tf_ops Documentation

Release 0.0.1

Fergal Cotter

Sep 27, 2017

Contents:

CHAPTER 1

Fergal's TF Ops

This library provides some convenience functions for doing some common operations in tensorflow. I recommend you also look at the tf.layers module, as there is a lot of overlap; this module aims to fill in the gaps that exist in tf.layers package.

If you are using tensorflow on a shared GPU server and want to control how many GPUs it grabs, have a look py3nvml, in particular the `py3nvml.grab_gpus()` function.

Installation

Direct install from github (useful if you use pip freeze). To get the master branch, try:

```
$ pip install -e git+https://github.com/fbcotter/tf_ops#egg=tf_ops
```

or for a specific tag (e.g. 0.0.1), try:

```
$ pip install -e git+https://github.com/fbcotter/tf_ops.git@0.0.1#egg=tf_ops
```

Download and pip install from Git:

```
$ git clone https://github.com/fbcotter/tf_ops
$ cd tf_ops
$ pip install -r requirements.txt
$ pip install -e .
```

I would recommend to download and install (with the editable flag), as it is likely you'll want to tweak things/add functions more quickly than I can handle pull requests.

Further documentation

There is more documentation available online and you can build your own copy via the Sphinx documentation system:

```
$ python setup.py build_sphinx
```

Compiled documentation may be found in `build/docs/html/` (index.html will be the homepage)

CHAPTER 2

TF Layers Example

After a little bit of looking at tf.layers, I have realized that the functionality it implements is very good, but the documentation for it is quite minimal (and leaves lots of gaps). However, looking at the source makes things much clearer.

In fact, there is quite a bit of functionality that is not available to you if you use the functional api from tf.layers. If you instead use the underlying classes:

- tensorflow.python.layers.convolutional.Conv2D
- tensorflow.python.layers.core.Dense
- tensorflow.python.layers.core.Dropout
- tensorflow.python.layers.normalization.BatchNormalization

you can get access to lots of helpful properties.

Convolution

For example, let us define a convolutional layer like so:

```
import tensorflow as tf, numpy as np
from tensorflow.python.ops import init_ops
from tensorflow.python.layers import convolutional
x = 255 * np.random.rand(1, 50, 50, 3).astype(np.float32)
v = tf.Variable(x)
# Use glorot initialization
init = init_ops.VarianceScaling(scale=1.0, mode='fan_out')
# Use l2 regularization
reg = tf.nn.l2_loss
# Create an object representing the layer
conv_layer = convolutional.Conv2D(
    out_filters, kernel_size=3, padding='same',
    kernel_initializer=init, kernel_regularizer=reg, name='conv')
```

```
# Now get the outputs
y = conv_layer.apply(x)
```

Now, we may want to get the weights that were defined to add some variable summaries, or maybe we want to inspect the losses. Now we can do so by looking at the properties of the *conv_layer*:

```
weights = conv_layer.trainable_weights
variables = conv_layer.variables
loss = conv_layer.losses
```

Batch Norm

I wanted to include an example of batch norm, as there are a few things to be careful about. In particular, the apply method has a parameter *training*. We can see the importance of this with an example:

```
import tensorflow as tf, numpy as np
from tensorflow.python.ops import init_ops
from tensorflow.python.layers import normalization
x = 255 * np.random.rand(50, 50, 3).astype(np.float32)
v = tf.Variable(x)

bn_layer1 = normalization.BatchNormalization(name='bn1')
bn_layer2 = normalization.BatchNormalization(name='bn2')
y1 = bn_layer1.apply(v, training=True)
y2 = bn_layer2.apply(v, training=False)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
y1_n, y2_n = sess.run([y1, y2])

print('Input mean and std: {:.2f}, {:.2f}'.format(np.mean(x), np.std(x)))
print('y1 mean and std: {:.2f}, {:.2f}'.format(np.mean(y1_n), np.std(y1_n)))
print('y2 mean and std: {:.2f}, {:.2f}'.format(np.mean(y2_n), np.std(y2_n)))
```

Will have output:

```
Input mean and std: 126.41, 74.26
y1 mean and std: -0.00, 1.00
y2 mean and std: 126.34, 74.22
```

This is because batch norm will subtract the batch mean and divide by the batch standard deviation during training time to approximate an estimate on the population mean and standard deviation. In this case we only had one example, so that meant it got zero centred.

