# pylint: disable=no-member, attribute-defined-outside-init, duplicate-code

"""

Contains the twml.layers.SparseMaxNorm layer.

"""

from .layer import Layer

from libtwml import OPLIB

import tensorflow.compat.v1 as tf

import twml

class SparseMaxNorm(Layer):

"""

Computes a max-normalization and adds bias to the sparse\_input,

forwards that through a sparse affine transform followed

by an non-linear activation on the resulting dense representation.

This layer has two parameters, one of which learns through gradient descent:

bias\_x (optional):

vector of shape [input\_size]. Learned through gradient descent.

max\_x:

vector of shape [input\_size]. Holds the maximas of input ``x`` for normalization.

Either calibrated through SparseMaxNorm calibrator, or calibrated online, or both.

The pseudo-code for this layer looks like:

.. code-block:: python

abs\_x = abs(x)

normed\_x = clip\_by\_value(x / max\_x, -1, 1)

biased\_x = normed\_x + bias\_x

return biased

Args:

max\_x\_initializer:

initializer vector of shape [input\_size] used by variable `max\_x`

bias\_x\_initializer:

initializer vector of shape [input\_size] used by parameter `bias\_x`

is\_training:

Are we training the layer to learn the normalization maximas.

If set to True, max\_x will be able to learn. This is independent of bias\_x

epsilon:

The minimum value used for max\_x. Defaults to 1E-5.

use\_bias:

Default True. Set to False to not use a bias term.

Returns:

A layer representing the output of the sparse\_max\_norm transformation.

"""

def \_\_init\_\_(

self,

input\_size=None,

max\_x\_initializer=None,

bias\_x\_initializer=None,

is\_training=True,

epsilon=1E-5,

use\_bias=True,

\*\*kwargs):

super(SparseMaxNorm, self).\_\_init\_\_(\*\*kwargs)

if input\_size:

raise ValueError('input\_size is deprecated - it is now automatically \

inferred from your input.')

if max\_x\_initializer is None:

max\_x\_initializer = tf.zeros\_initializer()

self.max\_x\_initializer = max\_x\_initializer

self.\_use\_bias = use\_bias

if use\_bias:

if bias\_x\_initializer is None:

bias\_x\_initializer = tf.zeros\_initializer()

self.bias\_x\_initializer = bias\_x\_initializer

self.epsilon = epsilon

self.is\_training = is\_training

def build(self, input\_shape): # pylint: disable=unused-argument

"""Creates the max\_x and bias\_x tf.Variables of the layer."""

self.max\_x = self.add\_variable(

'max\_x',

initializer=self.max\_x\_initializer,

shape=[input\_shape[1]],

dtype=tf.float32,

trainable=False)

if self.\_use\_bias:

self.bias\_x = self.add\_variable(

'bias\_x',

initializer=self.bias\_x\_initializer,

shape=[input\_shape[1]],

dtype=tf.float32,

trainable=True)

self.built = True

def compute\_output\_shape(self, input\_shape):

"""Computes the output shape of the layer given the input shape.

Args:

input\_shape: A (possibly nested tuple of) `TensorShape`. It need not

be fully defined (e.g. the batch size may be unknown).

Raises NotImplementedError.

"""

raise NotImplementedError

def \_call(self, inputs, \*\*kwargs): # pylint: disable=unused-argument

"""

The forward propagation logic of the layer lives here.

Arguments:

sparse\_input:

A 2D ``tf.SparseTensor`` of dense\_shape ``[batch\_size, input\_size]``

Returns:

A ``tf.SparseTensor`` representing the output of the max\_norm transformation, this can

be fed into twml.layers.FullSparse in order to be transformed into a ``tf.Tensor``.

"""

if isinstance(inputs, twml.SparseTensor):

inputs = inputs.to\_tf()

elif not isinstance(inputs, tf.SparseTensor):

raise TypeError("The inputs must be of type tf.SparseTensor or twml.SparseTensor")

indices\_x = inputs.indices[:, 1]

values\_x = inputs.values

if self.is\_training is False:

normalized\_x = OPLIB.sparse\_max\_norm\_inference(self.max\_x,

indices\_x,

values\_x,

self.epsilon)

update\_op = tf.no\_op()

else:

max\_x, normalized\_x = OPLIB.sparse\_max\_norm\_training(self.max\_x,

indices\_x,

values\_x,

self.epsilon)

update\_op = tf.assign(self.max\_x, max\_x)

with tf.control\_dependencies([update\_op]):

normalized\_x = tf.stop\_gradient(normalized\_x)

# add input bias

if self.\_use\_bias:

normalized\_x = normalized\_x + tf.gather(self.bias\_x, indices\_x)

# convert back to sparse tensor

return tf.SparseTensor(inputs.indices, normalized\_x, inputs.dense\_shape)

def call(self, inputs, \*\*kwargs): # pylint: disable=unused-argument

"""

The forward propagation logic of the layer lives here.

Arguments:

sparse\_input:

A 2D ``tf.SparseTensor`` of dense\_shape ``[batch\_size, input\_size]``

Returns:

A ``tf.SparseTensor`` representing the output of the max\_norm transformation, this can

be fed into twml.layers.FullSparse in order to be transformed into a ``tf.Tensor``.

"""

with tf.device(self.max\_x.device):

return self.\_call(inputs, \*\*kwargs)

# For backwards compatiblity and also because I don't want to change all the tests.

MaxNorm = SparseMaxNorm

def sparse\_max\_norm(inputs,

input\_size=None,

max\_x\_initializer=None,

bias\_x\_initializer=None,

is\_training=True,

epsilon=1E-5,

use\_bias=True,

name=None,

reuse=None):

"""

Functional inteface to SparseMaxNorm.

Args:

inputs:

A sparse tensor (can be twml.SparseTensor or tf.SparseTensor)

input\_size:

number of input units

max\_x\_initializer:

initializer vector of shape [input\_size] used by variable `max\_x`

bias\_x\_initializer:

initializer vector of shape [input\_size] used by parameter `bias\_x`

is\_training:

Are we training the layer to learn the normalization maximas.

If set to True, max\_x will be able to learn. This is independent of bias\_x

epsilon:

The minimum value used for max\_x. Defaults to 1E-5.

use\_bias:

Default True. Set to False to not use a bias term.

Returns:

Output after normalizing with the max value.

"""

if input\_size:

raise ValueError('input\_size is deprecated - it is now automatically \

inferred from your input.')

if isinstance(inputs, twml.SparseTensor):

inputs = inputs.to\_tf()

layer = SparseMaxNorm(max\_x\_initializer=max\_x\_initializer,

bias\_x\_initializer=bias\_x\_initializer,

is\_training=is\_training,

epsilon=epsilon,

use\_bias=use\_bias,

name=name,

\_scope=name,

\_reuse=reuse)

return layer(inputs)