↳ 1 cells hidden

Creating and Manipulating Tensors

Learning Objectives:

- Initialize and assign TensorFlow `Variables`
- Create and manipulate tensors
- Refresh your memory about addition and multiplication in linear algebra (consult an introduction to matrix addition and multiplication if these topics are new to you)
- Familiarize yourself with basic TensorFlow math and array operations

```
import tensorflow as tf
```

Vector Addition

You can perform many typical mathematical operations on tensors (TF API). The following code creates and manipulates two vectors (1-D tensors), each having exactly six elements:

```
with tf.Graph().as_default():
  # Create a six-element vector (1-D tensor).
  primes = tf.constant([2, 3, 5, 7, 11, 13], dtype=tf.int32)

  # Create another six-element vector. Each element in the vector will be
  # initialized to 1. The first argument is the shape of the tensor (more
  # on shapes below).
  ones = tf.ones([6], dtype=tf.int32)

  # Add the two vectors. The resulting tensor is a six-element vector.
  just_beyond_primes = tf.add(primes, ones)

  # Create a session to run the default graph.
  with tf.Session() as sess:
    print just_beyond_primes.eval()
```

Tensor Shapes

Shapes are used to characterize the size and number of dimensions of a tensor. The shape of a tensor is expressed as `list`, with the `i`th element representing the size along dimension `i`. The length of the list then indicates the rank of the tensor (i.e., the number of dimensions).

For more information, see the TensorFlow documentation.

A few basic examples:

```
with tf.Graph().as_default():
  # A scalar (0-D tensor).
  scalar = tf.zeros([])

  # A vector with 3 elements.
  vector = tf.zeros([3])

  # A matrix with 2 rows and 3 columns.
  matrix = tf.zeros([2, 3])

  with tf.Session() as sess:
    print 'scalar has shape', scalar.get_shape(), 'and value:\n', scalar.eval()
    print 'vector has shape', vector.get_shape(), 'and value:\n', vector.eval()
    print 'matrix has shape', matrix.get_shape(), 'and value:\n', matrix.eval()
```

Broadcasting

In mathematics, you can only perform element-wise operations (e.g. *add* and *equals*) on tensors of the same shape. In TensorFlow, however, you may perform operations on tensors that would traditionally have been incompatible. TensorFlow supports **broadcasting** (a concept borrowed from numpy), where the smaller array in an element-wise operation is enlarged to have the same shape as the larger array. For example, via broadcasting:

- If an operand requires a size `[6]` tensor, a size `[1]` or a size `[]` tensor can serve as an operand.
- If an operation requires a size `[4, 6]` tensor, any of the following sizes can serve as an operand:
  - `[1, 6]`
  - `[6]`
  - `[]`
- If an operation requires a size `[3, 5, 6]` tensor, any of the following sizes can serve as an operand:
  - `[1, 5, 6]`
  - `[3, 1, 6]`
  - `[3, 5, 1]`
  - `[1, 1, 1]`
  - `[5, 6]`
  - `[1, 6]`
  - `[6]`
  - `[1]`
  - `[]`

NOTE: When a tensor is broadcast, its entries are conceptually **copied**. (They are not actually copied for performance reasons. Broadcasting was invented as a performance optimization.)

The full broadcasting ruleset is well described in the easy-to-read [numpy broadcasting documentation].

The following code performs the same tensor addition as before, but using broadcasting:

```
with tf.Graph().as_default():
  # Create a six-element vector (1-D tensor).
  primes = tf.constant([2, 3, 5, 7, 11, 13], dtype=tf.int32)

  # Create a constant scalar with value 1.
  ones = tf.constant(1, dtype=tf.int32)

  # Add the two tensors. The resulting tensor is a six-element vector.
  just_beyond_primes = tf.add(primes, ones)

  with tf.Session() as sess:
    print just_beyond_primes.eval()
```

Matrix Multiplication

In linear algebra, when multiplying two matrices, the number of *columns* of the first matrix must equal the number of *rows* in the second matrix.

