vahidk / **EffectiveTensorflow**

TensorFlow tutorials and best practices.

#tensorflow #neural-network #deep-learning #machine-learning #ebook

102 commits | 1 branch | 0 releases | 10 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

vahidk Update readme. Latest commit bbb49b7 on 27 Oct 2017

code	Fix typo.	4 months ago
.gitignore	Update intro.	5 months ago
.gitmodules	Moved framework.	5 months ago
README.md	Update readme.	3 months ago

README.md

Effective TensorFlow

Table of Contents

We aim to gradually expand this series by adding new articles and keep the content up to date with the latest releases of TensorFlow API. If you have suggestions on how to improve this series or find the explanations ambiguous, feel free to create an issue, send patches, or reach out by email.

We encourage you to also check out the accompanied neural network training framework built on top of tf.contrib.learn API. The framework can be downloaded separately:

```
git clone https://github.com/vahidk/TensorflowFramework.git
```

TensorFlow Basics

The most striking difference between TensorFlow and other numerical computation libraries such as NumPy is that operations in TensorFlow are symbolic. This is a powerful concept that allows TensorFlow to do all sort of things (e.g. automatic differentiation) that are not possible with imperative libraries such as NumPy. But it also comes at the cost of making it harder to grasp. Our attempt here is to demystify TensorFlow and provide some guidelines and best practices for more effective use of TensorFlow.

Let's start with a simple example, we want to multiply two random matrices. First we look at an implementation done in NumPy:

```
import numpy as np

x = np.random.normal(size=[10, 10])
y = np.random.normal(size=[10, 10])
z = np.dot(x, y)

print(z)
```

Now we perform the exact same computation this time in TensorFlow:

```
import tensorflow as tf

x = tf.random_normal([10, 10])
y = tf.random_normal([10, 10])
z = tf.matmul(x, y)

sess = tf.Session()
z_val = sess.run(z)

print(z_val)
```

Unlike NumPy that immediately performs the computation and produces the result, tensorflow only gives us a handle (of type Tensor) to a node in the graph that represents the result. If we try printing the value of z directly, we get something like this:

```
Tensor("MatMul:0", shape=(10, 10), dtype=float32)
```

Since both the inputs have a fully defined shape, tensorflow is able to infer the shape of the tensor as well as its type. In order to compute the value of the tensor we need to create a session and evaluate it using Session.run() method.

Tip: When using Jupyter notebook make sure to call tf.reset_default_graph() at the beginning to clear the symbolic graph before defining new nodes.

To understand how powerful symbolic computation can be let's have a look at another example. Assume that we have samples from a curve (say f(x) = 5x^2 + 3) and we want to estimate f(x) based on these samples. We define a parametric function g(x, w) = w0 x^2 + w1 x + w2, which is a function of the input x and latent parameters w, our goal is then to find the latent parameters such that g(x, w) ≈ f(x). This can be done by minimizing the following loss function: L(w) = ∑ (f(x) - g(x, w))^2. Although there's a closed form solution for this simple problem, we opt to use a more general approach that can be applied to any arbitrary differentiable function, and that is using stochastic gradient descent. We simply compute the average gradient of L(w) with respect to w over a set of sample points and move in the opposite direction.

Here's how it can be done in TensorFlow:

```
import numpy as np
import tensorflow as tf
```

```
# Placeholders are used to feed values from python to TensorFlow ops. We define
# two placeholders, one for input feature x, and one for output y.
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

# Assuming we know that the desired function is a polynomial of 2nd degree, we
# allocate a vector of size 3 to hold the coefficients. The variable will be
# automatically initialized with random noise.
w = tf.get_variable("w", shape=[3, 1])

# We define yhat to be our estimate of y.
f = tf.stack([tf.square(x), x, tf.ones_like(x)], 1)
yhat = tf.squeeze(tf.matmul(f, w), 1)

# The loss is defined to be the l2 distance between our estimate of y and its
# true value. We also added a shrinkage term, to ensure the resulting weights
# would be small.
loss = tf.nn.l2_loss(yhat - y) + 0.1 * tf.nn.l2_loss(w)

# We use the Adam optimizer with learning rate set to 0.1 to minimize the loss.
train_op = tf.train.AdamOptimizer(0.1).minimize(loss)

def generate_data():
    x_val = np.random.uniform(-10.0, 10.0, size=100)
    y_val = 5 * np.square(x_val) + 3
    return x_val, y_val

sess = tf.Session()
# Since we are using variables we first need to initialize them.
sess.run(tf.global_variables_initializer())
for _ in range(1000):
    x_val, y_val = generate_data()
    _, loss_val = sess.run([train_op, loss], {x: x_val, y: y_val})
    print(loss_val)
print(sess.run([w]))
```

By running this piece of code you should see a result close to this:

```
[4.9924135, 0.00040895029, 3.4504161]
```

Which is a relatively close approximation to our parameters.

This is just tip of the iceberg for what TensorFlow can do. Many problems such as optimizing large neural networks with millions of parameters can be implemented efficiently in TensorFlow in just a few lines of code. TensorFlow takes care of scaling across multiple devices, and threads, and supports a variety of platforms.

Understanding static and dynamic shapes

Tensors in TensorFlow have a static shape attribute which is determined during graph construction. The static shape may be underspecified. For example we might define a tensor of shape [None, 128]:

```
import tensorflow as tf

a = tf.placeholder(tf.float32, [None, 128])
```

This means that the first dimension can be of any size and will be determined dynamically during Session.run(). You can query the static shape of a Tensor as follows:

```
static_shape = a.shape.as_list()  # returns [None, 128]
```

To get the dynamic shape of the tensor you can call tf.shape op, which returns a tensor representing the shape of the given tensor:

```
dynamic_shape = tf.shape(a)
```

The static shape of a tensor can be set with Tensor.set_shape() method:

```
a.set_shape([32, 128])  # static shape of a is [32, 128]
a.set_shape([None, 128])  # first dimension of a is determined dynamically
```

You can reshape a given tensor dynamically using tf.reshape function:

```
a =  tf.reshape(a, [32, 128])
```

It can be convenient to have a function that returns the static shape when available and dynamic shape when it's not. The following utility function does just that:

```
def get_shape(tensor):
  static_shape = tensor.shape.as_list()
  dynamic_shape = tf.unstack(tf.shape(tensor))
  dims = [s[1] if s[0] is None else s[0]
          for s in zip(static_shape, dynamic_shape)]
  return dims
```

Now imagine we want to convert a Tensor of rank 3 to a tensor of rank 2 by collapsing the second and third dimensions into one. We can use our get_shape() function to do that:

```
b = tf.placeholder(tf.float32, [None, 10, 32])
shape = get_shape(b)
b = tf.reshape(b, [shape[0], shape[1] * shape[2]])
```

Note that this works whether the shapes are statically specified or not.

