tf.add(tf.constant(1.0), tf.constant(3.0))
s = tf.add(tf.multiply(2, tf.pow(x, 2)), 5)
y = tf.add(x, tf.sqrt(s))
y.eval()
```

Your Turn

Task

Transform the following computation into tensorflow graph. We want to have x and y as tensors.

```
x = 1 + 3
s = 2*x**2 + 5
y = x + np.sqrt(s)
```

Using overloaded operators

We need at least one tf.Tensor in the expression to trigger the overloaded operator.

```
tf.InteractiveSession()
x = tf.constant(1.0) + 3
s = 2*x**2 + 5
y = x + tf.sqrt(s)
y.eval()
```

Recap

Building a Graph is like *defining* a python function, where the ops are the instructions and tensors are *immutable* local variables. Running the graph in a session is like executing the function in a python interpreter.

Recap

Building a Graph is like *defining* a python function, where the ops are the instructions and tensors are *immutable* local variables. Running the graph in a session is like executing the function in a python interpreter.

```
a = tf.constant(5)
b = tf.constant(10)
x = tf.add(a, b)
```

```
def calculate_x():
    a = 5
    b = 10
    x = a + b
    return x
```

Recap

Building a Graph is like *defining* a python function, where the ops are the instructions and tensors are *immutable* local variables. Running the graph in a session is like executing the function in a python interpreter.

```
a = tf.constant(5)
b = tf.constant(10)
x = tf.add(a, b)
```

```
def calculate_x():
    a = 5
    b = 10
    x = a + b
    return x
```

At this point, no calculations have been performed yet. For that we need to actually call the function.

```
session.run(x)
```

```
calculate_x()
```

Recap

Building a Graph is like *defining* a python function, where the ops are the instructions and tensors are *immutable* local variables. Running the graph in a session is like executing the function in a python interpreter.

```
a = tf.constant(5)
b = tf.constant(10)
x = tf.add(a, b)
```

```
def calculate_x():
    a = 5
    b = 10
    x = a + b
    return x
```

At this point, no calculations have been performed yet. For that we need to actually call the function.

```
session.run(x)
```

```
calculate_x()
```

What is missing still is the equivalent of function *arguments* and *global variables*.

Placeholders

Arguments for the Graph

A placeholder is an operation that cannot be evaluated. If its value is needed it has to be fed.

```
>>> p = tf.placeholder(tf.float32)
>>> a = tf.add(p, 1.0)
>>> a.eval()
Error
>>> a.eval(feed_dict={p: 2.0})
3.0
```

Shapes

tf.TensorShape

Partial Shapes

The most unspecific shape possible is `tf.TensorShape(None)` which can be any arbitrary shape. We can also just specify the rank `tf.TensorShape([None, None])` or single dimensions `tf.TensorShape([None, 10])`.

Guarantees on Dynamic Shapes

A static shape is just a guarantee we specify for the dynamical shape. Example: If y is a tensor of static shape `[None, 10]` and gets assigned data of shape [5, 10] everything is fine, but if we try to assign [5, 15] an error is raised.

Automatic Shape Inference

Most python functions that create operations also perform automatic shape inference. (add(a.shape=`[None, 10]`, b.shape=`[5, None]`)).shape == [5, 10]).

Rules as in numpy

Broadcasting works similarly to numpy and usually ”does the right thing”. Singleton dimensions (dimensions with size 1) will be expaned to match the dimension of the other tensor and missing dimensions are of size 1.

Rules as in numpy

Broadcasting works similarly to numpy and usually ”does the right thing”. Singleton dimensions (dimensions with size 1) will be expaned to match the dimension of the other tensor and missing dimensions are of size 1.

Examples

```
>>> a = tf.constant(1.0)
>>> b = tf.constant([1.0, 2.0, 3.0])
>>> (a+b).eval()
[2.0, 3.0, 4.0]
```

Variables

Storing values across run calls.

Not a single graph object.

A *Variable* is modeled by the `tf.Variable` class and not a single operation or tensor, but an interface for interaction with a region of memory. It can construct operations for reading and writing data and an for assigning the initial value.

Creating a Variable

A variable can be created using `tf.Variable` which needs at least an initial value. It can also be named and get its `data type` specified.

Variables

Storing values across run calls.

Assignments

To assign a a new value the assign operation can be used.

