# pylint: disable=no-member, arguments-differ, attribute-defined-outside-init, unused-argument

"""

Implementing factorization Layer

"""

from twitter.deepbird.sparse.sparse\_ops import \_pad\_empty\_outputs

import tensorflow.compat.v1 as tf

import twml

from twml.layers.layer import Layer

class FactorizationMachine(Layer):

"""factorization machine layer class.

This layer implements the factorization machine operation.

The paper is "Factorization Machines" by Steffen Rendle.

TDD: go/tf-fm-tdd

Arguments:

num\_latent\_variables:

num of latent variables

The number of parameter in this layer is num\_latent\_variables x n where n is number of

input features.

weight\_initializer:

Initializer function for the weight matrix.

This argument defaults to zeros\_initializer().

This is valid when the FullSparse is the first layer of

parameters but should be changed otherwise.

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

activation:

Activation function (callable). Set it to None to maintain a linear activation.

trainable:

Boolean, if `True` also add variables to the graph collection

``GraphKeys.TRAINABLE\_VARIABLES`` (see `tf.Variable

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/Variable>`\_).

name:

String, the name of the layer. Layers with the same name will

share weights, but to avoid mistakes we require ``reuse=True`` in such cases.

use\_sparse\_grads:

Boolean, if `True` do sparse mat mul with `embedding\_lookup\_sparse`, which will

make gradients to weight matrix also sparse in backward pass. This can lead to non-trivial

speed up at training time when input\_size is large and optimizer handles sparse gradients

correctly (eg. with SGD or LazyAdamOptimizer). If weight matrix is small, it's recommended

to set this flag to `False`; for most use cases of FullSparse, however, weight matrix will

be large, so it's better to set it to `True`

use\_binary\_values:

Assume all non zero values are 1. Defaults to False.

This can improve training if used in conjunction with MDL.

This parameter can also be a list of binary values if `inputs` passed to `call` a list.

"""

def \_\_init\_\_(self,

num\_latent\_variables=10,

weight\_initializer=None,

activation=None,

trainable=True,

name=None,

use\_sparse\_grads=True,

use\_binary\_values=False,

weight\_regularizer=None,

substract\_self\_cross=True,

\*\*kwargs):

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

if weight\_initializer is None:

weight\_initializer = tf.zeros\_initializer()

self.weight\_initializer = weight\_initializer

self.num\_latent\_variables = num\_latent\_variables

self.activation = activation

self.use\_sparse\_grads = use\_sparse\_grads

self.use\_binary\_values = use\_binary\_values

self.weight\_regularizer = weight\_regularizer

self.substract\_self\_cross = substract\_self\_cross

def build(self, input\_shape):

"""

creates``weight`` Variable of shape``[input\_size, num\_latent\_variables]``.

"""

shape = [input\_shape[1], self.num\_latent\_variables]

# There is a 2GB limitation for each tensor because of protobuf.

# 2\*\*30 is 1GB. 2 \* (2\*\*30) is 2GB.

dtype = tf.as\_dtype(self.dtype)

requested\_size = input\_shape[1] \* self.num\_latent\_variables \* dtype.size

if (requested\_size >= 2\*\*31):

raise ValueError("Weight tensor can not be larger than 2GB. " %

"Requested Dimensions(%d, %d) of type %s (%d bytes total)"

(input\_shape[1], self.num\_latent\_variables, dtype.name))

if not callable(self.weight\_initializer):

shape = None

# dense tensor

self.weight = self.add\_variable(

'weight',

initializer=self.weight\_initializer,

regularizer=self.weight\_regularizer,

shape=shape,

dtype=self.dtype,

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 logic of the layer lives here.

Arguments:

inputs:

A SparseTensor

Returns:

- If `inputs` is `SparseTensor`, then returns a number with cross info

"""

# The following are given:

# - inputs is a sparse tensor, we call it sp\_x.

# - The dense\_v tensor is a dense matrix, whose row i

# corresponds to the vector V\_i.

# weights has shape [num\_features, k]

sp\_x = inputs

if isinstance(inputs, twml.SparseTensor):

sp\_x = inputs.to\_tf()

elif not isinstance(sp\_x, tf.SparseTensor):

raise TypeError("The sp\_x must be of type tf.SparseTensor or twml.SparseTensor")

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

batch\_ids = sp\_x.indices[:, 0]

values = tf.reshape(sp\_x.values, [-1, 1], name=self.name)

if self.use\_sparse\_grads:

v = tf.nn.embedding\_lookup(self.weight, indices)

# if (self.use\_binary\_values):

# values = tf.ones(tf.shape(values), dtype=values.dtype)

v\_times\_x = v \* values

# First term: Sum\_k [Sum\_i (v\_ik \* x\_i)]^2

all\_crosses = tf.segment\_sum(v\_times\_x, batch\_ids, name=self.name)

all\_crosses\_squared = tf.reduce\_sum((all\_crosses \* all\_crosses), 1)

if self.substract\_self\_cross:

# Second term: Sum\_k Sum\_i [ (v\_ik \* x\_i)^2 ]

v\_times\_x\_2 = v\_times\_x\*\*2

self\_crosses = tf.reduce\_sum(tf.segment\_sum(v\_times\_x\_2, batch\_ids, name=self.name), 1)

outputs = all\_crosses\_squared - self\_crosses

else:

outputs = all\_crosses\_squared

else:

# need to check if prediction is faster with code below

crossTerm = tf.reduce\_sum((tf.sparse\_tensor\_dense\_matmul(sp\_x, self.weight)\*\*2), 1)

if self.substract\_self\_cross:

# compute self-cross term

self\_crossTerm = tf.reduce\_sum(tf.segment\_sum((tf.gather(self.weight, indices) \* values)\*\*2, batch\_ids), 1)

outputs = crossTerm - self\_crossTerm

else:

outputs = crossTerm

if self.activation is not None:

outputs = self.activation(outputs)

outputs = tf.reshape(outputs, [-1, 1], name=self.name)

outputs = \_pad\_empty\_outputs(outputs, tf.cast(sp\_x.dense\_shape[0], tf.int32))

# set more explicit and static shape to avoid shape inference error

# valueError: The last dimension of the inputs to `Dense` should be defined. Found `None`

outputs.set\_shape([None, 1])

return outputs