In fact we can write a general purpose reshape function to collapse any list of dimensions:

```
import tensorflow as tf
import numpy as np

def reshape(tensor, dims_list):
  shape = get_shape(tensor)
  dims_prod = []
  for dims in dims_list:
    if isinstance(dims, int):
      dims_prod.append(shape[dims])
    elif all([isinstance(shape[d], int) for d in dims]):
      dims_prod.append(np.prod([shape[d] for d in dims]))
    else:
      dims_prod.append(tf.prod([shape[d] for d in dims]))
  tensor = tf.reshape(tensor, dims_prod)
  return tensor
```

Then collapsing the second dimension becomes very easy:

```
b = tf.placeholder(tf.float32, [None, 10, 32])
b = reshape(b, [0, [1, 2]])
```

Scopes and when to use them

Variables and tensors in TensorFlow have a name attribute that is used to identify them in the symbolic graph. If you don't specify a name when creating a variable or a tensor, TensorFlow automatically assigns a name for you:

```
a = tf.constant(1)
print(a.name)  # prints "Const:0"

b = tf.Variable(1)
print(b.name)  # prints "Variable:0"
```

You can overwrite the default name by explicitly specifying it:

```
a = tf.constant(1, name="a")
print(a.name)  # prints "a:0"

b = tf.Variable(1, name="b")
print(b.name)  # prints "b:0"
```

TensorFlow introduces two different context managers to alter the name of tensors and variables. The first is tf.name_scope:

```
with tf.name_scope("scope"):
  a = tf.constant(1, name="a")
  print(a.name)  # prints "scope/a:0"

  b = tf.Variable(1, name="b")
  print(b.name)  # prints "scope/b:0"

  c = tf.get_variable(name="c", shape=[])
  print(c.name)  # prints "c:0"
```

Note that there are two ways to define new variables in TensorFlow, by creating a tf.Variable object or by calling tf.get_variable. Calling tf.get_variable with a new name results in creating a new variable, but if a variable with the same name exists it will raise a ValueError exception, telling us that re-declaring a variable is not allowed.

tf.name_scope affects the name of tensors and variables created with tf.Variable, but doesn't impact the variables created with tf.get_variable.

Unlike tf.name_scope, tf.variable_scope modifies the name of variables created with tf.get_variable as well:

```
with tf.variable_scope("scope"):
  a = tf.constant(1, name="a")
  print(a.name)  # prints "scope/a:0"

  b = tf.Variable(1, name="b")
  print(b.name)  # prints "scope/b:0"

  c = tf.get_variable(name="c", shape=[])
  print(c.name)  # prints "scope/c:0"


with tf.variable_scope("scope"):
  a1 = tf.get_variable(name="a", shape=[])
  a2 = tf.get_variable(name="a", shape=[])  # Disallowed
```

But what if we actually want to reuse a previously declared variable? Variable scopes also provide the functionality to do that:

```
with tf.variable_scope("scope"):
  a1 = tf.get_variable(name="a", shape=[])
with tf.variable_scope("scope", reuse=True):
  a2 = tf.get_variable(name="a", shape=[])  # OK
```

This becomes handy for example when using built-in neural network layers:

```
features1 = tf.layers.conv2d(image1, filters=32, kernel_size=3)
# Use the same convolution weights to process the second image:
with tf.variable_scope(tf.get_variable_scope(), reuse=True):
  features2 = tf.layers.conv2d(image2, filters=32, kernel_size=3)
```

This syntax may not look very clean to some. Especially if you want to do lots of variable sharing keeping track of when to define new variables and when to reuse them can be cumbersome and error prone. TensorFlow templates are designed to handle this automatically:

```
conv3x32 = tf.make_template("conv3x32", lambda x: tf.layers.conv2d(x, 32, 3))
features1 = conv3x32(image1)
features2 = conv3x32(image2)  # Will reuse the convolution weights.
```

You can turn any function to a TensorFlow template. Upon the first call to a template, the variables defined inside the function would be declared and in the consecutive invocations they would automatically get reused.

Broadcasting the good and the ugly

TensorFlow supports broadcasting elementwise operations. Normally when you want to perform operations like addition and multiplication, you need to make sure that shapes of the operands match, e.g. you can't add a tensor of shape [3, 2] to a tensor of shape [3, 4]. But there's a special case and that's when you have a singular dimension. TensorFlow implicitly tiles the tensor across its singular dimensions to match the shape of the other operand. So it's valid to add a tensor of shape [3, 2] to a tensor of shape [3, 1]

```
import tensorflow as tf

a = tf.constant([[1., 2.], [3., 4.]])
b = tf.constant([[1.], [2.]])
# c = a + tf.tile(b, [1, 2])
c = a + b
```

Broadcasting allows us to perform implicit tiling which makes the code shorter, and more memory efficient, since we don't need to store the result of the tiling operation. One neat place that this can be used is when combining features of varying length. In order to concatenate features of varying length we commonly tile the input tensors, concatenate the result and apply some nonlinearity. This is a common pattern across a variety of neural network architectures:

```
a = tf.random_uniform([5, 3, 5])
b = tf.random_uniform([5, 1, 6])

# concat a and b and apply nonlinearity
tiled_b = tf.tile(b, [1, 3, 1])
c = tf.concat([a, tiled_b], 2)
d = tf.layers.dense(c, 10, activation=tf.nn.relu)
```

But this can be done more efficiently with broadcasting. We use the fact that f(m(x + y)) is equal to f(mx + my). So we can do the linear operations separately and use broadcasting to do implicit concatenation:

```
pa = tf.layers.dense(a, 10, activation=None)
pb = tf.layers.dense(b, 10, activation=None)
d = tf.nn.relu(pa + pb)
```

In fact this piece of code is pretty general and can be applied to tensors of arbitrary shape as long as broadcasting between tensors is possible:

```
def merge(a, b, units, activation=tf.nn.relu):
    pa = tf.layers.dense(a, units, activation=None)
    pb = tf.layers.dense(b, units, activation=None)
    c = pa + pb
    if activation is not None:
        c = activation(c)
    return c
```

A slightly more general form of this function is included in the cookbook.

So far we discussed the good part of broadcasting. But what's the ugly part you may ask? Implicit assumptions almost always make debugging harder to do. Consider the following example:

```
a = tf.constant([[1.], [2.]])
b = tf.constant([1., 2.])
c = tf.reduce_sum(a + b)
```

What do you think the value of c would be after evaluation? If you guessed 6, that's wrong. It's going to be 12. This is because when rank of two tensors don't match, TensorFlow automatically expands the first dimension of the tensor with lower rank before the elementwise operation, so the result of addition would be [[2, 3], [3, 4]], and the reducing over all parameters would give us 12.

The way to avoid this problem is to be as explicit as possible. Had we specified which dimension we would want to reduce across, catching this bug would have been much easier:

```
a = tf.constant([[1.], [2.]])
b = tf.constant([1., 2.])
c = tf.reduce_sum(a + b, 0)
```

Here the value of c would be [5, 7], and we immediately would guess based on the shape of the result that there's something wrong. A general rule of thumb is to always specify the dimensions in reduction operations and when using tf.squeeze.

