"""

Contains the twml.layers.ZscoreNormalization layer.

"""

from twml.layers.layer import Layer

import tensorflow.compat.v1 as tf

from tensorflow.python.training import moving\_averages

# This is copied from tensorflow.contrib.framework.python.ops.add\_model\_variable in 1.15

# Not available in 2.x

# TODO: Figure out if this is really necessary.

def \_add\_model\_variable(var):

"""Adds a variable to the `GraphKeys.MODEL\_VARIABLES` collection.

Args:

var: a variable.

"""

if var not in tf.get\_collection(tf.GraphKeys.MODEL\_VARIABLES):

tf.add\_to\_collection(tf.GraphKeys.MODEL\_VARIABLES, var)

def update\_moving\_variable(batch\_var, moving\_var, decay, zero\_debias=True, name=None):

update\_op = moving\_averages.assign\_moving\_average(

moving\_var, batch\_var, decay, zero\_debias=zero\_debias, name=None)

\_add\_model\_variable(moving\_var)

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

return tf.identity(moving\_var)

class ZscoreNormalization(Layer):

"""

Perform z-score normalization using moving mean and std.

Missing values are not included during mean/std calculation

This layer should only be used right after input layer.

Args:

decay:

using large decay to include longer moving means.

data\_type:

use float64 to prevent overflow during variance calculation.

name:

Layer name

Returns:

A layer representing the output of the ZscoreNormalization transformation.

"""

def \_\_init\_\_(

self,

decay=0.9999,

data\_type=tf.float64,

name=None,

\*\*kwargs):

super(ZscoreNormalization, self).\_\_init\_\_(name=name, \*\*kwargs)

self.epsilon = tf.constant(1., data\_type)

self.decay = decay

self.data\_type = data\_type

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

"""Creates the moving\_mean and moving\_var tf.Variables of the layer."""

input\_dim = input\_shape[1]

self.moving\_mean = self.add\_variable(

'{}\_mean/EMA'.format(self.name),

initializer=tf.constant\_initializer(),

shape=[input\_dim],

dtype=self.data\_type,

trainable=False

)

self.moving\_var = self.add\_variable(

'{}\_variance/EMA'.format(self.name),

initializer=tf.constant\_initializer(),

shape=[input\_dim],

dtype=self.data\_type,

trainable=False

)

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).

"""

return input\_shape

def \_training\_pass(self, input, dense\_mask, input\_dtype, handle\_single, zero\_debias):

epsilon = self.epsilon

moving\_mean, moving\_var = self.moving\_mean, self.moving\_var

# calculate the number of exisiting value for each feature

tensor\_batch\_num = tf.reduce\_sum(tf.cast(dense\_mask, self.data\_type), axis=0)

mask\_ones = tf.cast(tensor\_batch\_num, tf.bool)

eps\_vector = tf.fill(tf.shape(tensor\_batch\_num), epsilon)

# the following filled 0 with epision

tensor\_batch\_num\_eps = tf.where(mask\_ones,

tensor\_batch\_num,

eps\_vector

)

tensor\_batch\_num\_eps\_broacast = tf.expand\_dims(tensor\_batch\_num\_eps, 0)

tensor\_batch\_divided = input / tensor\_batch\_num\_eps\_broacast

tensor\_batch\_mean = tf.reduce\_sum(tensor\_batch\_divided, axis=0)

# update moving mean here, and use it to calculate the std.

tensor\_moving\_mean = update\_moving\_variable(tensor\_batch\_mean, moving\_mean, self.decay,

zero\_debias, name="mean\_ema\_op")

tensor\_batch\_sub\_mean = input - tf.expand\_dims(tensor\_moving\_mean, 0)

tensor\_batch\_sub\_mean = tf.where(dense\_mask,

tensor\_batch\_sub\_mean,

tf.zeros\_like(tensor\_batch\_sub\_mean))

# divided by sqrt(n) before square, and then do summation for numeric stability.

broad\_sqrt\_num\_eps = tf.expand\_dims(tf.sqrt(tensor\_batch\_num\_eps), 0)

tensor\_batch\_sub\_mean\_div = tensor\_batch\_sub\_mean / broad\_sqrt\_num\_eps

tensor\_batch\_sub\_mean\_div\_square = tf.square(tensor\_batch\_sub\_mean\_div)

tensor\_batch\_var = tf.reduce\_sum(tensor\_batch\_sub\_mean\_div\_square, axis=0)

# update moving var here, dont replace 0 with eps before updating.

tensor\_moving\_var = update\_moving\_variable(tensor\_batch\_var, moving\_var, self.decay,

zero\_debias, name="var\_ema\_op")