```
tf.assign(ref, value, validate_shape=None, use_locking=None, name=None)
```

To change the shape of the variable, set `validate_shape` to `False`. Assign is also available as a method of the `Variable` class.

Initialization

A variable cannot be read from until it has been assigned a value. For the first time, this is done by its initialization op. Alternatively, the initial value can be read from a save file and be assigned to the Variable. To initialize all variables at once, run the operation created by `tf.global_variables_initializer()`.

Some More Operations

Constant tensors

Operations with constant output:

```
tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
tf.fill(dims, value, name=None)
tf.constant(value, dtype=None, shape=None, name='Const',
            verify_shape=False)
```

Some More Operations

Constant tensors

Operations with constant output:

```
tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
tf.fill(dims, value, name=None)
tf.constant(value, dtype=None, shape=None, name='Const',
            verify_shape=False)
```

And for your convenience

```
tf.zeros_like(tensor, dtype=None, name=None, optimize=True)
tf.ones_like(tensor, dtype=None, name=None, optimize=True)
```

Some More Operations

Constant tensors

Operations with constant output:

```
tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
tf.fill(dims, value, name=None)
tf.constant(value, dtype=None, shape=None, name='Const',
            verify_shape=False)
```

And for your convenience

```
tf.zeros_like(tensor, dtype=None, name=None, optimize=True)
tf.ones_like(tensor, dtype=None, name=None, optimize=True)
```

Why not just tf.constant?

Some More Operations

Constant tensors

Operations with constant output:

```
tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
tf.fill(dims, value, name=None)
tf.constant(value, dtype=None, shape=None, name='Const',
            verify_shape=False)
```

And for your convenience

```
tf.zeros_like(tensor, dtype=None, name=None, optimize=True)
tf.ones_like(tensor, dtype=None, name=None, optimize=True)
```

Why not just tf.constant? Because then all those zeros/ones need to be saved inside an `attribute` of the op, whereas tf.zeros can create them on the fly. Also readability.

Some More Operations

Reductions

Performing operations over all elements of one or more dimensions. They all share the same function signature. For numerical data

```
tf.reduce_sum(input_tensor, axis=None, keep_dims=False, name=None)
tf.reduce_prod, tf.reduce_max, tf.reduce_min,
tf.reduce_logsumexp
```

and for booleans / strings

```
tf.reduce_any, tf.reduce_all
tf.reduce_join
```

Linear Model

Forward pass

We now have everything to build the forward pass of a simple linear model.

```
W = tf.Variable(np.random((10, 784)))
b = tf.Variable(np.random(10))
x = tf.placeholder(tf.float32, (None, 784))
l = tf.matmul(W, x) + b
```

Linear Model

Forward pass

We now have everything to build the forward pass of a simple linear model.

```
W = tf.Variable(np.random((10, 784)))
b = tf.Variable(np.random(10))
x = tf.placeholder(tf.float32, (None, 784))
l = tf.matmul(W, x) + b
```

Now we need to convert this into a probability distribution over classes, and a *loss* function as optimization objective. Typical in classification tasks: *softmax* nonlinearity and *cross-entropy* loss.

Linear Model

Classification

Classification task: Exactly one of k classes is true.

Softmax

$$\mathrm{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{k} \exp(x_j)}.$$

Cross-Entropy

Let $p(i)$ be the true probability of class i and $q(i)$ be the predicted probability. The cross-entropy between the two distributions is

$$X(p, q) = \sum_{i=1}^{k} p(i) \log q(i).$$

"How good can we compress data distributed $\sim p$ if we assume it is distributed $\sim q$"

Linear Model

Loss Function

```
y = tf.placeholder(tf.float32, (None, 10))
p = tf.nn.softmax(l)
loss = tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=l)
```

Some Notes

- The function takes in logits instead of probabilities for numerical stability.
- Labels need not be one-hot vectors, but can be arbitrary probablity distributions over classes.

Stochastic Gradient Descent

Optimizing Differentiable Functions

Gradient Descent

Function f of input dataset $X = (x_1, \ldots, x_n)$ and *weights* θ, loss L and target values $Y = (y_1, \ldots, y_n)$, learning rate α

$$\theta \leftarrow \theta + \alpha \nabla_\theta L(f(X;\theta), Y) \tag{3}$$

Estimating from Minibatches

$$\nabla_\theta L(f(X;\theta), Y) = \sum_{i=1}^{n} \nabla_\theta L(f(x_i;\theta), y_i) = \frac{n}{k} \cdot \mathbb{E}\left[\sum_{i=1}^{k} \nabla_\theta L(f(x_{j_i};\theta), y_{j_i})\right]$$

For uniformly sampled j_i we can use the rhs. as an unbiased estimator for the true gradient. $\Rightarrow$ *Stochastic Gradient Descent*.

Optimizers

This Could be Your Slide about Backpropagation ...

Autodifferentiation is Your Friend

As long as each operation used in the mapping from input to loss tensor is *differentiable* we need not worry about calculating the gradient of the chained operation.

Minimizing the Loss