Feeding data to TensorFlow

TensorFlow is designed to work efficiently with large amount of data. So it's important not to starve your TensorFlow model in order to maximize its performance. There are various ways that you can feed your data to TensorFlow.

Constants

The simplest approach is to embed the data in your graph as a constant:

```
import tensorflow as tf
import numpy as np

actual_data = np.random.normal(size=[100])

data = tf.constant(actual_data)
```

This approach can be very efficient, but it's not very flexible. One problem with this approach is that, in order to use your model with another dataset you have to rewrite the graph. Also, you have to load all of your data at once and keep it in memory which would only work with small datasets.

Placeholders

Using placeholders solves both of these problems:

```
import tensorflow as tf
import numpy as np

data = tf.placeholder(tf.float32)

prediction = tf.square(data) + 1

actual_data = np.random.normal(size=[100])

tf.Session().run(prediction, feed_dict={data: actual_data})
```

Placeholder operator returns a tensor whose value is fetched through the feed_dict argument in Session.run function. Note that running Session.run without feeding the value of data in this case will result in an error.

Python ops

Another approach to feed the data to TensorFlow is by using Python ops:

```
def py_input_fn():
    actual_data = np.random.normal(size=[100])
    return actual_data

data = tf.py_func(py_input_fn, [], (tf.float32))
```

Python ops allow you to convert a regular Python function to a TensorFlow operation.

Dataset API

The recommended way of reading the data in TensorFlow however is through the dataset API:

```
actual_data = np.random.normal(size=[100])
dataset = tf.contrib.data.Dataset.from_tensor_slices(actual_data)
data = dataset.make_one_shot_iterator().get_next()
```

If you need to read your data from file, it may be more efficient to write it in TFrecord format and use TFRecordDataset to read it:

```
dataset = tf.contrib.data.TFRecordDataset(path_to_data)
```

See the official docs for an example of how to write your dataset in TFrecord format.

Dataset API allows you to make efficient data processing pipelines easily. For example this is how we process our data in the accompanied framework (See trainer.py):

```
dataset = ...
dataset = dataset.cache()
if mode == tf.estimator.ModeKeys.TRAIN:
    dataset = dataset.repeat()
    dataset = dataset.shuffle(batch_size * 5)
dataset = dataset.map(parse, num_threads=8)
dataset = dataset.batch(batch_size)
```

After reading the data, we use Dataset.cache method to cache it into memory for improved efficiency. During the training mode, we repeat the dataset indefinitely. This allows us to process the whole dataset many times. We also shuffle the dataset to get batches with different sample distributions. Next, we use the Dataset.map function to perform preprocessing on raw records and convert the data to a usable format for the model. We then create batches of samples by calling Dataset.batch.

Take advantage of the overloaded operators

Just like NumPy, TensorFlow overloads a number of python operators to make building graphs easier and the code more readable.

The slicing op is one of the overloaded operators that can make indexing tensors very easy:

```
z = x[begin:end]  # z = tf.slice(x, [begin], [end-begin])
```

Be very careful when using this op though. The slicing op is very inefficient and often better avoided, especially when the number of slices is high. To understand how inefficient this op can be let's look at an example. We want to manually perform reduction across the rows of a matrix:

```
import tensorflow as tf
import time

x = tf.random_uniform([500, 10])

z = tf.zeros([10])
for i in range(500):
    z += x[i]

sess = tf.Session()
start = time.time()
sess.run(z)
print("Took %f seconds." % (time.time() - start))
```

On my MacBook Pro, this took 2.67 seconds to run! The reason is that we are calling the slice op 500 times, which is going to be very slow to run. A better choice would have been to use tf.unstack op to slice the matrix into a list of vectors all at once:

```
z = tf.zeros([10])
for x_i in tf.unstack(x):
    z += x_i
```

This took 0.18 seconds. Of course, the right way to do this simple reduction is to use tf.reduce_sum op:

```
z = tf.reduce_sum(x, axis=0)
```

This took 0.008 seconds, which is 300x faster than the original implementation.

TensorFlow also overloads a range of arithmetic and logical operators:

```
z = -x  # z = tf.negative(x)
z = x + y  # z = tf.add(x, y)
z = x - y  # z = tf.subtract(x, y)
z = x * y  # z = tf.mul(x, y)
z = x / y  # z = tf.div(x, y)
z = x // y  # z = tf.floordiv(x, y)
z = x % y  # z = tf.mod(x, y)
z = x ** y  # z = tf.pow(x, y)
z = x @ y  # z = tf.matmul(x, y)
z = x > y  # z = tf.greater(x, y)
z = x >= y  # z = tf.greater_equal(x, y)
z = x < y  # z = tf.less(x, y)
z = x <= y  # z = tf.less_equal(x, y)
z = abs(x)  # z = tf.abs(x)
z = x & y  # z = tf.logical_and(x, y)
z = x | y  # z = tf.logical_or(x, y)
z = x ^ y  # z = tf.logical_xor(x, y)
z = ~x  # z = tf.logical_not(x)
```

You can also use the augmented version of these ops. For example `x += y` and `x **= 2` are also valid.

Note that Python doesn't allow overloading "and", "or", and "not" keywords.

TensorFlow also doesn't allow using tensors as booleans, as it may be error prone:

```
x = tf.constant(1.)
if x:  # This will raise a TypeError error
    ...
```

You can either use tf.cond(x, ...) if you want to check the value of the tensor, or use "if x is None" to check the value of the variable.

Other operators that aren't supported are equal (==) and not equal (!=) operators which are overloaded in NumPy but not in TensorFlow. Use the function versions instead which are `tf.equal` and `tf.not_equal`.

Understanding order of execution and control dependencies

As we discussed in the first item, TensorFlow doesn't immediately run the operations that are defined but rather creates corresponding nodes in a graph that can be evaluated with Session.run() method. This also enables TensorFlow to do optimizations at run time to determine the optimal order of execution and possible trimming of unused nodes. If you only have tf.Tensors in your graph you don't need to worry about dependencies but you most probably have tf.Variables too, and tf.Variables make things much more difficult. My advice to is to only use Variables if Tensors don't do the job. This might not make a lot of sense to you now, so let's start with an example.

```
import tensorflow as tf

a = tf.constant(1)
b = tf.constant(2)
a = a + b

tf.Session().run(a)
```

Evaluating "a" will return the value 3 as expected. Note that here we are creating 3 tensors, two constant tensors and another tensor that stores the result of the addition. Note that you can't overwrite the value of a tensor. If you want to modify it you have to create a new tensor. As we did here.

