creating_and_manipulating_tensors

April 24, 2019

1 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

```
In [0]: from __future__ import print_function

        import tensorflow as tf
        try:
          tf.contrib.eager.enable_eager_execution()
          print("TF imported with eager execution!")
        except ValueError:
          print("TF already imported with eager execution!")
```

1.1 Vector Addition

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

- A `primes` vector containing prime numbers.
- A `ones` vector containing all `1` values.

- A vector created by performing element-wise addition over the first two vectors.
- A vector created by doubling the elements in the `primes` vector.

```
In [0]: primes = tf.constant([2, 3, 5, 7, 11, 13], dtype=tf.int32)
        print("primes:", primes
        )
        ones = tf.ones([6], dtype=tf.int32)
        print("ones:", ones)

        just_beyond_primes = tf.add(primes, ones)
        print("just_beyond_primes:", just_beyond_primes)

        twos = tf.constant([2, 2, 2, 2, 2, 2], dtype=tf.int32)
        primes_doubled = primes * twos
        print("primes_doubled:", primes_doubled)
```

Printing a tensor returns not only its value, but also its shape (discussed in the next section) and the type of value stored in the tensor. Calling the `numpy` method of a tensor returns the value of the tensor as a numpy array:

```
In [0]: some_matrix = tf.constant([[1, 2, 3], [4, 5, 6]], dtype=tf.int32)
        print(some_matrix)
        print("\nvalue of some_matrix is:\n", some_matrix.numpy())
```

1.1.1 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:

```
In [0]: # 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])

        print('scalar has shape', scalar.get_shape(), 'and value:\n', scalar.numpy())
        print('vector has shape', vector.get_shape(), 'and value:\n', vector.numpy())
        print('matrix has shape', matrix.get_shape(), 'and value:\n', matrix.numpy())
```

1.1.2 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 arithmetic as before, but instead uses scalar values (instead of vectors containing all 1s or all 2s) and broadcasting.

```
In [0]: primes = tf.constant([2, 3, 5, 7, 11, 13], dtype=tf.int32)
        print("primes:", primes)

        one = tf.constant(1, dtype=tf.int32)
        print("one:", one)

        just_beyond_primes = tf.add(primes, one)
        print("just_beyond_primes:", just_beyond_primes)

        two = tf.constant(2, dtype=tf.int32)
        primes_doubled = primes * two
        print("primes_doubled:", primes_doubled)
```

1.1.3 Exercise #1: Arithmetic over vectors.

Perform vector arithmetic to create a "just_under_primes_squared" vector, where the ith element is equal to the ith element in primes squared, minus 1. For example, the second element would be equal to 3 * 3 - 1 = 8.

Make use of either the tf.multiply or tf.pow ops to square the value of each element in the primes vector.

```
In [0]: # Write your code for Task 1 here.
```

1.1.4 Solution

Click below for a solution.

```
In [0]: # Task: Square each element in the primes vector, then subtract 1.

        def solution(primes):
          primes_squared = tf.multiply(primes, primes)
          neg_one = tf.constant(-1, dtype=tf.int32)
          just_under_primes_squared = tf.add(primes_squared, neg_one)
          return just_under_primes_squared

        def alternative_solution(primes):
          primes_squared = tf.pow(primes, 2)
          one = tf.constant(1, dtype=tf.int32)
          just_under_primes_squared = tf.subtract(primes_squared, one)
          return just_under_primes_squared

        primes = tf.constant([2, 3, 5, 7, 11, 13], dtype=tf.int32)
        just_under_primes_squared = solution(primes)
        print("just_under_primes_squared:", just_under_primes_squared)
```

1.2 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.

```
In [0]: # A 3x4 matrix (2-d tensor).
        x = tf.constant([[5, 2, 4, 3], [5, 1, 6, -2], [-1, 3, -1, -2]],
                        dtype=tf.int32)

        # A 4x2 matrix (2-d tensor).
        y = tf.constant([[2, 2], [3, 5], [4, 5], [1, 6]], dtype=tf.int32)

        # Multiply `x` by `y`; result is 3x2 matrix.
        matrix_multiply_result = tf.matmul(x, y)

        print(matrix_multiply_result)
```