```
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)
```

The result is an operation that performs a single gradient descent step when run. This means calculating the gradients and updating the trainable variables.

The Full Network

Network Code

```
x = tf.placeholder(tf.float32, (None, 784))
y = tf.placeholder(tf.float32, (None, 10))
W = tf.Variable(np.random((10, 784)))
b = tf.Variable(np.random(10))
l = tf.matmul(W, x) + b
p = tf.nn.softmax(l)
loss = tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=l)
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)
```

Getting Data

```
import tensorflow.contrib.learn as tflearn
mnist = tflearn.datasets.load_dataset("mnist")
images = mnist.train.images
labels = mnist.train.labels
```

Basic Linear Model

Notebook (01 Linear Model.ipynb)

Summaries

Gather Monitoring Data inside the Graph

Add graph operations responsible for logging (*summaries*).

Motivation

- On-line monitoring of progress (e.g. loss, accuracy) and network internals (e.g. weight norms, regularization losses)
- Feed-dict does not scale well to many summaries.
- High level debugging. See that input images are correctly preprocessed, gradient norms remain reasonable etc.

Summary Operations

Take in a Tensor and produce a string summary (which is a serialized protobuf object). These can be collected and written to a summary file.

Summaries

Gather Monitoring Data inside the Graph

Summary Types

The following summaries are available in the `tf.summary` module.

scalar A single real value, e.g. the loss.

histogram Histogram of the values of a Tensor, e.g. to visualize distributions of activations.

image An image $[\mathrm{Batch} \times \mathrm{Height} \times \mathrm{Width} \times \mathrm{Channels}]$ tensor.

audio Audio data in format $[\mathrm{Batch} \times \mathrm{Frames} \times \mathrm{Channels}]$ or $[\mathrm{Batch} \times \mathrm{Frames}]$ in range $[-1.0, 1.0]$.

text A string tensor representing textual data.

Each summary takes at least two arguments: A `name` (or tag) for the operation, and the tensor to summarize.

Summaries

Gather Monitoring Data inside the Graph

Merging Summaries

Since summaries are operations, we need to explictly pass them as fetches to generate summary values. To make this more usable, multiple summaries can be merged into a single summary. This can be done for an explicit list of summaries (`tf.summary.merge`) or for all summaries in the graph (`tf.summary.merge_all`).

Example

```
a = tf.summary.scalar("loss", loss)
b = tf.summary.scalar("accuracy", accuracy)
s = tf.summary.merge_all() # = tf.summary.merge([a, b])
```

To generate the summaries during training:

```
_, summary = session.run([train_step, s], feed_dict=feed_dict)
```

Gets both ”loss” and ”accuracy” summaries.

Summary Writer

Saving Summaries

FileWriter

A `tf.summary.FileWriter` is responsible for writing log *events* to a file. Events can be summaries, graphs, session logs, run metadata etc.

```
tf.summary.FileWriter(logdir, graph, max_queue, flush_secs, filename_suffix)
```

Creates a new summary file with a unique name inside `logdir` and writes a graph event to it if `graph` is supplied.

Directories Group Runs

By convention **all** event files within a single directory are assumed to be from the same *run*. These containing events will be grouped together.

Summary Writer

Saving Summaries

Adding Events to Summary Files

To add a summary one needs the (serialized) Protocol Buffer and an associated *global step*.