TIP: If you don't define a new graph, TensorFlow automatically creates a graph for you by default. You can use tf.get_default_graph() to get a handle to the graph. You can then inspect the graph, for example by printing all its tensors:

```
print(tf.contrib.graph_editor.get_tensors(tf.get_default_graph()))
```

Unlike tensors, variables can be updated. So let's see how we may use variables to do the same thing:

```
a = tf.Variable(1)
b = tf.constant(2)
assign = tf.assign(a, a + b)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(assign))
```

Again, we get 3 as expected. Note that tf.assign returns a tensor representing the value of the assignment. So far everything seemed to be fine, but let's look at a slightly more complicated example:

```
a = tf.Variable(1)
b = tf.constant(2)
c = a + b

assign = tf.assign(a, 5)

sess = tf.Session()
for i in range(10):
    sess.run(tf.global_variables_initializer())
    print(sess.run([assign, c]))
```

Note that the tensor c here won't have a deterministic value. This value might be 3 or 7 depending on whether addition or assignment gets executed first.

You should note that the order that you define ops in your code doesn't matter to TensorFlow runtime. The only thing that matters is the control dependencies. Control dependencies for tensors are straightforward. Every time you use a tensor in an operation that op will define an implicit dependency to that tensor. But things get complicated with variables because they can take many values.

When dealing with variables, you may need to explicitly define dependencies using tf.control_dependencies() as follows:

```
a = tf.Variable(1)
b = tf.constant(2)
c = a + b

with tf.control_dependencies([c]):
    assign = tf.assign(a, 5)

sess = tf.Session()
for i in range(10):
    sess.run(tf.global_variables_initializer())
    print(sess.run([assign, c]))
```

This will make sure that the assign op will be called after the addition.

Control flow operations: conditionals and loops

When building complex models such as recurrent neural networks you may need to control the flow of operations through conditionals and loops. In this section we introduce a number of commonly used control flow ops.

Let's assume you want to decide whether to multiply to or add two given tensors based on a predicate. This can be simply implemented with tf.cond which acts as a python "if" function:

```
a = tf.constant(1)
b = tf.constant(2)

p = tf.constant(True)

x = tf.cond(p, lambda: a + b, lambda: a * b)

print(tf.Session().run(x))
```

Since the predicate is True in this case, the output would be the result of the addition, which is 3.

Most of the times when using TensorFlow you are using large tensors and want to perform operations in batch. A related conditional operation is tf.where, which like tf.cond takes a predicate, but selects the output based on the condition in batch.

```
a = tf.constant([1, 1])
b = tf.constant([2, 2])

p = tf.constant([True, False])

x = tf.where(p, a + b, a * b)

print(tf.Session().run(x))
```

This will return [3, 2].

Another widely used control flow operation is tf.while_loop. It allows building dynamic loops in TensorFlow that operate on sequences of variable length. Let's see how we can generate Fibonacci sequence with tf.while_loops:

```
n = tf.constant(5)

def cond(i, a, b):
    return i < n

def body(i, a, b):
    return i + 1, b, a + b

i, a, b = tf.while_loop(cond, body, (2, 1, 1))

print(tf.Session().run(b))
```

This will print 5. tf.while_loops takes a condition function, and a loop body function, in addition to initial values for loop variables. These loop variables are then updated by multiple calls to the body function until the condition returns false.

Now imagine we want to keep the whole series of Fibonacci sequence. We may update our body to keep a record of the history of current values:

```
n = tf.constant(5)

def cond(i, a, b, c):
    return i < n

def body(i, a, b, c):
    return i + 1, b, a + b, tf.concat([c, [a + b]], 0)

i, a, b, c = tf.while_loop(cond, body, (2, 1, 1, tf.constant([1, 1])))

print(tf.Session().run(c))
```

Now if you try running this, TensorFlow will complain that the shape of the the fourth loop variable is changing. So you must make that explicit that it's intentional:

```
i, a, b, c = tf.while_loop(
    cond, body, (2, 1, 1, tf.constant([1, 1])),
    shape_invariants=(tf.TensorShape([]),
                      tf.TensorShape([]),
                      tf.TensorShape([]),
                      tf.TensorShape([None])))
```

This is not only getting ugly, but is also somewhat inefficient. Note that we are building a lot of intermediary tensors that we don't use. TensorFlow has a better solution for this kind of growing arrays. Meet tf.TensorArray. Let's do the same thing this time with tensor arrays:

```
n = tf.constant(5)

c = tf.TensorArray(tf.int32, n)
c = c.write(0, 1)
c = c.write(1, 1)

def cond(i, a, b, c):
    return i < n

def body(i, a, b, c):
    c = c.write(i, a + b)
```

```
    return i + 1, b, a + b, c

i, a, b, c = tf.while_loop(cond, body, (2, 1, 1, c))

c = c.stack()

print(tf.Session().run(c))
```

TensorFlow while loops and tensor arrays are essential tools for building complex recurrent neural networks. As an exercise try implementing beam search using tf.while_loops. Can you make it more efficient with tensor arrays?

Prototyping kernels and advanced visualization with Python ops

Operation kernels in TensorFlow are entirely written in C++ for efficiency. But writing a TensorFlow kernel in C++ can be quite a pain. So, before spending hours implementing your kernel you may want to prototype something quickly, however inefficient. With tf.py_func() you can turn any piece of python code to a TensorFlow operation.

For example this is how you can implement a simple ReLU nonlinearity kernel in TensorFlow as a python op:

```
import numpy as np
import tensorflow as tf
import uuid

def relu(inputs):
    # Define the op in python
    def _relu(x):
        return np.maximum(x, 0.)

    # Define the op's gradient in python
    def _relu_grad(x):
        return np.float32(x > 0)

    # An adapter that defines a gradient op compatible with TensorFlow
    def _relu_grad_op(op, grad):
        x = op.inputs[0]
        x_grad = grad * tf.py_func(_relu_grad, [x], tf.float32)
        return x_grad

    # Register the gradient with a unique id
    grad_name = "MyReluGrad_" + str(uuid.uuid4())
    tf.RegisterGradient(grad_name)(_relu_grad_op)

    # Override the gradient of the custom op
    g = tf.get_default_graph()
    with g.gradient_override_map({"PyFunc": grad_name}):
        output = tf.py_func(_relu, [inputs], tf.float32)
    return output
```

To verify that the gradients are correct you can use TensorFlow's gradient checker:

```
x = tf.random_normal([10])
y = relu(x * x)

with tf.Session():
    diff = tf.test.compute_gradient_error(x, [10], y, [10])
    print(diff)
```

compute_gradient_error() computes the gradient numerically and returns the difference with the provided gradient. What we want is a very low difference.

Note that this implementation is pretty inefficient, and is only useful for prototyping, since the python code is not parallelizable and won't run on GPU. Once you verified your idea, you definitely would want to write it as a C++ kernel.