Similarly, for test time, the batch norm layer will want to subtract the population mean and divide by the population standard deviation. When we start training, these values are initialized to 0 and 1 respectively. When training, the moving_mean and moving_variance need to be updated. By default the update ops are placed in *tf.GraphKeys.UPDATE_OPS*, so they need to be added as a dependency to the *train_op*. For example:

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

CHAPTER 3

Function List

Layer Functions

The following functions are high level convenience functions. They will create weights, do the intended layer, and can add regularizations, non-linearities, batch-norm and other helpful features. A collection of helper tf functions.

tf_ops.**residual**(*x*, *filters*, *stride=1*, *train=True*, *wd=0.0001*)

Residual layer

Uses the _residual_core function to create F(x), then adds x to it.

Parameters

- **x** (*tf tensor*) – Input to be modified
- **filters** (*int*) – Number of output filters (will be used for all convolutions in the resnet core).
- **stride** (*int*) – Conv stride
- **train** (*bool or tf boolean tensor*) – Whether we are in the train phase or not. Can set to a tensorflow tensor so that it can be modified on the fly.
- **wd** (*float*) – Weight decay term for the convolutional weights

Notes

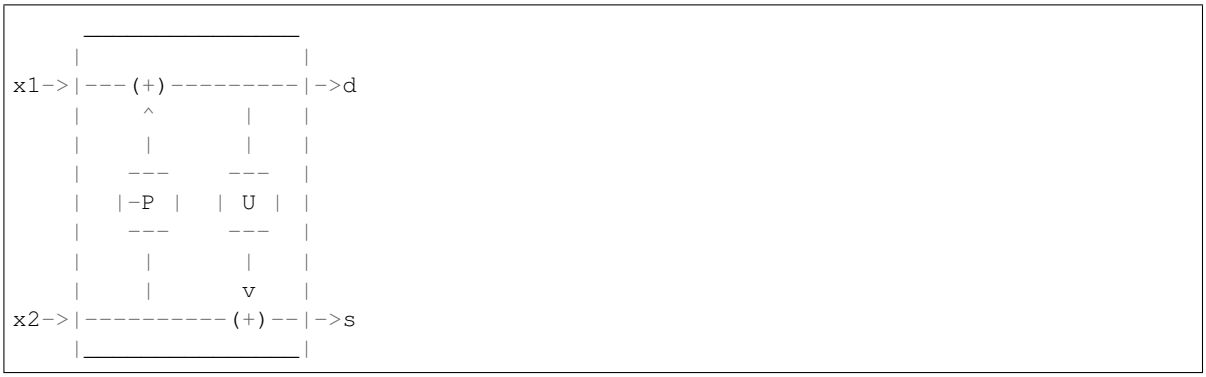
When training, the moving_mean and moving_variance need to be updated. By default the update ops are placed in tf.GraphKeys.UPDATE_OPS, so they need to be added as a dependency to the train_op. For example:

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

tf_ops.**lift_residual**(*x1*, *x2*, *train=True*, *wd=0.0001*)

Define a Lifting Layer

The P and the U blocks for this lifting layer are non-linear functions. These are the same form as the F(x) in a residual layer (i.e. two convolutions). In block form, a lifting layer looks like this:



Parameters

- **x1** (*tf tensor*) – Input tensor 1
- **x2** (*tf tensor*) – Input tensor 2
- **train** (*bool or tf boolean tensor*) – Whether we are in the train phase or not. Can set to a tensorflow tensor so that it can be modified on the fly.
- **wd** (*float*) – Weight decay term for the convolutional weights

