tf_ops Documentation

Release 0.0.1

Fergal Cotter

Nov 30, 2017

Contents:

1 Fergal's TF Ops **1**
- 1.1 Installation . . . 1
- 1.2 Further documentation . . . 1

2 TF Layers Example **3**
- 2.1 Convolution . . . 3
- 2.2 Batch Norm . . . 4

3 Function List **5**
- 3.1 Layer Functions . . . 5
- 3.2 Initializers and Regularizers . . . 8
- 3.3 Losses and Summaries . . . 11
- 3.4 Core Functions . . . 11
- 3.5 Wavelet Functions . . . 12

4 Indices and tables **19**

Python Module Index **21**

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.

1.1 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.

1.2 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.

2.1 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
```

2.2 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

3.1 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.general.`**`residual`**(*x*, *filters*, *kernel_size=3*, *stride=1*, *train=True*, *wd=0.0*, *bn_momentum=0.99*, *bn_epsilon=0.001*, *name='res'*)

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
- **bn_momentum** (*float*) – The momentum for the batch normalization layers in the resnet
- **bn_epsilon** (*float*) – The epsilon for the batch normalization layers in the resnet

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.general.`**`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

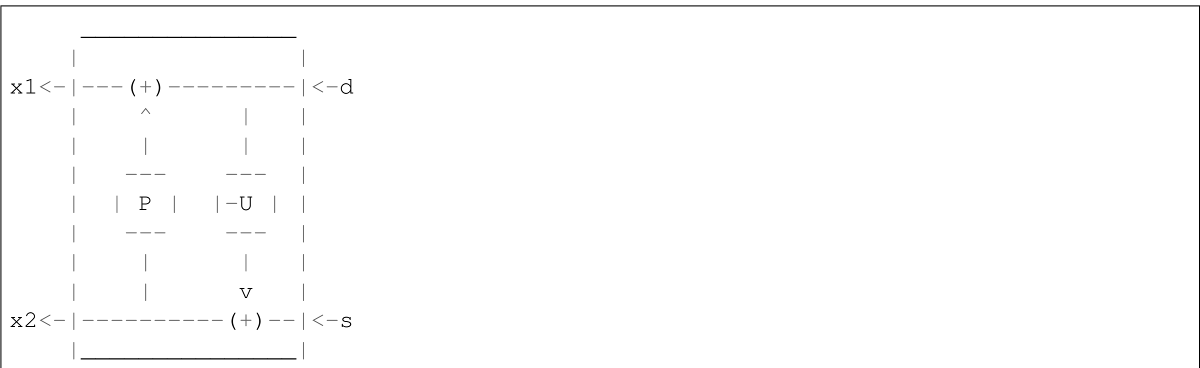
- **d** (*tf tensor*) – Detail coefficients
- **s** (*tf tensor*) – Scale coefficients

`tf_ops.general.`**`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.general.**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.general.**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`

3.2 Initializers and Regularizers

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

`tf_ops.general.`**`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.general.`**`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.general.`**`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.general.`**`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.

3.3 Losses and Summaries

A collection of helper tf functions.

`tf_ops.general.`**`loss`**(*labels*, *logits*, *one_hot=True*, *num_classes=None*, *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

- **`labels`** (*`ndarray (dtype=float, ndim=(N,C))`*) – The vector of labels.
- **`one_hot`** (*`bool`*) – True if the labels input is one_hot.
- **`num_classes`** (*`int`*) – Needed if the labels aren't one-hot already.
- **`logits`** (*`tf.Variable`*) – Logit outputs from the neural net.
- 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.general.`**`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

3.4 Core Functions

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

`tf_ops.general.`**`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.general.`**`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.general.`**`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, 1, c_in, c_out]*) – The spatial row filter
- **h_col** (*tf tensor of shape [1, 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`

3.5 Wavelet Functions

On top of the above general functions, there are also some DTCWT based wavelet functions.

`tf_ops.wave_ops.`**`lazy_wavelet`**(*x*)

Performs a lazy wavelet split on a tensor.

Designed to work nicely with the lifting blocks.

Output will have 4 times as many channels as the input, but will have one quarter the spatial size. I.e. if x is a tensor of size (batch, h, w, c), then the output will be of size (batch, h/2, w/2 4*c). The first c channels will be the A samples:

```
Input image

A B A B A B ...
C D C D C D ...
A B A B A B ...
...
```

Then the next c channels will be from the B channels, and so on.

Notes

If the spatial size is not even, then will mirror to make it even before downsampling.

Parameters **x** (*tf tensor*) – Input to apply lazy wavelet transform to.

Returns **y** – Result after applying transform.

Return type tf tensor

`tf_ops.wave_ops.`**`lazy_wavelet_inv`**(*x*, *out_size=None*)

Performs the inverse of a lazy wavelet transform - a 'lazy recombine'

Designed to work nicely with the lifting blocks.

Output will have 1/4 as many channels as the input, but will have one quadruple the spatial size. I.e. if x is a tensor of size (batch, h, w, c), then the output will be of size (batch, 2*h, 2*w c/4). If we call the first c channels the A group, then the second c the B group, and so on, the output image will be interleaved like so:

```
Output image

A B A B A B ...
C D C D C D ...
A B A B A B ...
...
```

Notes

If the forward lazy wavelet needed padding, then we should be undoing it here. For this, specify the out_size of the resulting tensor.

Parameters

- **x** (*tf tensor*) – Input to apply lazy wavelet transform to.
- **out_size** (*tuple of 4 ints or None*) – What the output size should be of the resulting tensor. The batch size will be ignored, but the spatial and channel size need to be correct. For an input spatial size of (h, r), the spatial dimensions (the 2nd and 3rd numbers in the tuple) should be either (2*h, 2*r), (2*h-1, 2*r), (2*h, 2*r-1) or (2*h-1, 2*r-1). Will raise a ValueError if not one of these options. Can also be None, in which (2*h, 2*r) is assumed. The channel size should be 1/4 of the input channel size.

Returns **y** – Result after applying transform.

Return type tf tensor

Raises

- ValueError when the out_size is invalid, or if the input tensor's channel

- dimension is not divisible by 4.

`tf_ops.wave_ops.`**`phase`**`(`*z*`)`

Calculate the elementwise arctan of z, choosing the quadrant correctly.

Quadrant I: arctan(y/x) Qaudrant II: π + arctan(y/x) (phase of x<0, y=0 is π) Quadrant III: -π + arctan(y/x) Quadrant IV: arctan(y/x)

Parameters **z** (*tf.complex64 datatype of any shape*) –

Returns **y** – Angle of z

Return type tf.float32

`tf_ops.wave_ops.`**`filter_across`**`(`*X*, *H*, *inv=False*`)`

Do 1x1 convolution as a matrix product.

Parameters

- **X** (*tf tensor of shape (batch, .., 12)*) – The input. Must have final dimension 12 - corresponding to the 6 orientations of the DTCWT and their conjugates.
- **H** (*tf tensor of shape (12, 12)*) – The filter.
- **inv** (*bool*) – True if this is the inv operation. If inverse, we first take the transpose conjugate of H. This way you can call the filter_across function with the same H for the fwd and inverse. Default is False.

Returns

Return type y = XH

`tf_ops.wave_ops.`**`up_with_zeros`**`(`*x*, *y*`)`

Upsample tensor by inserting zeros at new positions.

This keeps the input sparse in the new domain. For the moment, only works on square inputs.

Parameters

- **x** (*tf tensor*) – The input
- **y** (*int*) – The upsample rate (can only do 2 or 4 currently)

`tf_ops.wave_ops.`**`add_conjugates`**`(`*X*`)`

Concatenate tensor with its conjugates.

The DTCWT returns an array of 6 orientations for each coordinate, corresponding to the 6 orientations of [15, 45, 75, 105, 135, 165]. We can get another 6 rotations by taking complex conjugates of these.

Parameters **X** (*tf tensor of shape (batch, .., 6)*) –

Returns **Y**

Return type tf tensor of shape (batch, ... 12)

`tf_ops.wave_ops.`**`collapse_conjugates`**`(`*X*`)`

Invert the add_conjugates function. I.e. go from 12 dims to 6

To collapse, we add the first 6 dimensions with the conjugate of the last 6 and then divide by 2.

Parameters **X** (*tf tensor of shape (batch, .., 12)*) –