In practice we commonly use python ops to do visualization on Tensorboard. Consider the case that you are building an image classification model and want to visualize your model predictions during training. TensorFlow allows visualizing images with tf.summary.image() function:

```
image = tf.placeholder(tf.float32)
tf.summary.image("image", image)
```

But this only visualizes the input image. In order to visualize the predictions you have to find a way to add annotations to the image which may be almost impossible with existing ops. An easier way to do this is to do the drawing in python, and wrap it in a python op:

```
import io
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf

def visualize_labeled_images(images, labels, max_outputs=3, name="image"):
    def _visualize_image(image, label):
        # Do the actual drawing in python
        fig = plt.figure(figsize=(3, 3), dpi=80)
        ax = fig.add_subplot(111)
        ax.imshow(image[::-1,...])
        ax.text(0, 0, str(label),
          horizontalalignment="left",
          verticalalignment="top")
        fig.canvas.draw()

        # Write the plot as a memory file.
        buf = io.BytesIO()
        data = fig.savefig(buf, format="png")
        buf.seek(0)

        # Read the image and convert to numpy array
        img = PIL.Image.open(buf)
        return np.array(img.getdata()).reshape(img.size[0], img.size[1], -1)

    def _visualize_images(images, labels):
        # Only display the given number of examples in the batch
        outputs = []
        for i in range(max_outputs):
            output = _visualize_image(images[i], labels[i])
            outputs.append(output)
        return np.array(outputs, dtype=np.uint8)

    # Run the python op.
    figs = tf.py_func(_visualize_images, [images, labels], tf.uint8)
    return tf.summary.image(name, figs)
```

Note that since summaries are usually only evaluated once in a while (not per step), this implementation may be used in practice without worrying about efficiency.

Multi-GPU processing with data parallelism

If you write your software in a language like C++ for a single cpu core, making it run on multiple GPUs in parallel would require rewriting the software from scratch. But this is not the case with TensorFlow. Because of its symbolic nature, tensorflow can hide all that complexity, making it effortless to scale your program across many CPUs and GPUs.

Let's start with the simple example of adding two vectors on CPU:

```
import tensorflow as tf

with tf.device(tf.DeviceSpec(device_type="CPU", device_index=0)):
   a = tf.random_uniform([1000, 100])
   b = tf.random_uniform([1000, 100])
   c = a + b

tf.Session().run(c)
```

The same thing can as simply be done on GPU:

```
with tf.device(tf.DeviceSpec(device_type="GPU", device_index=0)):
    a = tf.random_uniform([1000, 100])
    b = tf.random_uniform([1000, 100])
    c = a + b
```

But what if we have two GPUs and want to utilize both? To do that, we can split the data and use a separate GPU for processing each half:

```
split_a = tf.split(a, 2)
split_b = tf.split(b, 2)

split_c = []
for i in range(2):
    with tf.device(tf.DeviceSpec(device_type="GPU", device_index=i)):
        split_c.append(split_a[i] + split_b[i])

c = tf.concat(split_c, axis=0)
```

Let's rewrite this in a more general form so that we can replace addition with any other set of operations:

```
def make_parallel(fn, num_gpus, **kwargs):
    in_splits = {}
    for k, v in kwargs.items():
        in_splits[k] = tf.split(v, num_gpus)

    out_split = []
    for i in range(num_gpus):
        with tf.device(tf.DeviceSpec(device_type="GPU", device_index=i)):
            with tf.variable_scope(tf.get_variable_scope(), reuse=i > 0):
                out_split.append(fn(**{k : v[i] for k, v in in_splits.items()}))

    return tf.concat(out_split, axis=0)


def model(a, b):
    return a + b

c = make_parallel(model, 2, a=a, b=b)
```

You can replace the model with any function that takes a set of tensors as input and returns a tensor as result with the condition that both the input and output are in batch. Note that we also added a variable scope and set the reuse to true. This makes sure that we use the same variables for processing both splits. This is something that will become handy in our next example.

Let's look at a slightly more practical example. We want to train a neural network on multiple GPUs. During training we not only need to compute the forward pass but also need to compute the backward pass (the gradients). But how can we parallelize the gradient computation? This turns out to be pretty easy.

Recall from the first item that we wanted to fit a second degree polynomial to a set of samples. We reorganized the code a bit to have the bulk of the operations in the model function:

```
import numpy as np
import tensorflow as tf

def model(x, y):
    w = tf.get_variable("w", shape=[3, 1])

    f = tf.stack([tf.square(x), x, tf.ones_like(x)], 1)
    yhat = tf.squeeze(tf.matmul(f, w), 1)

    loss = tf.square(yhat - y)
    return loss

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

loss = model(x, y)

train_op = tf.train.AdamOptimizer(0.1).minimize(
    tf.reduce_mean(loss))

def generate_data():
    x_val = np.random.uniform(-10.0, 10.0, size=100)
    y_val = 5 * np.square(x_val) + 3
    return x_val, y_val

sess = tf.Session()
```

```
sess.run(tf.global_variables_initializer())
for _ in range(1000):
    x_val, y_val = generate_data()
    _, loss_val = sess.run([train_op, loss], {x: x_val, y: y_val})

_, loss_val = sess.run([train_op, loss], {x: x_val, y: y_val})
print(sess.run(tf.contrib.framework.get_variables_by_name("w")))
```

Now let's use make_parallel that we just wrote to parallelize this. We only need to change two lines of code from the above code:

```
loss = make_parallel(model, 2, x=x, y=y)

train_op = tf.train.AdamOptimizer(0.1).minimize(
    tf.reduce_mean(loss),
    colocate_gradients_with_ops=True)
```

The only thing that we need to change to parallelize backpropagation of gradients is to set the colocate_gradients_with_ops flag to true. This ensures that gradient ops run on the same device as the original op.

Debugging TensorFlow models

Symbolic nature of TensorFlow makes it relatively more difficult to debug TensorFlow code compared to regular python code. Here we introduce a number of tools included with TensorFlow that make debugging much easier.

Probably the most common error one can make when using TensorFlow is passing Tensors of wrong shape to ops. Many TensorFlow ops can operate on tensors of different ranks and shapes. This can be convenient when using the API, but may lead to extra headache when things go wrong.

For example, consider the tf.matmul op, it can multiply two matrices:

```
a = tf.random_uniform([2, 3])
b = tf.random_uniform([3, 4])
c = tf.matmul(a, b)  # c is a tensor of shape [2, 4]
```

But the same function also does batch matrix multiplication:

```
a = tf.random_uniform([10, 2, 3])
b = tf.random_uniform([10, 3, 4])
tf.matmul(a, b)  # c is a tensor of shape [10, 2, 4]
```

Another example that we talked about before in the broadcasting section is add operation which supports broadcasting:

```
a = tf.constant([[1.], [2.]])
b = tf.constant([1., 2.])
c = a + b  # c is a tensor of shape [2, 2]
```

Validating your tensors with tf.assert* ops

One way to reduce the chance of unwanted behavior is to explicitly verify the rank or shape of intermediate tensors with tf.assert* ops.

```
a = tf.constant([[1.], [2.]])
b = tf.constant([1., 2.])
check_a = tf.assert_rank(a, 1)  # This will raise an InvalidArgumentError exception
check_b = tf.assert_rank(b, 1)
with tf.control_dependencies([check_a, check_b]):
    c = a + b  # c is a tensor of shape [2, 2]
```

Remember that assertion nodes like other operations are part of the graph and if not evaluated would get pruned during Session.run(). So make sure to create explicit dependencies to assertion ops, to force TensorFlow to execute them.

