nn.batchnorm2d equivalent for tf.compat.v1

I'll help you find the TensorFlow equivalent for PyTorch's nn.BatchNorm2d. Let me search for the most up-to-date information about this.

tf.compat.v1 equivalent to pytorch nn.batchnorm2d 10 results

Let me check for more specific information about the TensorFlow equivalent to PyTorch's nn.BatchNorm2d.

tf.compat.v1 batch_normalization for 2D convolutional networks 10 results

Let me check one more specific document to get the details about the batch normalization implementation in TensorFlow:

Fetched https://www.tensorflow.org/api_docs/python/tf/compat/v1/layers/BatchNormalization ⚠

Let me try accessing a different URL:

Fetched https://www.tensorflow.org/api_docs/python/tf/compat/v1/layers/batch_normalization ⚠

Based on my research, I can provide you with the TensorFlow v1 equivalent to PyTorch's nn.BatchNorm2d.

TensorFlow Equivalent to PyTorch's nn.BatchNorm2d

In TensorFlow 1.x compatibility mode (`tf.compat.v1`), there are two main ways to implement batch normalization that are equivalent to PyTorch's `nn.BatchNorm2d`:

Option 1: tf.compat.v1.layers.BatchNormalization

```python
# For TensorFlow 1.x compatibility
import tensorflow as tf
```

```
# Apply batch normalization after a 2D convolutional layer
conv = tf.compat.v1.layers.conv2d(inputs=x, filters=32, kernel_size=[3, 3], padding='same')
bn = tf.compat.v1.layers.BatchNormalization()(conv, training=is_training)
```

Option 2: tf.compat.v1.layers.batch_normalization

```python
# For TensorFlow 1.x compatibility
import tensorflow as tf

# Apply batch normalization after a 2D convolutional layer
conv = tf.compat.v1.layers.conv2d(inputs=x, filters=32, kernel_size=[3, 3], padding='same')
bn = tf.compat.v1.layers.batch_normalization(inputs=conv, training=is_training)
```

Key Parameters Comparison

PyTorch's `nn.BatchNorm2d` parameters have slightly different names in TensorFlow:

PyTorch Parameter	TensorFlow Equivalent	Description
num_features	(inferred from input)	Number of features/channels
eps	epsilon	Small constant for numerical stability
momentum	momentum	Momentum for moving average
affine	scale/center	If True (scale=True/center=True), has learnable parameters
track_running_stats	renorm	Controls tracking of running statistics

Complete Example

Here's a more complete example of using batch normalization in a convolutional network with TensorFlow v1 compatibility:

```python
```

```python
import tensorflow as tf

def conv_bn_relu(inputs, filters, kernel_size, is_training):
    # Convolutional layer
    conv = tf.compat.v1.layers.conv2d(
        inputs=inputs,
        filters=filters,
        kernel_size=kernel_size,
        padding='same',
        use_bias=False  # Often set to False when using batch norm
    )

    # Batch normalization
    bn = tf.compat.v1.layers.batch_normalization(
        inputs=conv,
        momentum=0.9,  # Similar to PyTorch default of 0.1 (TF uses 1-momentum)
        epsilon=1e-5,  # Similar to PyTorch default
        center=True,   # Similar to PyTorch's affine=True (for beta)
        scale=True,    # Similar to PyTorch's affine=True (for gamma)
        training=is_training
    )

    # Activation
    return tf.nn.relu(bn)

# Usage in a model
def create_model(inputs, is_training):
    # First conv block
    net = conv_bn_relu(inputs, filters=32, kernel_size=[3, 3], is_training=is_training)
    # Additional layers...
    return net
```

This implementation provides functionality equivalent to PyTorch's nn.BatchNorm2d for 2D convolutional networks in TensorFlow's v1 compatibility mode.