Returns

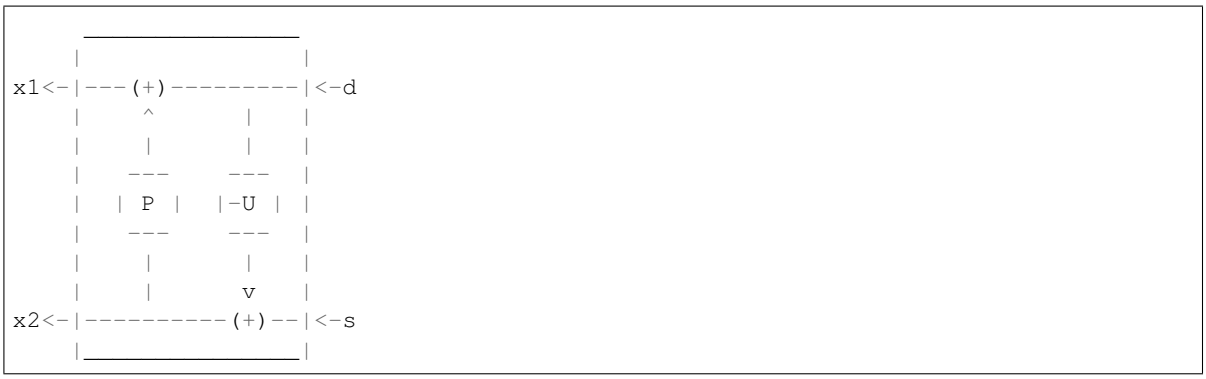
- **d** (*tf tensor*) – Detail coefficients
- **s** (*tf tensor*) – Scale coefficients

tf_ops.**lift_residual_inv**(*d*, *s*, *train=True*, *wd=0.0001*)

Define the inverse of a lifting layer

We share the variables with the forward lifting.

In block form, the inverse lifting layer looks like this (note the sign swap and flow direction reversal compared to the forward case):



Parameters

- **d** (*tf tensor*) – Input tensor 1
- **s** (*tf tensor*) – Input tensor 2
- **filters** (*int*) – Number of output channels for Px2 and Ud

- **train** (*bool or tf boolean tensor*) – Whether we are in the train phase or not. Can set to a tensorflow tensor so that it can be modified on the fly.
- **wd** (*float*) – Weight decay term for the convolutional weights

Returns

- **x1** (*tf tensor*) – Reconstructed x1
- **x2** (*tf tensor*) – Reconstructed x2

tf_ops.**complex_convolution**(*x*, *output_dim*, *size=3*, *stride=1*, *stddev=None*, *wd=0.0*, *norm=1.0*, *name='conv2d'*, *with_bias=False*, *bias_start=0.0*)

Function to do complex convolution

In a similar way we have a convenience function, `convolution()` to wrap tf.nn.conv2d (create variables, add a relu, etc.), this function wraps *cconv2d()*. If you want more fine control over things, use cconv2d directly, but for most purposes, this function should do what you need. Adds the variables to tf.GraphKeys.REGULARIZATION_LOSSES if the wd parameter is positive.

Parameters

- **x** (`tf.Tensor`) – The input variable
- **output_dim** (*int*) – number of filters to have
- **size** (*int*) – kernel spatial support
- **stride** (*int*) – what stride to use for convolution
- **stddev** (*None or positive float*) – Initialization stddev. If set to None, will use *get_xavier_stddev()*
- **wd** (*None or positive float*) – What weight decay to use
- **norm** (*positive float*) – Which regularizer to apply. E.g. norm=2 uses L2 regularization, and norm=p adds $wd \times ||w||_p^p$ to the REGULARIZATION_LOSSES. See *real_reg()*.
- **name** (*str*) – The tensorflow variable scope to create the variables under
- **with_bias** (*bool*) – add a bias after convolution? (this will be ignored if batch norm is used)
- **bias_start** (*complex float*) – If a bias is used, what to initialize it to.

Returns **y** – Result of applying complex convolution to x

Return type `tf.Tensor`

tf_ops.**complex_convolution_transpose**(*x*, *output_dim*, *shape*, *size=3*, *stride=1*, *stddev=None*, *wd=0.0*, *norm=1*, *name='conv2d'*)

Function to do the conjugate transpose of complex convolution

In a similar way we have a convenience function, `convolution()` to wrap tf.nn.conv2d (create variables, add a relu, etc.), this function wraps *cconv2d_transpose()*. If you want more fine control over things, use cconv2d_transpose directly, but for most purposes, this function should do what you need. Adds the variables to tf.GraphKeys.REGULARIZATION_LOSSES if the wd parameter is positive.

We do not subtract the bias after doing the transpose convolution.