- It is ***valid*** to multiply a 3x4 matrix by a 4x2 matrix. This will result in a 3x2 matrix.
- It is ***invalid*** to multiply a 4x2 matrix by a 3x4 matrix.

```
with tf.Graph().as_default():
  # Create a matrix (2-d tensor) with 3 rows and 4 columns.
  x = tf.constant([[5, 2, 4, 3], [5, 1, 6, -2], [-1, 3, -1, -2]],
                  dtype=tf.int32)

  # Create a matrix with 4 rows and 2 columns.
  y = tf.constant([[2, 2], [3, 5], [4, 5], [1, 6]], dtype=tf.int32)

  # Multiply `x` by `y`.
  # The resulting matrix will have 3 rows and 2 columns.
  matrix_multiply_result = tf.matmul(x, y)

  with tf.Session() as sess:
    print matrix_multiply_result.eval()
```

Tensor Reshaping

With tensor addition and matrix multiplication each imposing constraints on operands, TensorFlow programmers must frequently reshape tensors.

You can use the `tf.reshape` method to reshape a tensor. For example, you can reshape a 8x2 tensor into a 2x8 tensor or a 4x4 tensor:

```
with tf.Graph().as_default():
  # Create an 8x2 matrix (2-D tensor).
  matrix = tf.constant([[1,2], [3,4], [5,6], [7,8],
                        [9,10], [11,12], [13, 14], [15,16]], dtype=tf.int32)

  # Reshape the 8x2 matrix into a 2x8 matrix.
  reshaped_2x8_matrix = tf.reshape(matrix, [2,8])

  # Reshape the 8x2 matrix into a 4x4 matrix
  reshaped_4x4_matrix = tf.reshape(matrix, [4,4])

  with tf.Session() as sess:
    print "Original matrix (8x2):"
    print matrix.eval()
    print "Reshaped matrix (2x8):"
    print reshaped_2x8_matrix.eval()
    print "Reshaped matrix (4x4):"
    print reshaped_4x4_matrix.eval()
```

You can also use `tf.reshape` to change the number of dimensions (the "rank") of the tensor. For example, you could reshape that 8x2 tensor into a 3-D 2x2x4 tensor or a 1-D 16-element tensor.

```
with tf.Graph().as_default():
  # Create an 8x2 matrix (2-D tensor).
  matrix = tf.constant([[1,2], [3,4], [5,6], [7,8],
                        [9,10], [11,12], [13, 14], [15,16]], dtype=tf.int32)

  # Reshape the 8x2 matrix into a 3-D 2x2x4 tensor.
  reshaped_2x2x4_tensor = tf.reshape(matrix, [2,2,4])

  # Reshape the 8x2 matrix into a 1-D 16-element tensor.
  one_dimensional_vector = tf.reshape(matrix, [16])

  with tf.Session() as sess:
    print "Original matrix (8x2):"
    print matrix.eval()
    print "Reshaped 3-D tensor (2x2x4):"
    print reshaped_2x2x4_tensor.eval()
    print "1-D vector:"
    print one_dimensional_vector.eval()
```

Exercise #1: Reshape two tensors in order to multiply them.

The following two vectors are incompatible for matrix multiplication:

- `a = tf.constant([5, 3, 2, 7, 1, 4])`
- `b = tf.constant([4, 6, 3])`

Reshape these vectors into compatible operands for matrix multiplication. Then, invoke a matrix multiplication operation on the reshaped tensors.

```
  # Write your code for Task 1 here.
```

Solution

Click below for a solution.

```
with tf.Graph().as_default(), tf.Session() as sess:
  # Task: Reshape two tensors in order to multiply them

  # Here are the original operands, which are incompatible
  # for matrix multiplication:
  a = tf.constant([5, 3, 2, 7, 1, 4])
  b = tf.constant([4, 6, 3])
  # We need to reshape at least one of these operands so that
  # the number of columns in the first operand equals the number
  # of rows in the second operand.

  # Reshape vector "a" into a 2-D 2x3 matrix:
  reshaped_a = tf.reshape(a, [2,3])

  # Reshape vector "b" into a 2-D 3x1 matrix:
  reshaped_b = tf.reshape(b, [3,1])

  # The number of columns in the first matrix now equals
  # the number of rows in the second matrix. Therefore, you
  # can matrix mutiply the two operands.
  c = tf.matmul(reshaped_a, reshaped_b)
  print(c.eval())

  # An alternate approach: [6,1] x [1, 3] -> [6,3]
```

Variables, Initialization and Assignment

So far, all the operations we performed were on static values (`tf.constant`); calling `eval()` always returned the same result. TensorFlow allows you to define `Variable` objects, whose values can be changed.