How is that different than this?

```
mean, variance = tf.nn.moments(convolved, [3], keep_dims=True)
            normalized = (convolved - mean) / tf.sqrt(variance + 1e-5)
            gamma = tf.get_variable(f'lngamma{i}', [1, 1, 1, nfilt],
initializer=tf.ones_initializer(), dtype=dtype)
            beta = tf.get_variable(f'lnbeta{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(),
dtype=dtype)
            normalized = normalized * gamma + beta
            # Then use normalized in pooling
            pool_shape = [1, ptime, pband, 1]
            pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape,
padding='SAME')

######################################################################
##########
            # TensorFlow 1.x equivalent of PyTorch's nn.BatchNorm2d
            normalized = tf.compat.v1.layers.batch_normalization(convolved)
            pool_shape = [1, ptime, pband, 1]
            pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape,
padding='SAME')
```

Looking at the code you shared, I notice there are two different normalization approaches being compared:

Layer Normalization vs. Batch Normalization

The first block of code implements **Layer Normalization**:

```python
mean, variance = tf.nn.moments(convolved, [3], keep_dims=True)
normalized = (convolved - mean) / tf.sqrt(variance + 1e-5)
gamma = tf.get_variable(f'ln_gamma_{i}', [1, 1, 1, nfilt], initializer=tf.ones_initializer(), dtype=dtype)
beta = tf.get_variable(f'ln_beta_{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(), dtype=dtype)
normalized = normalized * gamma + beta
```

The second block uses **Batch Normalization**:

```python
normalized = tf.compat.v1.layers.batch_normalization(convolved)
```

Here are the key differences:

1. **Normalization Dimension**:
    - The first code block computes statistics along dimension `[3]` (the channel dimension), which is typical of Layer Normalization
    - Batch Normalization (in the second block) typically computes statistics across the batch dimension for each feature map
2. **Implementation Complexity**:
    - Your Layer Normalization is manually implemented with low-level operations
    - The Batch Normalization uses TensorFlow's built-in implementation which handles many details automatically
3. **Parameters and Tracking**:
    - Your Layer Normalization manually creates gamma/beta parameters with explicit shapes and initializers
    - The Batch Normalization call automatically creates parameters and handles tracking of running statistics for inference
4. **Missing Arguments**:
    - The Batch Normalization call is missing the `training` parameter, which is crucial to distinguish between training and inference modes

- It's also missing parameters like `momentum` for tracking running statistics

Proper BatchNorm2d Implementation

For a more complete and correct equivalent to PyTorch's `nn.BatchNorm2d`, the TensorFlow implementation should include:

```python
normalized = tf.compat.v1.layers.batch_normalization(
    inputs=convolved,
    momentum=0.9,           # Equivalent to PyTorch's 0.1 (TF uses 1-momentum)
    epsilon=1e-5,           # Small constant for numerical stability
    center=True,            # Use beta parameter (PyTorch's affine=True)
    scale=True,             # Use gamma parameter (PyTorch's affine=True)
    training=is_training,   # CRITICAL: Must indicate training vs inference mode
    fused=True              # Use fused implementation for performance when possible
)
```

The main issue with the second block in your code is it doesn't specify the `training` flag, which means the behavior will be inconsistent between training and inference. In PyTorch's `nn.BatchNorm2d`, this distinction is handled when you call `model.train()` or `model.eval()`.