Parameters

- **x** (`tf.Tensor`) – The input variable
- **output_dim** (*int*) – number of filters to have

- **output_shape** (*list-like or 1-d Tensor*) – list/tensor representing the output shape of the deconvolution op
- **size** (*int*) – kernel spatial support
- **stride** (*int*) – what stride to use for convolution
- **stddev** (*None or positive float*) – Initialization stddev. If set to None, will use *get_xavier_stddev()*
- **wd** (*None or positive float*) – What weight decay to use
- **norm** (*positive float*) – Which regularizer to apply. E.g. norm=2 uses L2 regularization, and norm=p adds $wd \times ||w||_p^p$ to the REGULARIZATION_LOSSES. See *real_reg()*.
- **name** (*str*) – The tensorflow variable scope to create the variables under

Returns **y** – Result of applying complex convolution transpose to x

Return type tf.Tensor

Initializers and Regularizers

The following functions are helpers to initialize weights and add regularizers to them. A collection of helper tf functions.

tf_ops.**variable_with_wd**(*name*, *shape*, *stddev=None*, *wd=None*, *norm=2*)

Helper to create an initialized variable with weight decay.

Note that the variable is initialized with a truncated normal distribution. A weight decay is added only if one is specified. Also will add summaries for this variable.

Internally, it calls tf.get_variable, so you can use this to re-get already defined variables (so long as the reuse scope is set to true). If it re-fetches an already existing variable, it will not add regularization again.

Parameters

- **name** (*str*) – name of the variable
- **shape** (*list of ints*) – shape of the variable you want to create
- **stddev** (*positive float or None*) – standard deviation of a truncated Gaussian
- **wd** (*positive float or None*) – add L2Loss weight decay multiplied by this float. If None, weight decay is not added for this variable.
- **norm** (*positive float*) – Which regularizer to apply. E.g. norm=2 uses L2 regularization, and norm=p adds $wd \times ||w||_p^p$ to the REGULARIZATION_LOSSES. See *real_reg()*.

Returns **out**

Return type variable tensor

tf_ops.**get_xavier_stddev**(*shape*, *uniform=False*, *factor=1.0*, *mode='FAN_AVG'*)

Get the correct stddev for a set of weights

When initializing a deep network, it is in principle advantageous to keep the scale of the input variance constant, so it does not explode or diminish by reaching the final layer. This initializer use the following formula:

```
if mode='FAN_IN': # Count only number of input connections.
    n = fan_in
elif mode='FAN_OUT': # Count only number of output connections.
    n = fan_out
elif mode='FAN_AVG': # Average number of inputs and output connections.
    n = (fan_in + fan_out)/2.0
    truncated_normal(shape, 0.0, stddev=sqrt(factor/n))
```

•To get Delving Deep into Rectifiers, use:

```
factor=2.0
mode='FAN_IN'
uniform=False
```

•To get Convolutional Architecture for Fast Feature Embedding , use:

```
factor=1.0
mode='FAN_IN'
uniform=True
```

•To get Understanding the difficulty of training deep feedforward neural networks use:

```
factor=1.0
mode='FAN_AVG'
uniform=True
```

•To get *xavier_initializer* use either:

```
factor=1.0
mode='FAN_AVG'
uniform=True
```

or:

```
factor=1.0
mode='FAN_AVG'
uniform=False
```

Parameters

- **factor** (*float*) – A multiplicative factor.
- **mode** (*str*) – ‘FAN_IN’, ‘FAN_OUT’, ‘FAN_AVG’.
- **uniform** (*bool*) – Whether to use uniform or normal distributed random initialization.
- **seed** (*int*) – Used to create random seeds. See tf.set_random_seed for behaviour.
- **dtype** (*tf.dtype*) – The data type. Only floating point types are supported.

Returns **out** – The stddev/limit to use that generates tensors with unit variance.

Return type float

Raises

- ValueError : if *dtype* is not a floating point type.
- TypeError : if *mode* is not in [’FAN_IN’, ‘FAN_OUT’, ‘FAN_AVG’].

tf_ops.**real_reg**(*w*, *wd=0.01*, *norm=2*)

Apply regularization on real weights

norm can be any positive float. Of course the most commonly used values would be 2 and 1 (for L2 and L1 regularization), but you can experiment by making it some value in between. A value of p returns:

$$wd \times \sum_i ||w_i||_p^p$$

Parameters

- **w** (tf.Tensor) – The weights to regularize
- **wd** (*positive float, optional (default=0.01)*) – Regularization parameter
- **norm** (*positive float, optional (default=2)*) – The norm to use for regularization. E.g. set norm=1 for the L1 norm.