Returns **Y**

Return type tf tensor of shape (batch, ... 6)

`tf_ops.wave_ops.`**`response_normalization`**(*x*, *power=2*)

Function to spread out the activations.

The aim is to keep the top activation as it is, and send the others towards zero. We can do this by mapping our data through a polynomial function, ax^power. We adjust a so that the max value remains unchanged.

Negative inputs should not happen as we are competing after a magnitude operation. However, they can sometimes occur due to upsampling that may happen. If this is the case, we clip them to 0.

`tf_ops.wave_ops.`**`wavelet`**(*x*, *nlevels*, *biort='near_sym_b_bp'*, *qshift='qshift_b_bp'*, *data_format='nhwc'*)

Perform an nlevel dtcwt on the input data.

Parameters

- **x** (*tf tensor of shape (batch, h, w) or (batch, h, w, c)*) – The input to be transformed. If the input has a channel dimension, the dtcwt will be applied to each of the channels independently.
- **nlevels** (*int*) – the number of scales to use. 0 is a special case, if nlevels=0, then we return a lowpassed version of the x, and Yh and Yscale will be empty lists
- **biort** (*str*) – which biorthogonal filters to use. 'near_sym_b_bp' are my favourite, as they have 45° and 135° filters with the same period as the others.
- **qshift** (*str*) – which quarter shift filters to use. These should match up with the biorthogonal used. 'qshift_b_bp' are my favourite for the same reason.
- **data_format** (*str*) – An optional string of the form "nchw" or "nhwc" (for 4D data), or "nhw" or "hwn" (for 3D data). This specifies the data format of the input. E.g. If format is "nchw" (the default), then data is in the form [batch, channels, h, w]. If the format is "nhwc", then the data is in the form [batch, h, w, c].

Returns

out –

- Lowpass is a tensor of the lowpass data. This is a real float. If x has shape [batch, height, width, channels], the dtcwt will be applied independently for each channel and combined.
- Highpasses is a list of length <nlevels>, each entry has the six orientations of wavelet coefficients for the given scale. These are returned as tf.complex64 data type.
- Scales is a list of length <nlevels>, each entry containing the lowpass signal that gets passed to the next level of the dtcwt transform.

Return type a tuple of (lowpass, highpasses and scales)

`tf_ops.wave_ops.`**`wavelet_inv`**(*Yl*, *Yh*, *biort='near_sym_b_bp'*, *qshift='qshift_b_bp'*, *data_format='nhwc'*)

Perform an nlevel inverse dtcwt on the input data.

Parameters

- **Yl** (`tf.Tensor`) – Real tensor of shape (batch, h, w) or (batch, h, w, c) holding the lowpass input. If the shape has a channel dimension, then c inverse dtcwt's will be performed (the other inputs need to also match this shape).
- **Yh** (list(`tf.Tensor`)) – A list of length nlevels. Each entry has the high pass for the scales. Shape has to match Yl, with a 6 on the end.
- **biort** (*str*) – Which biorthogonal filters to use. 'near_sym_b_bp' are my favourite, as they have 45° and 135° filters with the same period as the others.

- **qshift** (*str*) – Which quarter shift filters to use. These should match up with the biorthogonal used. 'qshift_b_bp' are my favourite for the same reason.
- **data_format** (*str*) – An optional string of the form "nchw" or "nhwc" (for 4D data), or "nhw" or "hwn" (for 3D data). This specifies the data format of the input. E.g. If format is "nchw" (the default), then data is in the form [batch, channels, h, w]. If the format is "nhwc", then the data is in the form [batch, h, w, c].

Returns **X** – An input of size [batch, h', w'], where h' and w' will be larger than h and w by a factor of 2**nlevels

Return type `tf.Tensor`

`tf_ops.wave_ops.`**`combine_channels`**(*x*, *dim*, *combine_weights=None*)