You can also use assertions to validate the value of tensors at runtime:

```
check_pos = tf.assert_positive(a)
```

See the official docs for a full list of assertion ops.

Logging tensor values with tf.Print

Another useful built-in function for debugging is tf.Print which logs the given tensors to the standard error:

```
input_copy = tf.Print(input, tensors_to_print_list)
```

Note that tf.Print returns a copy of its first argument as output. One way to force tf.Print to run is to pass its output to another op that gets executed. For example if we want to print the value of tensors a and b before adding them we could do something like this:

```
a = ...
b = ...
a = tf.Print(a, [a, b])
c = a + b
```

Alternatively we could manually define a control dependency.

Check your gradients with tf.compute_gradient_error

Not all the operations in TensorFlow come with gradients, and it's easy to unintentionally build graphs for which TensorFlow can not compute the gradients.

Let's look at an example:

```
import tensorflow as tf

def non_differentiable_softmax_entropy(logits):
    probs = tf.nn.softmax(logits)
    return tf.nn.softmax_cross_entropy_with_logits(labels=probs, logits=logits)

w = tf.get_variable("w", shape=[5])
y = -non_differentiable_softmax_entropy(w)

opt = tf.train.AdamOptimizer()
train_op = opt.minimize(y)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
for i in range(10000):
    sess.run(train_op)

print(sess.run(tf.nn.softmax(w)))
```

We are using tf.nn.softmax_cross_entropy_with_logits to define entropy over a categorical distribution. We then use Adam optimizer to find the weights with maximum entropy. If you have passed a course on information theory, you would know that uniform distribution contains maximum entropy. So you would expect for the result to be [0.2, 0.2, 0.2, 0.2, 0.2]. But if you run this you may get unexpected results like this:

```
[ 0.34081486  0.24287023  0.23465775  0.08935683  0.09230034]
```

It turns out tf.nn.softmax_cross_entropy_with_logits has undefined gradients with respect to labels! But how may we spot this if we didn't know?

Fortunately for us TensorFlow comes with a numerical differentiator that can be used to find symbolic gradient errors. Let's see how we can use it:

```
with tf.Session():
    diff = tf.test.compute_gradient_error(w, [5], y, [])
    print(diff)
```

If you run this, you would see that the difference between the numerical and symbolic gradients are pretty high (0.06 - 0.1 in my tries).

Now let's fix our function with a differentiable version of the entropy and check again:

```
import tensorflow as tf
import numpy as np

def softmax_entropy(logits, dim=-1):
    plogp = tf.nn.softmax(logits, dim) * tf.nn.log_softmax(logits, dim)
    return -tf.reduce_sum(plogp, dim)

w = tf.get_variable("w", shape=[5])
y = -softmax_entropy(w)

print(w.get_shape())
print(y.get_shape())

with tf.Session() as sess:
    diff = tf.test.compute_gradient_error(w, [5], y, [])
    print(diff)
```

The difference should be ~0.0001 which looks much better.

Now if you run the optimizer again with the correct version you can see the final weights would be:

```
[ 0.2  0.2  0.2  0.2  0.2]
```

which are exactly what we wanted.

TensorFlow summaries, and tfdbg (TensorFlow Debugger) are other tools that can be used for debugging. Please refer to the official docs to learn more.

Numerical stability in TensorFlow

When using any numerical computation library such as NumPy or TensorFlow, it's important to note that writing mathematically correct code doesn't necessarily lead to correct results. You also need to make sure that the computations are stable.

Let's start with a simple example. From primary school we know that x * y / y is equal to x for any non zero value of x. But let's see if that's always true in practice:

```
import numpy as np

x = np.float32(1)

y = np.float32(1e-50)  # y would be stored as zero
z = x * y / y

print(z)  # prints nan
```

The reason for the incorrect result is that y is simply too small for float32 type. A similar problem occurs when y is too large:

```
y = np.float32(1e39)  # y would be stored as inf
z = x * y / y

print(z)  # prints 0
```

The smallest positive value that float32 type can represent is 1.4013e-45 and anything below that would be stored as zero. Also, any number beyond 3.40282e+38, would be stored as inf.

```
print(np.nextafter(np.float32(0), np.float32(1)))  # prints 1.4013e-45
print(np.finfo(np.float32).max)  # print 3.40282e+38
```

To make sure that your computations are stable, you want to avoid values with small or very large absolute value. This may sound very obvious, but these kind of problems can become extremely hard to debug especially when doing gradient descent in TensorFlow. This is because you not only need to make sure that all the values in the forward pass are within the valid range of your data types, but also you need to make sure of the same for the backward pass (during gradient computation).

Let's look at a real example. We want to compute the softmax over a vector of logits. A naive implementation would look something like this:

```
import tensorflow as tf

def unstable_softmax(logits):
    exp = tf.exp(logits)
    return exp / tf.reduce_sum(exp)

tf.Session().run(unstable_softmax([1000., 0.]))  # prints [ nan, 0.]
```

Note that computing the exponential of logits for relatively small numbers results to gigantic results that are out of float32 range. The largest valid logit for our naive softmax implementation is ln(3.40282e+38) = 88.7, anything beyond that leads to a nan outcome.

But how can we make this more stable? The solution is rather simple. It's easy to see that exp(x - c) / ∑ exp(x - c) = exp(x) / ∑ exp(x). Therefore we can subtract any constant from the logits and the result would remain the same. We choose this constant to be the maximum of logits. This way the domain of the exponential function would be limited to [-inf, 0], and consequently its range would be [0.0, 1.0] which is desirable:

```
import tensorflow as tf

def softmax(logits):
    exp = tf.exp(logits - tf.reduce_max(logits))
    return exp / tf.reduce_sum(exp)

tf.Session().run(softmax([1000., 0.]))  # prints [ 1., 0.]
```

Let's look at a more complicated case. Consider we have a classification problem. We use the softmax function to produce probabilities from our logits. We then define our loss function to be the cross entropy between our predictions and the labels. Recall that cross entropy for a categorical distribution can be simply defined as xe(p, q) = -∑ p_i log(q_i). So a naive implementation of the cross entropy would look like this:

```
def unstable_softmax_cross_entropy(labels, logits):
    logits = tf.log(softmax(logits))
    return -tf.reduce_sum(labels * logits)

labels = tf.constant([0.5, 0.5])
logits = tf.constant([1000., 0.])

xe = unstable_softmax_cross_entropy(labels, logits)

print(tf.Session().run(xe))  # prints inf
```

Note that in this implementation as the softmax output approaches zero, the log's output approaches infinity which causes instability in our computation. We can rewrite this by expanding the softmax and doing some simplifications:

```
def softmax_cross_entropy(labels, logits):
    scaled_logits = logits - tf.reduce_max(logits)
    normalized_logits = scaled_logits - tf.reduce_logsumexp(scaled_logits)
    return -tf.reduce_sum(labels * normalized_logits)

labels = tf.constant([0.5, 0.5])
logits = tf.constant([1000., 0.])

xe = softmax_cross_entropy(labels, logits)

print(tf.Session().run(xe))  # prints 500.0
```

We can also verify that the gradients are also computed correctly:

```
g = tf.gradients(xe, logits)
print(tf.Session().run(g))  # prints [0.5, -0.5]
```

which is correct.