When creating a variable, you can set an initial value explicitly, or you can use an initializer (like a distribution):

```
g = tf.Graph()
with g.as_default():
  # Create a variable with the initial value 3.
  v = tf.Variable([3])

  # Create a variable of shape [1], with a random initial value,
  # sampled from a normal distribution with mean 1 and standard deviation 0.35.
  w = tf.Variable(tf.random_normal([1], mean=1.0, stddev=0.35))
```

One peculiarity of TensorFlow is that **variable initialization is not automatic**. For example, the following block will cause an error:

```
with g.as_default():
  with tf.Session() as sess:
    try:
      v.eval()
    except tf.errors.FailedPreconditionError as e:
      print "Caught expected error: ", e
```

The easiest way to initialize a variable is to call `global_variables_initializer`. Note the use of `Session.run()`, which is roughly equivalent to `eval()`.

```
with g.as_default():
  with tf.Session() as sess:
    initialization = tf.global_variables_initializer()
    sess.run(initialization)
    # Now, variables can be accessed normally, and have values assigned to them.
    print v.eval()
    print w.eval()
```

Once initialized, variables will maintain their value within the same session (however, when starting a new session, you will need to re-initialize them):

```
with g.as_default():
  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # These three prints will print the same value.
    print w.eval()
    print w.eval()
    print w.eval()
```

To change the value of a variable, use the `assign` op. Note that simply creating the `assign` op will not have any effect. As with initialization, you have to `run` the assignment op to update the variable value:

```
with g.as_default():
  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # This should print the variable's initial value.
    print v.eval()

    assignment = tf.assign(v, [7])
    # The variable has not been changed yet!
    print v.eval()

    # Execute the assignment op.
    sess.run(assignment)
    # Now the variable is updated.
    print v.eval()
```

There are many more topics about variables that we didn't cover here, such as loading and storing. To learn more, see the TensorFlow docs.

Exercise #2: Simulate 10 rolls of two dice.

Create a dice simulation, which generates a `10x3` 2-D tensor in which:

- Columns `1` and `2` each hold one throw of one six-sided die (with values 1–6).
- Column `3` holds the sum of Columns `1` and `2` on the same row.

For example, the first row might have the following values:

- Column `1` holds `4`
- Column `2` holds `3`
- Column `3` holds `7`

You'll need to explore the TensorFlow documentation to solve this task.

```
# Write your code for Task 2 here.
```

Solution

Click below for a solution.

```
with tf.Graph().as_default(), tf.Session() as sess:
  # Task 2: Simulate 10 throws of two six-sided dice. Store the results
  # in a 10x3 matrix.

  # We're going to place dice throws inside two separate
  # 10x1 matrices. We could have placed dice throws inside
  # a single 10x2 matrix, but adding different columns of
# the same matrix is tricky. We also could have placed
# dice throws inside two 1-D tensors (vectors); doing so
# would require transposing the result.
dice1 = tf.Variable(tf.random_uniform([10, 1],
                                      minval=1, maxval=7,
                                      dtype=tf.int32))
dice2 = tf.Variable(tf.random_uniform([10, 1],
                                      minval=1, maxval=7,
                                      dtype=tf.int32))

# We may add dice1 and dice2 since they share the same shape
# and size.
dice_sum = tf.add(dice1, dice2)

# We've got three separate 10x1 matrices. To produce a single
# 10x3 matrix, we'll concatenate them along dimension 1.
resulting_matrix = tf.concat(
    values=[dice1, dice2, dice_sum], axis=1)

# The variables haven't been initialized within the graph yet,
# so let's remedy that.
sess.run(tf.global_variables_initializer())

print(resulting_matrix.eval())
```