import warnings

from twml.contrib.layers import ZscoreNormalization

from ...libs.customized\_full\_sparse import FullSparse

from ...libs.get\_feat\_config import FEAT\_CONFIG\_DEFAULT\_VAL as MISSING\_VALUE\_MARKER

from ...libs.model\_utils import (

\_sparse\_feature\_fixup,

adaptive\_transformation,

filter\_nans\_and\_infs,

get\_dense\_out,

tensor\_dropout,

)

import tensorflow.compat.v1 as tf

# checkstyle: noqa

def light\_ranking\_mlp\_ngbdt(features, is\_training, params, label=None):

return deepnorm\_light\_ranking(

features,

is\_training,

params,

label=label,

decay=params.momentum,

dense\_emb\_size=params.dense\_embedding\_size,

base\_activation=tf.keras.layers.LeakyReLU(),

input\_dropout\_rate=params.dropout,

use\_gbdt=False,

)

def deepnorm\_light\_ranking(

features,

is\_training,

params,

label=None,

decay=0.99999,

dense\_emb\_size=128,

base\_activation=None,

input\_dropout\_rate=None,

input\_dense\_type="self\_atten\_dense",

emb\_dense\_type="self\_atten\_dense",

mlp\_dense\_type="self\_atten\_dense",

use\_gbdt=False,

):

# --------------------------------------------------------

# Initial Parameter Checking

# --------------------------------------------------------

if base\_activation is None:

base\_activation = tf.keras.layers.LeakyReLU()

if label is not None:

warnings.warn(

"Label is unused in deepnorm\_gbdt. Stop using this argument.",

DeprecationWarning,

)

with tf.variable\_scope("helper\_layers"):

full\_sparse\_layer = FullSparse(

output\_size=params.sparse\_embedding\_size,

activation=base\_activation,

use\_sparse\_grads=is\_training,

use\_binary\_values=False,

dtype=tf.float32,

)

input\_normalizing\_layer = ZscoreNormalization(decay=decay, name="input\_normalizing\_layer")

# --------------------------------------------------------

# Feature Selection & Embedding

# --------------------------------------------------------

if use\_gbdt:

sparse\_gbdt\_features = \_sparse\_feature\_fixup(features["gbdt\_sparse"], params.input\_size\_bits)

if input\_dropout\_rate is not None:

sparse\_gbdt\_features = tensor\_dropout(

sparse\_gbdt\_features, input\_dropout\_rate, is\_training, sparse\_tensor=True

)

total\_embed = full\_sparse\_layer(sparse\_gbdt\_features, use\_binary\_values=True)

if (input\_dropout\_rate is not None) and is\_training:

total\_embed = total\_embed / (1 - input\_dropout\_rate)

else:

with tf.variable\_scope("dense\_branch"):

dense\_continuous\_features = filter\_nans\_and\_infs(features["continuous"])

if params.use\_missing\_sub\_branch:

is\_missing = tf.equal(dense\_continuous\_features, MISSING\_VALUE\_MARKER)

continuous\_features\_filled = tf.where(

is\_missing,

tf.zeros\_like(dense\_continuous\_features),

dense\_continuous\_features,

)

normalized\_features = input\_normalizing\_layer(

continuous\_features\_filled, is\_training, tf.math.logical\_not(is\_missing)

)

with tf.variable\_scope("missing\_sub\_branch"):

missing\_feature\_embed = get\_dense\_out(

tf.cast(is\_missing, tf.float32),

dense\_emb\_size,

activation=base\_activation,

dense\_type=input\_dense\_type,

)

else:

continuous\_features\_filled = dense\_continuous\_features

normalized\_features = input\_normalizing\_layer(continuous\_features\_filled, is\_training)

with tf.variable\_scope("continuous\_sub\_branch"):

normalized\_features = adaptive\_transformation(

normalized\_features, is\_training, func\_type="tiny"

)

if input\_dropout\_rate is not None:

normalized\_features = tensor\_dropout(

normalized\_features,

input\_dropout\_rate,

is\_training,

sparse\_tensor=False,

)

filled\_feature\_embed = get\_dense\_out(

normalized\_features,

dense\_emb\_size,

activation=base\_activation,

dense\_type=input\_dense\_type,

)

if params.use\_missing\_sub\_branch:

dense\_embed = tf.concat(

[filled\_feature\_embed, missing\_feature\_embed], axis=1, name="merge\_dense\_emb"

)

else:

dense\_embed = filled\_feature\_embed

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

sparse\_discrete\_features = \_sparse\_feature\_fixup(

features["sparse\_no\_continuous"], params.input\_size\_bits

)

if input\_dropout\_rate is not None:

sparse\_discrete\_features = tensor\_dropout(

sparse\_discrete\_features, input\_dropout\_rate, is\_training, sparse\_tensor=True

)

discrete\_features\_embed = full\_sparse\_layer(sparse\_discrete\_features, use\_binary\_values=True)

if (input\_dropout\_rate is not None) and is\_training:

discrete\_features\_embed = discrete\_features\_embed / (1 - input\_dropout\_rate)

total\_embed = tf.concat(

[dense\_embed, discrete\_features\_embed],

axis=1,

name="total\_embed",

)

total\_embed = tf.layers.batch\_normalization(

total\_embed,

training=is\_training,

renorm\_momentum=decay,

momentum=decay,

renorm=is\_training,

trainable=True,

)

# --------------------------------------------------------

# MLP Layers

# --------------------------------------------------------

with tf.variable\_scope("MLP\_branch"):

assert params.num\_mlp\_layers >= 0

embed\_list = [total\_embed] + [None for \_ in range(params.num\_mlp\_layers)]

dense\_types = [emb\_dense\_type] + [mlp\_dense\_type for \_ in range(params.num\_mlp\_layers - 1)]

for xl in range(1, params.num\_mlp\_layers + 1):

neurons = params.mlp\_neuron\_scale \*\* (params.num\_mlp\_layers + 1 - xl)

embed\_list[xl] = get\_dense\_out(

embed\_list[xl - 1], neurons, activation=base\_activation, dense\_type=dense\_types[xl - 1]

)

if params.task\_name in ["Sent", "HeavyRankPosition", "HeavyRankProbability"]:

logits = get\_dense\_out(embed\_list[-1], 1, activation=None, dense\_type=mlp\_dense\_type)

else:

raise ValueError("Invalid Task Name !")

output\_dict = {"output": logits}

return output\_dict