G

Please let me know what needs to be fixed with the following code:

```
nn_output = feats_audio
        if do_cnn:
            layer_last = feats_audio
            nfilt_last = audio_nchannels
            for i, ((ntime, nband, nfilt), (ptime, pband)) in enumerate(zip(cnn_filter_shapes,
cnn_pool)):
                layername = 'cnn{}'.format(i)
                with tf.variable_scope(layer_name):
                    filters = tf.get_variable('filters', [ntime, nband, nfilt_last, nfilt], initializer=cnn_init,
dtype=dtype)
                    biases = tf.get_variable('biases', [nfilt], initializer=tf.constant_initializer(0.1),
dtype=dtype)
                if cnn_rnn_zack:
                    padding = 'SAME'
                else:
                    padding = 'VALID'
                conv = tf.nn.conv2d(layer_last, filters, [1, 1, 1, 1], padding=padding)
                biased = tf.nn.bias_add(conv, biases)
                convolved = tf.nn.relu(biased)

                if do_batch_norm:

##########################################################################
###########
                    # Simpler alternative: Layer normalization
                    # Normalize across channels for each example
                    mean, variance = tf.nn.moments(convolved, [3], keep_dims=True)
                    normalized = (convolved - mean) / tf.sqrt(variance + 1e-5)
                    gamma = tf.get_variable(f'lngamma{i}', [1, 1, 1, nfilt],
initializer=tf.ones_initializer(), dtype=dtype)
```

```
                beta = tf.get_variable(f'lnbeta{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(),
dtype=dtype)
                normalized = normalized * gamma + beta
                # Then use normalized in pooling
                pool_shape = [1, ptime, pband, 1]
                pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape,
padding='SAME')

################################################################################
##########
                # TensorFlow 1.x equivalent of PyTorch's nn.BatchNorm2d
                normalized = tf.compat.v1.layers.batch_normalization(convolved)
                pool_shape = [1, ptime, pband, 1]
                pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape,
padding='SAME')
            else:
                pool_shape = [1, ptime, pband, 1]
                pooled = tf.nn.max_pool(convolved, ksize=pool_shape, strides=pool_shape,
padding='SAME')
#               print('{}: {}'.format(layer_name, pooled.get_shape()))
            export_feat_tensors[layer_name] = pooled
            # Apply CNN dropout after activation and pooling
            if mode == 'train' and cnn_keep_prob < 1.0:
                pooled = tf.nn.dropout(pooled, cnn_keep_prob)
#                   print('{}_dropout: {}'.format(layer_name, pooled.get_shape()))
            layer_last = pooled
            nfilt_last = nfilt
        cnn_output = layer_last
```

Looking at your code, I've identified several issues that need to be fixed with the batch normalization implementation:

Issues to Fix:

1. **Duplicate Normalization**: You're applying both layer normalization and batch normalization to the same input, but only using one of the results. Both normalizations are being calculated, but the second one (batch normalization) overwrites the result from the first one.
2. **Missing Training Parameter**: The batch normalization call doesn't include the crucial `training` parameter, which is needed to determine whether to use batch statistics (during training) or running statistics (during inference).
3. **Missing Momentum Parameter**: Without specifying momentum, the default value may not match what you expect from PyTorch's nn.BatchNorm2d (which uses 0.1 by default).
4. **No Running Statistics Update Control**: There's no mechanism to update/use running statistics appropriately during training/inference.
5. **Scope Issues**: Variable creation for batch normalization should ideally be within a variable scope for better organization.

Fixed Code:

```python
nn_output = feats_audio
if do_cnn:
    layer_last = feats_audio
    nfilt_last = audio_nchannels
    for i, ((ntime, nband, nfilt), (ptime, pband)) in enumerate(zip(cnn_filter_shapes, cnn_pool)):
        layer_name = 'cnn_{}'.format(i)
        with tf.variable_scope(layer_name):
            filters = tf.get_variable('filters', [ntime, nband, nfilt_last, nfilt], initializer=cnn_init, dtype=dtype)
            biases = tf.get_variable('biases', [nfilt], initializer=tf.constant_initializer(0.1), dtype=dtype)
        if cnn_rnn_zack:
            padding = 'SAME'
        else:
            padding = 'VALID'
        conv = tf.nn.conv2d(layer_last, filters, [1, 1, 1, 1], padding=padding)
        biased = tf.nn.bias_add(conv, biases)
        convolved = tf.nn.relu(biased)

        if do_batch_norm:
            # Choose EITHER layer norm OR batch norm (not both)
            if use_layer_norm:  # Define this flag elsewhere in your code
                ##############################################################################
                # Layer normalization implementation
                with tf.variable_scope(f'layer_norm_{i}'):
                    mean, variance = tf.nn.moments(convolved, [3], keep_dims=True)
                    normalized = (convolved - mean) / tf.sqrt(variance + 1e-5)
                    gamma = tf.get_variable('ln_gamma', [1, 1, 1, nfilt],
                                           initializer=tf.ones_initializer(), dtype=dtype)
                    beta = tf.get_variable('ln_beta', [1, 1, 1, nfilt],
                                          initializer=tf.zeros_initializer(), dtype=dtype)
                    normalized = normalized * gamma + beta
            else:
                ##############################################################################
                # Proper TensorFlow 1.x equivalent of PyTorch's nn.BatchNorm2d
                with tf.variable_scope(f'batch_norm_{i}'):
                    # is_training should be a placeholder or a boolean indicating training mode
                    is_training = (mode == 'train')  # Assuming 'mode' is defined elsewhere
                    normalized = tf.compat.v1.layers.batch_normalization(
                        inputs=convolved,
                        momentum=0.9,          # Equivalent to PyTorch's 0.1 (TF uses 1-momentum)
                        epsilon=1e-5,
                        center=True,           # Use beta parameter
                        scale=True,            # Use gamma parameter
                        training=is_training,  # CRITICAL: Must indicate training vs inference mode
                        fused=True             # Use fused implementation for performance when possible
                    )

            # Then use normalized in pooling
            pool_shape = [1, ptime, pband, 1]
            pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape, padding='SAME')
        else:
            pool_shape = [1, ptime, pband, 1]
            pooled = tf.nn.max_pool(convolved, ksize=pool_shape, strides=pool_shape, padding='SAME')

        export_feat_tensors[layer_name] = pooled
        # Apply CNN dropout after activation and pooling
        if mode == 'train' and cnn_keep_prob < 1.0:
            pooled = tf.nn.dropout(pooled, cnn_keep_prob)
        layer_last = pooled
        nfilt_last = nfilt
    cnn_output = layer_last
```