Sum over over the specified dimension with optional summing weights.

Parameters

- **x** (*tf tensor*) – Tensor which will be summed over
- **dim** (*int*) – which dimension to sum over
- **combine_weights** (*None or list of floats*) – The weights to use when summing. If left as none, the weights will be 1.

Returns **Y**

Return type `tf.Tensor` of shape one less than x

`tf_ops.wave_ops.`**`complex_mag`**(*x*, *bias_start=0.0*, *learnable_bias=False*, *combine_channels=False*, *combine_weights=None*, *return_direction=False*)

Perform wavelet magnitude operation on complex highpass outputs.

Will subtract a bias term from the sum of the real and imaginary parts squared, before taking the square root. I.e. y=max(^{2}+^{2}-b^2, 0). This bias can be learnable or set.

Parameters

- **x** (`tf.Tensor`) – Tf tensor of shape (batch, h, w, ..., 6)
- **bias_start** (*float*) – What the b term will be set to to begin with.
- **learnable_bias** (*bool*) – If true, bias will be a tensorflow variable, and will be added to the trainable variables list. Bias will have shape: [channels, 6] if the input had channels, or simply [6] if it did not.
- **combine_channels** (*bool*) – Whether to combine the channels magnitude or not. If true, the output will be [batch, height, width, 6]. Combination will be done by simply summing up the square roots. I.e.:

  $$({}_1^2+{}_1^2+{}_2^2+{}_2^2+...+c^2+c^2 - b^2)$$

  In this situation, the bias will have shape [6]
- **combine_weights** (*bool*) – A list of weights used to combine channels. This must be of the same length as channels. If omitted, the channels will be combined equally.
- **return_direction** (*bool*) – If true, also return a unit magnitude direction vector

Returns **out** – Tensor of same shape as input (unless combine_channels was True), which is the magnitude of the real and imaginary components. Will return a tuple of (abs(in), angle(in)) if return_phase is True)

Return type tuple of (abs(x),) or (abs(x), unit(x))

`tf_ops.wave_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 variable of shape [Batch, height, width, c]*) – The input variable. Should be of shape
- **h_row** (*tf tensor of shape [1, 1, c_in, c_out]*) – The spatial row filter
- **h_col** (*tf tensor of shape [1, 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 variable

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

tf_ops.general, 11
tf_ops.wave_ops, 12

Index

A

add_conjugates() (in module tf_ops.wave_ops), 14

C

cconv2d() (in module tf_ops.general), 11
cconv2d_transpose() (in module tf_ops.general), 12
collapse_conjugates() (in module tf_ops.wave_ops), 14
combine_channels() (in module tf_ops.wave_ops), 16
complex_convolution() (in module tf_ops.general), 7
complex_convolution_transpose() (in module tf_ops.general), 7
complex_mag() (in module tf_ops.wave_ops), 16
complex_reg() (in module tf_ops.general), 10

F

filter_across() (in module tf_ops.wave_ops), 14

G

get_xavier_stddev() (in module tf_ops.general), 9

L

lazy_wavelet() (in module tf_ops.wave_ops), 12
lazy_wavelet_inv() (in module tf_ops.wave_ops), 13
lift_residual() (in module tf_ops.general), 6
lift_residual_inv() (in module tf_ops.general), 6
loss() (in module tf_ops.general), 11

P

phase() (in module tf_ops.wave_ops), 14

R

real_reg() (in module tf_ops.general), 10
residual() (in module tf_ops.general), 5
response_normalization() (in module tf_ops.wave_ops), 14

S

separable_conv_with_pad() (in module tf_ops.general), 12
separable_conv_with_pad() (in module tf_ops.wave_ops), 16

T

tf_ops.general (module), 5, 8, 11
tf_ops.wave_ops (module), 12

U

up_with_zeros() (in module tf_ops.wave_ops), 14

V

variable_summaries() (in module tf_ops.general), 11
variable_with_wd() (in module tf_ops.general), 8

W

wavelet() (in module tf_ops.wave_ops), 15
wavelet_inv() (in module tf_ops.wave_ops), 15