# if std is 0, replace it with epsilon

tensor\_moving\_std = tf.sqrt(tensor\_moving\_var)

tensor\_moving\_std\_eps = tf.where(tf.equal(tensor\_moving\_std, 0),

eps\_vector,

tensor\_moving\_std)

missing\_input\_norm = tensor\_batch\_sub\_mean / tf.expand\_dims(tensor\_moving\_std\_eps, 0)

if handle\_single:

# if std==0 and value not missing, reset it to 1.

moving\_var\_mask\_zero = tf.math.equal(tensor\_moving\_var, 0)

moving\_var\_mask\_zero = tf.expand\_dims(moving\_var\_mask\_zero, 0)

missing\_input\_norm = tf.where(

tf.math.logical\_and(dense\_mask, moving\_var\_mask\_zero),

tf.ones\_like(missing\_input\_norm),

missing\_input\_norm

)

if input\_dtype != self.data\_type:

missing\_input\_norm = tf.cast(missing\_input\_norm, input\_dtype)

return missing\_input\_norm

def \_infer\_pass(self, input, dense\_mask, input\_dtype, handle\_single):

epsilon = tf.cast(self.epsilon, input\_dtype)

testing\_moving\_mean = tf.cast(self.moving\_mean, input\_dtype)

tensor\_moving\_std = tf.cast(tf.sqrt(self.moving\_var), input\_dtype)

broad\_mean = tf.expand\_dims(testing\_moving\_mean, 0)

tensor\_batch\_sub\_mean = input - broad\_mean

tensor\_batch\_sub\_mean = tf.where(dense\_mask,

tensor\_batch\_sub\_mean,

tf.zeros\_like(tensor\_batch\_sub\_mean)

)

tensor\_moving\_std\_eps = tf.where(tf.equal(tensor\_moving\_std, 0),

tf.fill(tf.shape(tensor\_moving\_std), epsilon),

tensor\_moving\_std)

missing\_input\_norm = tensor\_batch\_sub\_mean / tf.expand\_dims(tensor\_moving\_std\_eps, 0)

if handle\_single:

# if std==0 and value not missing, reset it to 1.

moving\_var\_broad = tf.expand\_dims(tensor\_moving\_std, 0)

moving\_var\_mask\_zero = tf.math.logical\_not(tf.cast(moving\_var\_broad, tf.bool))

missing\_input\_norm = tf.where(tf.math.logical\_and(dense\_mask, moving\_var\_mask\_zero),

tf.ones\_like(missing\_input\_norm),

missing\_input\_norm

)

return missing\_input\_norm

def call(

self,

input,

is\_training,

dense\_mask=None,

zero\_debias=True,

handle\_single=False):

"""

Args:

-----------

input: B x D : float32/float64

missing value must be set to 0.

is\_training: bool

training phase or testing phase

dense\_mask: B x D : bool

missing value should be marked as 0, non-missing as 1. same shape as input

zero\_debias: bool

bias correction of the moving average. (biased towards 0 in the beginning.

see adam paper. https://arxiv.org/abs/1412.6980)

handle\_single: bool

if std==0, and feature is not missing value, set the value to 1, instead of 0.

This is super rare if input only consists of continous feature.

But if one-hot feature is included,

they will all have same values 1, in that case, make sure to set handle\_single to true.

"""

if dense\_mask is None:

dense\_mask = tf.math.logical\_not(tf.equal(input, 0))

input\_dtype = input.dtype

if is\_training:

if input\_dtype != self.data\_type:

input = tf.cast(input, self.data\_type)

return self.\_training\_pass(input, dense\_mask, input\_dtype, handle\_single, zero\_debias)

else:

return self.\_infer\_pass(input, dense\_mask, input\_dtype, handle\_single)

def zscore\_normalization(

input,

is\_training,

decay=0.9999,

data\_type=tf.float64,

name=None,

dense\_mask=None,

zero\_debias=True,

handle\_single=False, \*\*kwargs):

"""

Args:

------------

input: B x D : float32/float64

missing value must be set to 0.

is\_training: bool

training phase or testing phase

decay:

using large decay to include longer moving means.

data\_type:

use float64 to zprevent overflow during variance calculation.

name:

Layer name

dense\_mask: B x D : bool

missing value should be marked as 0, non-missing as 1. same shape as input

zero\_debias: bool

bias correction of the moving average. (biased towards 0 in the beginning.

see adam paper. https://arxiv.org/abs/1412.6980)

handle\_single: bool

if std==0, and feature is not missing value, set the value to 1, instead of 0.

This is super rare if input only consists of continous feature.

But if one-hot feature is included,

they will all have same values 1, in that case, make sure to set handle\_single to true.

"""

norm\_layer = ZscoreNormalization(decay=decay, data\_type=data\_type, name=name, \*\*kwargs)

return norm\_layer(input,

is\_training,

dense\_mask=dense\_mask,

zero\_debias=zero\_debias,

handle\_single=handle\_single)