```
summary = session.run(summaries)
writer.add_summary(summary, global_step)
```

While the step is typically gathered from a global_step variable, the summary writer accepts any `python integer`. The step is used to form a time axis for your log data.

Writes Are Asynchronous

To prevent training slowdowns by summary writes, the `FileWriter` writes only asynchronously to the summary file. This can be controlled with the `max_queue` and `flush_secs` parameters.

Some More Operations

Comparisons

Comparison Operations

Compare tensors element-wise

```
tf.equal(x, y, name=None)
```

and analogly

```
tf.less, tf.less_equal, tf.greater, tf.greater_equal
```

Some More Operations

Comparisons

Comparison Operations

Compare tensors element-wise

`tf.equal(x, y, name=None)`

and analogly

`tf.less`, `tf.less_equal`, `tf.greater`, `tf.greater_equal`

Booleans to numbers

`tf.cast(bool_tensor, data_type)`

converts every `True` to 1.0 and every `False` to 0.0.

Calculating Accuracy

```
is_correct = tf.equal(tf.argmax(logits, axis=1), y)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float64))
```

Linear Model with Summaries

Notebook (02 Summaries.ipynb)

Tensorboard

Tensorboard Demo

tf.train.Saver

Writing Variables to Files

The Saver class is responsible for building operations that save and restore Variables to/from a checkpoint file.

```
tf.train.Saver(var_list=None, reshape=False, max_to_keep=5,
keep_checkpoint_every_n_hours=10000.0,
name=None, restore_sequentially=False, saver_def=None,
builder=None, defer_build=False, allow_empty=False,
save_relative_paths=False, filename=None)
```

var_list List (or dictionary) of Variables to save. If None is submitted all *global variables* are saved.

max_to_keep How many checkpoint files to keep before starting to delete older ones.

Creating a Saver does not yet save anything!

tf.train.Saver

Writing Variables to Files

Saving to a Checkpoint

Create a Saver object **after** the model has been build. Then

```
saver.save(session, save_path, global_step=None, latest_filename=None)
```

This saves the model to `save_path`, and if `global_step` is supplied also registers the new checkpoint in the *latest checkpoints file* (named `"checkpoint"` or `latest_filename`).

Step Counting

Save step count inside a tf.Variable to have consistent counts across checkpoints. Either manually or using

```
global_step = tf.Variable(0, dtype=tf.int64, trainable=False)
global_step = tf.train.create_global_step()
```

Automatically increment the global step for each optimization step.

```
train_step = optimizer.minimize(loss, global_step=global_step)
```

tf.train.Saver

Restoring

Loading a Checkpoint

To load the values into an *already existing* model do

```
saver.restore(session, save_path)
```

To also restore the graph call create your saver as

```
new_saver = tf.train.import_meta_graph(meta_graph_file_name)
```

Finding the Correct Checkpoint File

To restore you need the complete checkpoint file name, including the step suffix. This can be found using the `latest_checkpoint` function:

```
checkpoint = tf.train.latest_checkpoint(checkpoint_dir ,latest_filename=None)
saver.restore(session, checkpoint)
```

Linear Model with Checkpoints

Notebook (03 Checkpoints.ipynb)

Useful Python Features

Context Managers

Allows do some code in a certain context by executing an `__enter__` function when the context starts and an `__exit__` function when the context ends. For example:

```
with open("filename") as file:
    # do sth with the file
# here, the file will be close again
```

This is used extensively by tensorflow for default sessions, default graphs, devices, scoping etc.

Named Tuples

Makes a helper type that behaves like a tuple, but can be indexed with named keys.

```
NamedTuple = namedtuple("NamedTuple", ("a", "b"))
data = NamedTuple(a=5, b="test")
```

Default Session and Default Graph

Instead of

```
# build your model here
session = tf.Session()
# your main loop here
session.close()
```

Problems

- Risks forgetting to close session. Leaks in case of exception.
- Pollutes global default graph.

Do

```
graph = tf.Graph()
with graph.as_default():
    # build your model here
with tf.Session(graph=graph) as session:
    # Your main loop here
```

Advantages

- Guaranteed cleanup.
- Can easily create multiple graphs and runs in a single program without interference.

Model Function

Separating Model Definition and Training Routine

Instead of building the model inside the default graph, put the model into its own function. Keep the model inputs as arguments.