Let me remind again that extra care must be taken when doing gradient descent to make sure that the range of your functions as well as the gradients for each layer are within a valid range. Exponential and logarithmic functions when used naively are especially problematic because they can map small numbers to enormous ones and the other way around.

Building a neural network training framework with learn API

For simplicity, in most of the examples here we manually create sessions and we don't care about saving and loading checkpoints but this is not how we usually do things in practice. You most probably want to use the learn API to take care of session management and logging. We provide a simple but practical framework for training neural networks using TensorFlow. In this item we explain how this framework works.

When experimenting with neural network models you usually have a training/test split. You want to train your model on the training set, and once in a while evaluate it on test set and compute some metrics. You also need to store the model parameters as a checkpoint, and ideally you want to be able to stop and resume training. TensorFlow's learn API is designed to make this job easier, letting us focus on developing the actual model.

The most basic way of using tf.learn API is to use tf.Estimator object directly. You need to define a model function that defines a loss function, a train op, one or a set of predictions, and optinoally a set of metric ops for evaluation:

```
import tensorflow as tf

def model_fn(features, labels, mode, params):
    predictions = ...
    loss = ...
    train_op = ...
    metric_ops = ...
    return tf.estimator.EstimatorSpec(
        mode=mode,
        predictions=predictions,
        loss=loss,
        train_op=train_op,
        eval_metric_ops=metric_ops)

params = ...
run_config = tf.contrib.learn.RunConfig(model_dir=FLAGS.output_dir)
estimator = tf.estimator.Estimator(
    model_fn=model_fn, config=run_config, params=params)
```

To train the model you would then simply call Estimator.train() function while providing an input function to read the data:

```
def input_fn():
    features = ...
    labels = ...
    return features, labels

estimator.train(input_fn=input_fn, max_steps=...)
```

and to evaluate the model, simply call Estimator.evaluate():

```
estimator.evaluate(input_fn=input_fn)
```

Estimator object might be good enough for simple cases, but TensorFlow provides a higher level object called Experiment which provides some additional useful functionality. Creating an experiment object is very easy:

```
experiment = tf.contrib.learn.Experiment(
    estimator=estimator,
    train_input_fn=train_input_fn,
    eval_input_fn=eval_input_fn)
```

Now we can call train_and_evaluate function to compute the metrics while training:

```
experiment.train_and_evaluate()
```

An even higher level way of running experiments is by using learn_runner.run() function. Here's how our main function looks like in the provided framework:

```
import tensorflow as tf

tf.flags.DEFINE_string("output_dir", "", "Optional output dir.")
tf.flags.DEFINE_string("schedule", "train_and_evaluate", "Schedule.")
tf.flags.DEFINE_string("hparams", "", "Hyper parameters.")

FLAGS = tf.flags.FLAGS

def experiment_fn(run_config, hparams):
  estimator = tf.estimator.Estimator(
    model_fn=make_model_fn(),
    config=run_config,
    params=hparams)
  return tf.contrib.learn.Experiment(
    estimator=estimator,
    train_input_fn=make_input_fn(tf.estimator.ModeKeys.TRAIN, hparams),
    eval_input_fn=make_input_fn(tf.estimator.ModeKeys.EVAL, hparams))

def main(unused_argv):
  run_config = tf.contrib.learn.RunConfig(model_dir=FLAGS.output_dir)
  hparams = tf.contrib.training.HParams()
  hparams.parse(FLAGS.hparams)

  estimator = tf.contrib.learn.learn_runner.run(
    experiment_fn=experiment_fn,
    run_config=run_config,
    schedule=FLAGS.schedule,
    hparams=hparams)

if __name__ == "__main__":
  tf.app.run()
```

The schedule flag decides which member function of the Experiment object gets called. So, if you for example set schedule to "train_and_evaluate", experiment.train_and_evaluate() would be called.

The input function returns two tensors (or dictionaries of tensors) providing the features and labels to be passed to the model:

```
def input_fn():
    features = ...
    labels = ...
    return features, labels
```

See mnist.py for an example of how to read your data with the dataset API. To learn about various ways of reading your data in TensorFlow refer to this item.

The framework also comes with a simple convolutional network classifier in alexnet.py that includes an example model.

And that's it! This is all you need to get started with TensorFlow learn API. I recommend to have a look at the framework source code and see the official python API to learn more about the learn API.

TensorFlow Cookbook

This section includes implementation of a set of common operations in TensorFlow.

Get shape

```
def get_shape(tensor):
  """Returns static shape if available and dynamic shape otherwise."""
  static_shape = tensor.shape.as_list()
  dynamic_shape = tf.unstack(tf.shape(tensor))
  dims = [s[1] if s[0] is None else s[0]
```

```
        for s in zip(static_shape, dynamic_shape)]
  return dims
```

Batch Gather

```
def batch_gather(tensor, indices):
  """Gather in batch from a tensor of arbitrary size.

  In pseudocode this module will produce the following:
  output[i] = tf.gather(tensor[i], indices[i])

  Args:
    tensor: Tensor of arbitrary size.
    indices: Vector of indices.
  Returns:
    output: A tensor of gathered values.
  """
  shape = get_shape(tensor)
  flat_first = tf.reshape(tensor, [shape[0] * shape[1]] + shape[2:])
  indices = tf.convert_to_tensor(indices)
  offset_shape = [shape[0]] + [1] * (indices.shape.ndims - 1)
  offset = tf.reshape(tf.range(shape[0]) * shape[1], offset_shape)
  output = tf.gather(flat_first, indices + offset)
  return output
```

Beam Search

```
import tensorflow as tf

def rnn_beam_search(update_fn, initial_state, sequence_length, beam_width,
                    begin_token_id, end_token_id, name="rnn"):
  """Beam-search decoder for recurrent models.

  Args:
    update_fn: Function to compute the next state and logits given the current
               state and ids.
    initial_state: Recurrent model states.
    sequence_length: Length of the generated sequence.
    beam_width: Beam width.
    begin_token_id: Begin token id.
    end_token_id: End token id.
    name: Scope of the variables.
  Returns:
    ids: Output indices.
    logprobs: Output log probabilities probabilities.
  """
  batch_size = initial_state.shape.as_list()[0]

  state = tf.tile(tf.expand_dims(initial_state, axis=1), [1, beam_width, 1])

  sel_sum_logprobs = tf.log([[1.] + [0.] * (beam_width - 1)])

  ids = tf.tile([[begin_token_id]], [batch_size, beam_width])
  sel_ids = tf.zeros([batch_size, beam_width, 0], dtype=ids.dtype)

  mask = tf.ones([batch_size, beam_width], dtype=tf.float32)

  for i in range(sequence_length):
    with tf.variable_scope(name, reuse=True if i > 0 else None):

      state, logits = update_fn(state, ids)
      logits = tf.nn.log_softmax(logits)

      sum_logprobs = (
          tf.expand_dims(sel_sum_logprobs, axis=2) +
          (logits * tf.expand_dims(mask, axis=2)))

      num_classes = logits.shape.as_list()[-1]

      sel_sum_logprobs, indices = tf.nn.top_k(
          tf.reshape(sum_logprobs, [batch_size, num_classes * beam_width]),
          k=beam_width)

      ids = indices % num_classes
```

```
    beam_ids = indices // num_classes

    state = batch_gather(state, beam_ids)

    sel_ids = tf.concat([batch_gather(sel_ids, beam_ids),
                         tf.expand_dims(ids, axis=2)], axis=2)

    mask = (batch_gather(mask, beam_ids) *
            tf.to_float(tf.not_equal(ids, end_token_id)))

  return sel_ids, sel_sum_logprobs
```