Returns **reg_loss** – The loss. This method does not add anything to the REGULARIZATION_LOSSES collection. The calling function needs to do that.

Return type tf.Tensor

Raises ValueError : If norm is less than 0

tf_ops.**complex_reg**(*w*, *wd=0.01*, *norm=1*)

Apply regularization on complex weights.

norm can be any positive float. Of course the most commonly used values would be 2 and 1 (for L2 and L1 regularization), but you can experiment by making it some value in between. A value of p returns:

$$wd \times \sum_i ||w_i||_p^p$$

Parameters

- **w** (tf.Tensor (dtype=complex)) – The weights to regularize
- **wd** (*positive float, optional (default=0.01)*) – Regularization parameter
- **norm** (*positive float, optional (default=1)*) – The norm to use for regularization. E.g. set norm=1 for the L1 norm.

Returns **reg_loss** – The loss. This method does not add anything to the REGULARIZATION_LOSSES collection. The calling function needs to do that.

Return type tf.Tensor

Raises ValueError : If norm is less than 0

Notes

Can call this function with real weights too, making it perhaps a better de-facto function to call, as it able to handle both cases.

Losses and Summaries

A collection of helper tf functions.

tf_ops.**loss**(*labels*, *logits*, *λ=1*)
Compute sum of data + regularization losses.

loss = data_loss + λ * reg_losses

The regularization loss will sum over all the variables that already exist in the GraphKeys.REGULARIZATION_LOSSES.

Parameters

- **Y** (*ndarray(dtype=float, ndim=(N,C))*) – The vector of labels. It must be a one-hot vector
- λ (*float*) – Multiplier to use on all regularization losses. Be careful not to apply things twice, as all the functions in this module typically set regularization losses at a block level (for more fine control). For this reason it defaults to 1, but can be useful to set to some other value to get quick scaling of loss terms.

Returns **losses** – For optimization, only need to use the first element in the tuple. I return the other two for displaying purposes.

Return type tuple of (*loss*, data_loss, reg_loss)

tf_ops.**variable_summaries**(*var*, *name='summaries'*)
Attach a lot of summaries to a variable (for TensorBoard visualization).

Parameters

- **var** (tf.Tensor) – variable for which you wish to create summaries
- **name** (*str*) – scope under which you want to add your summary ops

Core Functions

Some new functions I wrote to do things tensorflow currently doesn't do. A collection of helper tf functions.

tf_ops.**cconv2d**(*x*, *w*, ***kwargs*)
Performs convolution with complex inputs and weights

Need to create the weights and feed to this function. If you want to have this done for you automatically, use *complex_convolution()*.

Parameters

- **x** (*tf tensor*) – input tensor
- **w** (*tf tensor*) – weights tensor
- **kwargs** (*(key, val) pairs*) – Same as tf.nn.conv2d

Returns **y** – Result of applying convolution to x

Return type tf.Tensor

Notes

Uses tf.nn.conv2d which I believe is actually cross-correlation.

tf_ops.**cconv2d_transpose**(*y*, *w*, *output_shape*, ***kwargs*)
Performs transpose convolution with complex outputs and weights.

Need to create the weights and feed to this function. If you want to have this done for you automatically, use *complex_convolution_transpose()*.

Parameters

- **x** (*tf tensor*) – input tensor
- **w** (*tf tensor*) – weights tensor
- **kwargs** (*(key, val) pairs*) – Same as tf.nn.conv2d_transpose

Notes

Takes the complex conjugate of w before doing convolution. Uses tf.nn.conv2d_transpose which I believe is actually convolution.

Returns **y** – Result of applying convolution to x

Return type tf.Tensor

tf_ops.**separable_conv_with_pad**(*x*, *h_row*, *h_col*, *stride=1*)

Function to do spatial separable convolution.

The filter weights must already be defined. It will use symmetric extension before convolution.

Parameters

- **x** (tf.Tensor of shape [Batch, height, width, c]) – The input variable. Should be of shape
- **h_row** (*tf tensor of shape [1, l, c_in, c_out]*) – The spatial row filter
- **h_col** (*tf tensor of shape [l, 1, c_in, c_out]*) – The column filter.
- **stride** (*int*) – What stride to use on the convolution.

Returns **y** – Result of applying convolution to x

Return type tf.Tensor

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index