```
Model = namedtuple("Model", ("loss", "train_step"))

def model_fn(x, y):
    W = tf.Variable(np.random((10, 784)))
    b = tf.Variable(np.random(10))
    l = tf.matmul(W, x) + b
    labels = tf.one_hot(y, depth=10)
    loss = tf.nn.softmax_cross_entropy_with_logits(labels=labels, logits=l)
    loss = tf.reduce_mean(loss)
    train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)
    return Model(loss=loss, train_step=train_step)
```

Model Function

Separating Model Definition and Training Routine

Use as follows:

```python
graph = tf.Graph()
with graph.as_default():
    x = tf.placeholder(tf.float32, (None, 784), name="x")
    y = tf.placeholder(tf.int64, (None), name="y")
    loss, train_step = model_fn(x, y)
    init_op = tf.global_variables_initializer()

with tf.Session(graph=graph) as session:
    init_op.run()
    # Your main loop here
```

Scoping

Structure in Operation Names

Name Scopes

Every operation created within a name scope will have its name prefixed by that scope name.

```
with tf.name_scope("prefix"):
    a = tf.add(5, 4)
print(a.name) # "prefix/Add:0"
```

Nesting

Name scopes stack.

```
with tf.name_scope("prefix"):
    with tf.name_scope("inner"):
        a = tf.add(5, 4)
print(a.name) # "prefix/inner/Add:0"
```

Scoping

Structure in Operation Names

Re-Opening Name Scopes

Using the same name scope again will create a new, unique prefix

```
with tf.name_scope("prefix"):
    a = tf.add(5, 4)
with tf.name_scope("prefix"):
    a = tf.add(5, 4)
print(a.name) # "prefix_1/Add:0"
```

But you can remember a scope and reuse it as an absolute scope.

```
with tf.name_scope("prefix") as prefix_scope:
    pass
with tf.name_scope("other"):
    with tf.name_scope(prefix_scope):
        a = tf.add(5, 4)
print(a.name) # "prefix/Add:0"
```

Scoping
Variable Reuse

Weight Sharing

In some models we want to use the same weights in different parts of the computation (e.g. in a GAN). This can be achieved using *variable scopes*.

Variable Scope

A variable scope sets the name scope in which variables are created or looked up. Use in conjunction with `tf.get_variable` which either gets an existing variable or creates a new variable. Opening a variable scope of the same name again will open the *exact same scope*

```
with tf.variable_scope("S"):
    tf.get_variable("a", shape=()).name # S/a:0
with tf.variable_scope("S"):
    tf.get_variable("b", shape=()).name # S/b:0
```

Scoping
Variable Reuse

Reusing a Scope

Inside a variable scope you can either create new variables **or** reuse existing ones, never both:

```
with tf.variable_scope("S"):
    tf.get_variable("a", shape=()).name # S/a:0
with tf.variable_scope("S"):
    tf.get_variable("a", shape=()).name # ERROR
with tf.variable_scope("S", reuse=True):
    tf.get_variable("a").name # S/a:0
```

Entering a non-reusing scope as subscope inside a reusing one is not possible, and neither is the reverse:

```
with tf.variable_scope("S", reuse=True):
    with tf.variable_scope("T", reuse=False)
        pass # ERROR
```

Variable Scopes and Name Scopes

Entering a Variable Scope automatically enters a name scope of the same name.

```
with tf.name_scope("other") as other_scope:
    pass
with tf.variable_scope("S"):
    tf.get_variable("a", shape=())
with tf.variable_scope("S", reuse=True):
    tf.add(5, 4) # S_1/Add:0
    with tf.name_scope(other_scope):
        a = tf.get_variable("a", shape=()) # S/a:0
        tf.add(a, 4) # other/Add:0
```

get_variable

A Imporved Interface to Variables

`get_variable` acts as a Variable constructor that respects the current variable scope.

```
tf.get_variable(name, shape=None, dtype=None, initializer=None,
                regularizer=None, trainable=True, collections=None,
                caching_device=None, partitioner=None,
                validate_shape=True, use_resource=None,
                custom_getter=None, constraint=None)
```

Instead of an initial value, an `initializer` has to be passed. This is a function that takes in the desired shape and data type and produces the initial value. The `regularizer` is a function that takes the variables value and outputs a *regularization* loss, and `constraint` maps the variables value to a constrained value.

Graph Building Functions

Passing Around Recipies for Subgraphs

Building Functions

Passing around `building functions` instead of pre-built tensors allows to build computations in the correct context.

initializer The initial value is calculated in the Variable's name scope.

constraint The constraint should be applied after an update to the variable, but this has not happened yet at construction time.

Another Level of Indirection

$$\text{building function} \xrightarrow{\text{build}} \text{computation graph} \xrightarrow{\text{run}} \text{values}$$

Initializer

Building Functions for Initial Values

The following initializers are available in tensorflow:

```
zeros(dtype=tf.float32)
ones(dtype=tf.float32)
constant(value=0, dtype=tf.float32, verify_shape=False, dtype=tf.float32)
random_uniform(minval=0, maxval=None, seed=None, dtype=tf.float32)
random_normal(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)
truncated_normal(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)
variance_scaling(scale=1.0, mode="fan_in", distribution="normal",
                seed=None, dtype=tf.float32)
orthogonal(gain=1.0, seed=None, dtype=tf.float32)
identity(gain=1.0, dtype=tf.float32)
```

The functions are available as `tf.*_initializer` or `tf.initializers.*`

Random Operations

Deterministic Pseudo-Random Numbers

To get deterministic pseudo-random numbers, a (graph level) random seed has to be set by `tf.set_random_seed`. In addition, each random operation has its own seed. The randomization is then as follows

both not set A random seed is generated for each run and operation.

graph seed is set Operation seeds are picked deterministically from the graph seed.

operation seed set Use a default graph seed together with the operation seed.

both set Combine graph and operation seed.

Random Operations

List of Ops

Drawing from a Distribution

All operations below also do have a `seed` and a `name` argument.

```
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32)
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32)
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32)
tf.random_gamma(shape, alpha, beta=None, dtype=tf.float32)
tf.multinomial(logits, num_samples)
```

Modify Existing Values

```
tf.random_shuffle(value, seed=None, name=None)
tf.random_crop(value, size, seed=None, name=None)
```

Graph Collections

Categorizing Ops and Tensors

Functions like `merge_all_summaries`, `global_variables_initializer`, etc need to find the graph elements that they should operate on. This is facilitated by graph collections.

Graph Collection

A graph collection is a list of graph elements that is registered under a certain name in the graph.

```
tf.get_collection(key, scope=None)
tf.add_to_collection(name, value)
```

These collections then allow to categorize tensors according to their purpose.

Graph Collections

Standard Collections

By default tensorflow uses the among others following collection keys (all defined in `tf.GraphKeys`)

GLOBAL_VARIABLES All model weights, global step etc.

TRAINABLE_VARIABLES Variables that will be updated by the optimizer.

SUMMARIES All summaries.

REGULARIZATION_LOSSES All regularization losses.

More keys are only used by subsystems

GLOBAL_STEP The global step variable when used with `tf.train`.

LOSSES Losses built with `tf.losses`

WEIGHTS, BIASES Kernels and biases of `tf.layers`

Linear Model

Notebook (04 Structure.ipynb)

Layers Interface

Quickly Building Sequential Models

The functions in `tf.layers` take an input value and build a complete neural network *layer* that transforms the value. This includes inferring the shapes of the invoved tensors and creating or reusing variables. There are layers for

- Flattening
- Dense (fully connected) multiplication
- Convolution
- Max Pooling
- Dropout
- Batch Normalization

Layers Interface

Quickly Building Sequential Models

The functions in `tf.layers` take an input value and build a complete neural network *layer* that transforms the value. This includes inferring the shapes of the invoved tensors and creating or reusing variables. There are layers for

- Flattening
- Dense (fully connected) multiplication
- Convolution
- Max Pooling
- Dropout
- Batch Normalization

If your favourite layer is not listed here, the necessary primitives might still be in `tf.nn`, or it might be in `tf.contrib`.

Dense Layer

Argument Overview

Most layer functions take a vast amount of arguments. There are three per variable for *initializer*, *regularizer* and *constraint*.

```
tf.layers.dense(
    inputs,
    units,
    activation=None,
    use_bias=True,
    kernel_initializer=None,
    bias_initializer=tf.zeros_initializer(),
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    trainable=True,
    name=None,
    reuse=None
)
```

Dense Layer

Argument Overview

Most layer functions take a vast amount of arguments. There are three per variable for *initializer*, *regularizer* and *constraint*.