Merge

```
import tensorflow as tf

def merge(tensors, units, activation=tf.nn.relu, name=None, **kwargs):
  """Merge features with broadcasting support.

  This operation concatenates multiple features of varying length and applies
  non-linear transformation to the outcome.

  Example:
    a = tf.zeros([m, 1, d1])
    b = tf.zeros([1, n, d2])
    c = merge([a, b], d3)  # shape of c would be [m, n, d3].

  Args:
    tensors: A list of tensor with the same rank.
    units: Number of units in the projection function.
  """
  with tf.variable_scope(name, default_name="merge"):
    # Apply linear projection to input tensors.
    projs = []
    for i, tensor in enumerate(tensors):
      proj = tf.layers.dense(
          tensor, units, activation=None,
          name="proj_%d" % i,
          **kwargs)
      projs.append(proj)

    # Compute sum of tensors.
    result = projs.pop()
    for proj in projs:
      result = result + proj

    # Apply nonlinearity.
    if activation:
      result = activation(result)
  return result
```

Entropy

```
import tensorflow as tf

def softmax_entropy(logits, dim=-1):
  """Compute entropy over specified dimensions."""
  plogp = tf.nn.softmax(logits, dim) * tf.nn.log_softmax(logits, dim)
  return -tf.reduce_sum(plogp, dim)
```

KL-Divergence

```
def gaussian_kl(q, p=(0., 0.)):
  """Computes KL divergence between two isotropic Gaussian distributions.

  To ensure numerical stability, this op uses mu, log(sigma^2) to represent
  the distribution. If q is not provided, it's assumed to be unit Gaussian.
```

```
  Args:
    q: A tuple (mu, log(sigma^2)) representing a multi-variatie Gaussian.
    p: A tuple (mu, log(sigma^2)) representing a multi-variatie Gaussian.
  Returns:
    A tensor representing KL(q, p).
  """
  mu1, log_sigma1_sq = q
  mu2, log_sigma2_sq = p
  return tf.reduce_sum(
    0.5 * (log_sigma2_sq - log_sigma1_sq +
           tf.exp(log_sigma1_sq - log_sigma2_sq) +
           tf.square(mu1 - mu2) / tf.exp(log_sigma2_sq) -
           1), axis=-1)
```

Make parallel

```
def make_parallel(fn, num_gpus, **kwargs):
  """Parallelize given model on multiple gpu devices.

  Args:
    fn: Arbitrary function that takes a set of input tensors and outputs a
        single tensor. First dimension of inputs and output tensor are assumed
        to be batch dimension.
    num_gpus: Number of GPU devices.
    **kwargs: Keyword arguments to be passed to the model.
  Returns:
    A tensor corresponding to the model output.
  """
  in_splits = {}
  for k, v in kwargs.items():
    in_splits[k] = tf.split(v, num_gpus)

  out_split = []
  for i in range(num_gpus):
    with tf.device(tf.DeviceSpec(device_type="GPU", device_index=i)):
      with tf.variable_scope(tf.get_variable_scope(), reuse=i > 0):
        out_split.append(fn(**{k : v[i] for k, v in in_splits.items()}))

  return tf.concat(out_split, axis=0)
```

Leaky relu

```
def leaky_relu(tensor, alpha=0.1):
    """Computes the leaky rectified linear activation."""
    return tf.maximum(tensor, alpha * tensor)
```

Batch normalization

```
def batch_normalization(tensor, training=False, epsilon=0.001, momentum=0.9,
                        fused_batch_norm=False, name=None):
  """Performs batch normalization on given 4-D tensor.

  The features are assumed to be in NHWC format. Noe that you need to
  run UPDATE_OPS in order for this function to perform correctly, e.g.:

  with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
    train_op = optimizer.minimize(loss)

  Based on: https://arxiv.org/abs/1502.03167
  """
  with tf.variable_scope(name, default_name="batch_normalization"):
    channels = tensor.shape.as_list()[-1]
    axes = list(range(tensor.shape.ndims - 1))

    beta = tf.get_variable(
      'beta', channels, initializer=tf.zeros_initializer())
    gamma = tf.get_variable(
      'gamma', channels, initializer=tf.ones_initializer())
```

```
    avg_mean = tf.get_variable(
      "avg_mean", channels, initializer=tf.zeros_initializer(),
      trainable=False)
    avg_variance = tf.get_variable(
      "avg_variance", channels, initializer=tf.ones_initializer(),
      trainable=False)

    if training:
      if fused_batch_norm:
        mean, variance = None, None
      else:
        mean, variance = tf.nn.moments(tensor, axes=axes)
    else:
      mean, variance = avg_mean, avg_variance

    if fused_batch_norm:
      tensor, mean, variance = tf.nn.fused_batch_norm(
        tensor, scale=gamma, offset=beta, mean=mean, variance=variance,
        epsilon=epsilon, is_training=training)
    else:
      tensor = tf.nn.batch_normalization(
        tensor, mean, variance, beta, gamma, epsilon)

    if training:
      update_mean = tf.assign(
        avg_mean, avg_mean * momentum + mean * (1.0 - momentum))
      update_variance = tf.assign(
        avg_variance, avg_variance * momentum + variance * (1.0 - momentum))

      tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_mean)
      tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_variance)

  return tensor
```

Squeeze and excitation

```
def squeeze_and_excite(tensor, ratio=16, name=None):
  """Apply squeeze/excite on given 4-D tensor.

  Based on: https://arxiv.org/abs/1709.01507
  """
  with tf.variable_scope(name, default_name="squeeze_and_excite"):
    original = tensor
    units = tensor.shape.as_list()[-1]
    tensor = tf.reduce_mean(tensor, [1, 2], keep_dims=True)
    tensor = tf.layers.dense(tensor, units / ratio, use_bias=False)
    tensor = tf.nn.relu(tensor)
    tensor = tf.layers.dense(tensor, units, use_bias=False)
    tensor = tf.nn.sigmoid(tensor)
    tensor = original * tensor
  return tensor
```