import sys

import twml

from .initializer import customized\_glorot\_uniform

import tensorflow.compat.v1 as tf

import yaml

# checkstyle: noqa

def read\_config(whitelist\_yaml\_file):

with tf.gfile.FastGFile(whitelist\_yaml\_file) as f:

try:

return yaml.safe\_load(f)

except yaml.YAMLError as exc:

print(exc)

sys.exit(1)

def \_sparse\_feature\_fixup(features, input\_size\_bits):

"""Rebuild a sparse tensor feature so that its dense shape attribute is present.

Arguments:

features (SparseTensor): Sparse feature tensor of shape ``(B, sparse\_feature\_dim)``.

input\_size\_bits (int): Number of columns in ``log2`` scale. Must be positive.

Returns:

SparseTensor: Rebuilt and non-faulty version of `features`."""

sparse\_feature\_dim = tf.constant(2\*\*input\_size\_bits, dtype=tf.int64)

sparse\_shape = tf.stack([features.dense\_shape[0], sparse\_feature\_dim])

sparse\_tf = tf.SparseTensor(features.indices, features.values, sparse\_shape)

return sparse\_tf

def self\_atten\_dense(input, out\_dim, activation=None, use\_bias=True, name=None):

def safe\_concat(base, suffix):

"""Concats variables name components if base is given."""

if not base:

return base

return f"{base}:{suffix}"

input\_dim = input.shape.as\_list()[1]

sigmoid\_out = twml.layers.FullDense(

input\_dim, dtype=tf.float32, activation=tf.nn.sigmoid, name=safe\_concat(name, "sigmoid\_out")

)(input)

atten\_input = sigmoid\_out \* input

mlp\_out = twml.layers.FullDense(

out\_dim,

dtype=tf.float32,

activation=activation,

use\_bias=use\_bias,

name=safe\_concat(name, "mlp\_out"),

)(atten\_input)

return mlp\_out

def get\_dense\_out(input, out\_dim, activation, dense\_type):

if dense\_type == "full\_dense":

out = twml.layers.FullDense(out\_dim, dtype=tf.float32, activation=activation)(input)

elif dense\_type == "self\_atten\_dense":

out = self\_atten\_dense(input, out\_dim, activation=activation)

return out

def get\_input\_trans\_func(bn\_normalized\_dense, is\_training):

gw\_normalized\_dense = tf.expand\_dims(bn\_normalized\_dense, -1)

group\_num = bn\_normalized\_dense.shape.as\_list()[1]

gw\_normalized\_dense = GroupWiseTrans(group\_num, 1, 8, name="groupwise\_1", activation=tf.tanh)(

gw\_normalized\_dense

)

gw\_normalized\_dense = GroupWiseTrans(group\_num, 8, 4, name="groupwise\_2", activation=tf.tanh)(

gw\_normalized\_dense

)

gw\_normalized\_dense = GroupWiseTrans(group\_num, 4, 1, name="groupwise\_3", activation=tf.tanh)(

gw\_normalized\_dense

)

gw\_normalized\_dense = tf.squeeze(gw\_normalized\_dense, [-1])

bn\_gw\_normalized\_dense = tf.layers.batch\_normalization(

gw\_normalized\_dense,

training=is\_training,

renorm\_momentum=0.9999,

momentum=0.9999,

renorm=is\_training,

trainable=True,

)

return bn\_gw\_normalized\_dense

def tensor\_dropout(

input\_tensor,

rate,

is\_training,

sparse\_tensor=None,

):

"""

Implements dropout layer for both dense and sparse input\_tensor

Arguments:

input\_tensor:

B x D dense tensor, or a sparse tensor

rate (float32):

dropout rate

is\_training (bool):

training stage or not.

sparse\_tensor (bool):

whether the input\_tensor is sparse tensor or not. Default to be None, this value has to be passed explicitly.

rescale\_sparse\_dropout (bool):

Do we need to do rescaling or not.

Returns:

tensor dropped out"""

if sparse\_tensor == True:

if is\_training:

with tf.variable\_scope("sparse\_dropout"):

values = input\_tensor.values

keep\_mask = tf.keras.backend.random\_binomial(

tf.shape(values), p=1 - rate, dtype=tf.float32, seed=None

)

keep\_mask.set\_shape([None])

keep\_mask = tf.cast(keep\_mask, tf.bool)

keep\_indices = tf.boolean\_mask(input\_tensor.indices, keep\_mask, axis=0)

keep\_values = tf.boolean\_mask(values, keep\_mask, axis=0)

dropped\_tensor = tf.SparseTensor(keep\_indices, keep\_values, input\_tensor.dense\_shape)

return dropped\_tensor

else:

return input\_tensor

elif sparse\_tensor == False:

return tf.layers.dropout(input\_tensor, rate=rate, training=is\_training)

def adaptive\_transformation(bn\_normalized\_dense, is\_training, func\_type="default"):

assert func\_type in [

"default",

"tiny",

], f"fun\_type can only be one of default and tiny, but get {func\_type}"

gw\_normalized\_dense = tf.expand\_dims(bn\_normalized\_dense, -1)

group\_num = bn\_normalized\_dense.shape.as\_list()[1]

if func\_type == "default":

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 1, 8, name="groupwise\_1", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 8, 4, name="groupwise\_2", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 4, 1, name="groupwise\_3", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

elif func\_type == "tiny":

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 1, 2, name="groupwise\_1", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 2, 1, name="groupwise\_2", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

gw\_normalized\_dense = FastGroupWiseTrans(

group\_num, 1, 1, name="groupwise\_3", activation=tf.tanh, init\_multiplier=8

)(gw\_normalized\_dense)

gw\_normalized\_dense = tf.squeeze(gw\_normalized\_dense, [-1])

bn\_gw\_normalized\_dense = tf.layers.batch\_normalization(

gw\_normalized\_dense,

training=is\_training,

renorm\_momentum=0.9999,

momentum=0.9999,

renorm=is\_training,

trainable=True,

)

return bn\_gw\_normalized\_dense

class FastGroupWiseTrans(object):