Key Improvements:

1. **Conditional Logic**: Added a flag to choose between layer normalization and batch normalization instead of computing both.
2. **Variable Scopes**: Added proper variable scopes for both normalization types to better organize the variables.
3. **Training Flag**: Properly set the `training` parameter based on the model's mode.
4. **Complete Parameters**: Added all relevant parameters to match PyTorch's nn.BatchNorm2d behavior.
5. **Fused Implementation**: Enabled the fused implementation for better performance when possible.

By making these changes, your batch normalization will work correctly in both training and inference modes, making it properly equivalent to PyTorch's nn.BatchNorm2d.

G

```
Creating train model
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call
last)
Cell In[2], line 131
    128     vars(training_args).update(training_args_dict)
    130     print(f"Starting run {run_idx+1}/{num_runs}: {run_id}")
--> 131     train_onset_model(training_args, OnsetModel=OnsetNet_Gina5)
    132 else:
    133     print(f"Not running {run_idx+1}/{num_runs}: {run_id}")

File /users/6/croal008/5527 Project/ddc/learn/onset_train.py:198, in
train_onset_model(args, OnsetModel)
    196     print('Creating train model')
    197     with tf.variable_scope('model_sp', reuse=None):
--> 198         model_train = OnsetModel(mode='train',
target_weight_strategy=args.weight_strategy, batch_size=args.batch_size,
**model_config)
    200 if do_train_eval or do_eval or do_cnn_export:
    201     with tf.variable_scope('model_sp', reuse=do_train):

File /users/6/croal008/5527 Project/ddc/learn/onset_net_gina5.py:119, in
OnsetNet_Gina5.__init__(self, mode, batch_size, audio_context_radius,
audio_nbands, audio_nchannels, nfeats, cnn_filter_shapes, cnn_init,
cnn_pool, cnn_rnn_zack, rnn_cell_type, rnn_size, rnn_nlayers, rnn_init,
rnn_nunroll, rnn_keep_prob, dnn_sizes, dnn_init, dnn_keep_prob,
dnn_nonlin, target_weight_strategy, grad_clip, opt, export_feat_name,
zack_hack, cnn_keep_prob, input_keep_prob, do_batch_norm, do_attention)
    102                 convolved = tf.nn.relu(biased)
    104                 if do_batch_norm:
```

```
    105
##########################################################################
####
    106 #                     # Simpler alternative: Layer normalization
   (...)
    117
##########################################################################
####
    118                     # TensorFlow 1.x equivalent of PyTorch's
nn.BatchNorm2d
--> 119                     normalized =
tf.compat.v1.layers.batch_normalization(convolved, training=is_training)
    121                     pool_shape = [1, ptime, pband, 1]
    122                     pooled = tf.nn.max_pool(normalized,
ksize=pool_shape, strides=pool_shape, padding='SAME')

File
/common/software/install/manual/jupyter_kernels/DeepLearning/lib/python3.1
1/site-packages/tensorflow/python/util/lazy_loader.py:207, in
KerasLazyLoader.__getattr__(self, item)
    200     raise AttributeError(
    201         "`tf.compat.v2.keras` is not available with Keras 3. Just
use "
    202         "`import keras` instead."
    203     )
    204   elif self._tfll_submodule and self._tfll_submodule.startswith(
    205       "__internal__.legacy."
    206   ):
--> 207     raise AttributeError(
    208         f"`{item}` is not available with Keras 3."
    209     )
    210 module = self._load()
    211 return getattr(module, item)

AttributeError: `batch_normalization` is not available with Keras 3.
```

I see the issue now. The error is showing that `tf.compat.v1.layers.batch_normalization` is not available with Keras 3, which is the version used in your environment. This is a compatibility issue that needs to be addressed.