```
tf.layers.dense(
    inputs,
    units,
    activation=None,
    use_bias=True,
    activity_regularizer=None,
    trainable=True,
    name=None,
    reuse=None
)
```

`name`, `reuse` and `trainable` are also passed for the variables and determine variable scope, reuse and trainability. `activity_regularizer` adds a regularization loss depending on the layers output.

Dense Layer

The main interface of the layer is

```
tf.layers.dense(inputs, units, activation=None, use_bias=True)
```

Arguments

inputs Input tensor x of shape `[Batch Size, Input Size]`

units Number of output elements.

activation An activation function σ that is applied to the outputs. `None` means linear activation.

use_bias whether to add a bias b to the result.

Calculation

$$o = \sigma(Wx + b) \tag{4}$$

Activation Functions

```
tf.nn.relu, tf.nn.sigmoid, tf.nn.tanh, tf.nn.softmax, ...
```

Multilayer Perceptron

The model function of a multilayer perceptron network is now

```
def mlp_fn(x, hidden_units=(50, 30, 10)):
    hidden = x
    for units in hidden_units:
        hidden = tf.layers.dense(hidden, units, tf.nn.sigmoid)
    return hidden
```

Convolution

The main interface of the 2d convolution is

```
tf.layers.conv2d(inputs, filters, kernel_size, strides=(1, 1),
                 padding='valid', data_format='channels_last',
                 dilation_rate=(1, 1), activation=None, use_bias=True)
```

Arguments

- **filters** Number of output channels
- **kernel_size** Size of the receptive field.
- **strides** Step size for striding.
- **padding** Either "valid" (no padding) or "same"
- **data_format** Either "channels_first" or "channels_last"
- **dilation_rate** For dilated convultions.

Convolutional Classifier

The model has a few convolutional layers, followed by a fully connected classifier.

```
def cnn_fn(x, channels=(32, 64), outputs=10):
    hidden = x
    for c in channels:
        hidden = tf.layers.conv2d(hidden, c, kernel_size=3, strides=2,
                                  activation=tf.nn.relu)
    hidden = tf.layers.flatten(hidden)
    return tf.layers.dense(hidden, outputs)
```

Dropout

The *dropout* layer behaves differently in training mode compared to evaluation/prediction mode.

```
tf.layers.dropout(inputs, rate=0.5, noise_shape=None, seed=None,
                  training=False, name=None)
```

Arguments

- **rate** Fraction of values to drop.
- **noise_shape** Shape of the dropout mask.
- **training** If `True`, drops out `rate` values and rescales by `rate`$^{-1}$, otherwise does nothing.

Instead of a dynamic training value, we can use an additional parameter to our `model_fn` to build the graph either in training or in inference mode.

Convolutional Classifier with Dropout

The model has a few convolutional layers, followed by a fully connected classifier. Performs dropout in training mode.

```
def cnn_fn(x, channels=(32, 64), outputs=10, is_training=True):
    hidden = x
    for c in channels:
        hidden = tf.layers.conv2d(hidden, c, kernel_size=3, strides=2,
                                  activation=tf.nn.relu)
    hidden = tf.layers.flatten(hidden)
    hidden = tf.layers.dropout(hidden, 0.5, training=is_training)
    return tf.layers.dense(hidden, outputs)
```

Losses

Utilities for Defining Loss Functions

A higher level interface to typical losses is given in `tf.losses`. These add (optional) name scoping and a unified interface for weighted losses. The following losses are available:

- Absolute Difference
- Cosine Distance
- Hinge Loss
- Huber Loss
- Log Loss
- (sigmoid/Softmax) Cross Entropy
- Mean Squared Error

Mean Squared Error

Interface

```
tf.losses.mean_squared_error(labels, predictions, weights=1.0,
                             scope=None,
                             loss_collection=tf.GraphKeys.LOSSES,
                             reduction=Reduction.SUM_BY_NONZERO_WEIGHTS)
```

Arguments

weights Weights for the losses. Either a single scalar, of of the same shape as `labels`.

scope Name scope in which all computations will be put. If None use current scope.

loss_collection GraphCollection where the loss is registered.

reduction How the loss is reduced to a single scalar. Can be `NONE`, `MEAN`, `SUM` or `SUM_BY_NONZERO_WEIGHTS`.

The other losses work analogly.

Regularization Losses

Gathering Additional Loss Functions

Getting Regularization Losses

Each Variable with a *regularizer* registers the resulting loss. These losses can be retrieved either individually

```
tf.losses.get_regularization_losses(scope=None)
```

or as a total

```
tf.losses.get_regularization_loss(scope=None)
```

Regularizers

Typical regularization functions are e.g.

```
tf.contrib.layers.l1_regularizer(scale, scope=None)
tf.contrib.layers.l2_regularizer(scale, scope=None)
```

Model Function Example