1.3 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:

```
In [0]: # 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)

        reshaped_2x8_matrix = tf.reshape(matrix, [2, 8])
        reshaped_4x4_matrix = tf.reshape(matrix, [4, 4])

        print("Original matrix (8x2):")
        print(matrix.numpy())
        print("Reshaped matrix (2x8):")
        print(reshaped_2x8_matrix.numpy())
        print("Reshaped matrix (4x4):")
        print(reshaped_4x4_matrix.numpy())
```

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.

```
In [0]: # 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)

        reshaped_2x2x4_tensor = tf.reshape(matrix, [2, 2, 4])
        one_dimensional_vector = tf.reshape(matrix, [16])

        print("Original matrix (8x2):")
        print(matrix.numpy())
        print("Reshaped 3-D tensor (2x2x4):")
        print(reshaped_2x2x4_tensor.numpy())
        print("1-D vector:")
        print(one_dimensional_vector.numpy())
```

1.3.1 Exercise #2: 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.

```
In [0]: # Write your code for Task 2 here.
```

1.3.2 Solution

Click below for a solution.

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

One possible solution is to reshape `a` into a 2x3 matrix and reshape `b` into a a 3x1 matrix, resulting in a 2x1 matrix after multiplication:

```
In [0]: # Task: Reshape two tensors in order to multiply them

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

        reshaped_a = tf.reshape(a, [2, 3])
        reshaped_b = tf.reshape(b, [3, 1])
        c = tf.matmul(reshaped_a, reshaped_b)

        print("reshaped_a (2x3):")
        print(reshaped_a.numpy())
        print("reshaped_b (3x1):")
        print(reshaped_b.numpy())
        print("reshaped_a x reshaped_b (2x1):")
        print(c.numpy())
```

An alternative solution would be to reshape `a` into a 6x1 matrix and `b` into a 1x3 matrix, resulting in a 6x3 matrix after multiplication.

1.4 Variables, Initialization and Assignment

So far, all the operations we performed were on static values (`tf.constant`); calling `numpy()` 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):

```
In [0]: # Create a scalar variable with the initial value 3.
        v = tf.contrib.eager.Variable([3])

        # Create a vector variable of shape [1, 4], with random initial values,
        # sampled from a normal distribution with mean 1 and standard deviation 0.35.
        w = tf.contrib.eager.Variable(tf.random_normal([1, 4], mean=1.0, stddev=0.35))

        print("v:", v.numpy())
        print("w:", w.numpy())
```

To change the value of a variable, use the `assign` op:

```
In [0]: v = tf.contrib.eager.Variable([3])
        print(v.numpy())
```

```
tf.assign(v, [7])
print(v.numpy())

v.assign([5])
print(v.numpy())
```

When assigning a new value to a variable, its shape must be equal to its previous shape:

```
In [0]: v = tf.contrib.eager.Variable([[1, 2, 3], [4, 5, 6]])
        print(v.numpy())

        try:
          print("Assigning [7, 8, 9] to v")
          v.assign([7, 8, 9])
        except ValueError as e:
          print("Exception:", e)
```

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.

1.4.1 Exercise #3: 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 API reference to solve this task.

```
In [0]: # Write your code for Task 3 here.
```

1.4.2 Solution

Click below for a solution.

We're going to place dice throws inside two separate 10x1 matrices, `die1` and `die2`. The summation of the dice rolls will be stored in `dice_sum`, then the resulting 10x3 matrix will be created by *concatenating* the three 10x1 matrices together into a single matrix.

Alternatively, we could have placed dice throws inside a single 10x2 matrix, but adding different columns of the same matrix would be more complicated. We also could have placed dice throws inside two 1-D tensors (vectors), but doing so would require transposing the result.

In [0]:
```
# Task: Simulate 10 throws of two dice. Store the results in a 10x3 matrix.

die1 = tf.contrib.eager.Variable(
    tf.random_uniform([10, 1], minval=1, maxval=7, dtype=tf.int32))
die2 = tf.contrib.eager.Variable(
    tf.random_uniform([10, 1], minval=1, maxval=7, dtype=tf.int32))

dice_sum = tf.add(die1, die2)
resulting_matrix = tf.concat(values=[die1, die2, dice_sum], axis=1)

print(resulting_matrix.numpy())
```