"""

used to apply group-wise fully connected layers to the input.

it applies a tiny, unique MLP to each individual feature."""

def \_\_init\_\_(self, group\_num, input\_dim, out\_dim, name, activation=None, init\_multiplier=1):

self.group\_num = group\_num

self.input\_dim = input\_dim

self.out\_dim = out\_dim

self.activation = activation

self.init\_multiplier = init\_multiplier

self.w = tf.get\_variable(

name + "\_group\_weight",

[1, group\_num, input\_dim, out\_dim],

initializer=customized\_glorot\_uniform(

fan\_in=input\_dim \* init\_multiplier, fan\_out=out\_dim \* init\_multiplier

),

trainable=True,

)

self.b = tf.get\_variable(

name + "\_group\_bias",

[1, group\_num, out\_dim],

initializer=tf.constant\_initializer(0.0),

trainable=True,

)

def \_\_call\_\_(self, input\_tensor):

"""

input\_tensor: batch\_size x group\_num x input\_dim

output\_tensor: batch\_size x group\_num x out\_dim"""

input\_tensor\_expand = tf.expand\_dims(input\_tensor, axis=-1)

output\_tensor = tf.add(

tf.reduce\_sum(tf.multiply(input\_tensor\_expand, self.w), axis=-2, keepdims=False),

self.b,

)

if self.activation is not None:

output\_tensor = self.activation(output\_tensor)

return output\_tensor

class GroupWiseTrans(object):

"""

Used to apply group fully connected layers to the input.

"""

def \_\_init\_\_(self, group\_num, input\_dim, out\_dim, name, activation=None):

self.group\_num = group\_num

self.input\_dim = input\_dim

self.out\_dim = out\_dim

self.activation = activation

w\_list, b\_list = [], []

for idx in range(out\_dim):

this\_w = tf.get\_variable(

name + f"\_group\_weight\_{idx}",

[1, group\_num, input\_dim],

initializer=tf.keras.initializers.glorot\_uniform(),

trainable=True,

)

this\_b = tf.get\_variable(

name + f"\_group\_bias\_{idx}",

[1, group\_num, 1],

initializer=tf.constant\_initializer(0.0),

trainable=True,

)

w\_list.append(this\_w)

b\_list.append(this\_b)

self.w\_list = w\_list

self.b\_list = b\_list

def \_\_call\_\_(self, input\_tensor):

"""

input\_tensor: batch\_size x group\_num x input\_dim

output\_tensor: batch\_size x group\_num x out\_dim

"""

out\_tensor\_list = []

for idx in range(self.out\_dim):

this\_res = (

tf.reduce\_sum(input\_tensor \* self.w\_list[idx], axis=-1, keepdims=True) + self.b\_list[idx]

)

out\_tensor\_list.append(this\_res)

output\_tensor = tf.concat(out\_tensor\_list, axis=-1)

if self.activation is not None:

output\_tensor = self.activation(output\_tensor)

return output\_tensor

def add\_scalar\_summary(var, name, name\_scope="hist\_dense\_feature/"):

with tf.name\_scope("summaries/"):

with tf.name\_scope(name\_scope):

tf.summary.scalar(name, var)

def add\_histogram\_summary(var, name, name\_scope="hist\_dense\_feature/"):

with tf.name\_scope("summaries/"):

with tf.name\_scope(name\_scope):

tf.summary.histogram(name, tf.reshape(var, [-1]))

def sparse\_clip\_by\_value(sparse\_tf, min\_val, max\_val):

new\_vals = tf.clip\_by\_value(sparse\_tf.values, min\_val, max\_val)

return tf.SparseTensor(sparse\_tf.indices, new\_vals, sparse\_tf.dense\_shape)

def check\_numerics\_with\_msg(tensor, message="", sparse\_tensor=False):

if sparse\_tensor:

values = tf.debugging.check\_numerics(tensor.values, message=message)

return tf.SparseTensor(tensor.indices, values, tensor.dense\_shape)

else:

return tf.debugging.check\_numerics(tensor, message=message)

def pad\_empty\_sparse\_tensor(tensor):

dummy\_tensor = tf.SparseTensor(

indices=[[0, 0]],

values=[0.00001],

dense\_shape=tensor.dense\_shape,

)

result = tf.cond(

tf.equal(tf.size(tensor.values), 0),

lambda: dummy\_tensor,

lambda: tensor,

)

return result

def filter\_nans\_and\_infs(tensor, sparse\_tensor=False):

if sparse\_tensor:

sparse\_values = tensor.values

filtered\_val = tf.where(

tf.logical\_or(tf.is\_nan(sparse\_values), tf.is\_inf(sparse\_values)),

tf.zeros\_like(sparse\_values),

sparse\_values,

)

return tf.SparseTensor(tensor.indices, filtered\_val, tensor.dense\_shape)

else:

return tf.where(

tf.logical\_or(tf.is\_nan(tensor), tf.is\_inf(tensor)), tf.zeros\_like(tensor), tensor

)

def generate\_disliked\_mask(labels):

"""Generate a disliked mask where only samples with dislike labels are set to 1 otherwise set to 0.

Args:

labels: labels of training samples, which is a 2D tensor of shape batch\_size x 3: [OONCs, engagements, dislikes]

Returns:

1D tensor of shape batch\_size x 1: [dislikes (booleans)]

"""

return tf.equal(tf.reshape(labels[:, 2], shape=[-1, 1]), 1)