The Problem

The error message is clear:

```
AttributeError: `batch_normalization` is not available with Keras 3.
```

This happens because newer versions of TensorFlow have fully integrated Keras, and the legacy compatibility functions have changed or been removed in Keras 3.

Solution

Here are three alternative approaches you can use to fix this issue:

Option 1: Use tf.keras.layers.BatchNormalization directly

```python
python

# Replace this:
normalized = tf.compat.v1.layers.batch_normalization(convolved, training=is_training)

# With this:
normalized = tf.keras.layers.BatchNormalization()(convolved, training=is_training)
```

Option 2: Use the lower-level tf.nn functions

```python
python

# Replace with direct tf.nn implementation
# Define these variables somewhere in your variable scope
```

```
scale = tf.get_variable(f'bn_scale_{i}', [1, 1, 1, nfilt], initializer=tf.ones_initializer(), dtype=dtype)
offset = tf.get_variable(f'bn_offset_{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(), dtype=dtype)
moving_mean = tf.get_variable(f'bn_mean_{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(),
                             dtype=dtype, trainable=False)
moving_var = tf.get_variable(f'bn_var_{i}', [1, 1, 1, nfilt], initializer=tf.ones_initializer(),
                            dtype=dtype, trainable=False)

# During training
if is_training:
    batch_mean, batch_var = tf.nn.moments(convolved, axes=[0, 1, 2], keep_dims=True)
    update_mean = tf.assign(moving_mean, moving_mean * 0.9 + batch_mean * 0.1)
    update_var = tf.assign(moving_var, moving_var * 0.9 + batch_var * 0.1)
    with tf.control_dependencies([update_mean, update_var]):
        normalized = tf.nn.batch_normalization(
            convolved, batch_mean, batch_var, offset, scale, 1e-5)
else:
    # During inference
    normalized = tf.nn.batch_normalization(
        convolved, moving_mean, moving_var, offset, scale, 1e-5)
```

Option 3: Use your layer normalization implementation

Since you already have a working layer normalization implementation, you could use that instead if batch normalization isn't strictly required:

python

```
# Use your existing layer normalization code
mean, variance = tf.nn.moments(convolved, [3], keep_dims=True)
normalized = (convolved - mean) / tf.sqrt(variance + 1e-5)
gamma = tf.get_variable(f'ln_gamma_{i}', [1, 1, 1, nfilt], initializer=tf.ones_initializer(), dtype=dtype)
beta = tf.get_variable(f'ln_beta_{i}', [1, 1, 1, nfilt], initializer=tf.zeros_initializer(), dtype=dtype)
normalized = normalized * gamma + beta
```

Recommended Solution

I recommend Option 1 as it's the cleanest approach for newer TensorFlow versions. Here's how your updated code should look:

```python
# In your __init__ method
if do_batch_norm:
    # Define training flag
    is_training = (mode == 'train')

    # Use tf.keras.layers.BatchNormalization
    bn_layer = tf.keras.layers.BatchNormalization(
        momentum=0.9,  # Equivalent to PyTorch's 0.1
        epsilon=1e-5,
        center=True,
        scale=True
    )
    normalized = bn_layer(convolved, training=is_training)

    pool_shape = [1, ptime, pband, 1]
    pooled = tf.nn.max_pool(normalized, ksize=pool_shape, strides=pool_shape, padding='SAME')
else:
    pool_shape = [1, ptime, pband, 1]
    pooled = tf.nn.max_pool(convolved, ksize=pool_shape, strides=pool_shape, padding='SAME')
```

This approach is compatible with newer TensorFlow versions while providing the same functionality as the original code.