import tensorflow.compat.v1 as tf

from tensorflow.python.ops import array\_ops, math\_ops

# Copied from metrics\_impl.py

# https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/metrics\_impl.py#L216

def safe\_div(numerator, denominator, name=None):

"""

Example usage: calculating NDCG = DCG / IDCG to handle cases when

IDCG = 0 returns 0 instead of Infinity

Do not use this dividing funciton unless it makes sense to your problem

Divides two tensors element-wise, returns 0 if the denominator is <= 0.

Args:

numerator: a real `Tensor`.

denominator: a real `Tensor`, with dtype matching `numerator`.

name: Name for the returned op.

Returns:

0 if `denominator` <= 0, else `numerator` / `denominator`

"""

t = math\_ops.truediv(numerator, denominator)

zero = array\_ops.zeros\_like(t, dtype=denominator.dtype)

condition = math\_ops.greater(denominator, zero)

zero = math\_ops.cast(zero, t.dtype)

return array\_ops.where(condition, t, zero, name=name)

def cal\_ndcg(label\_scores, predicted\_scores, top\_k\_int=1):

"""

Calculate NDCG score for top\_k\_int ranking positions

Args:

label\_scores: a real `Tensor`.

predicted\_scores: a real `Tensor`, with dtype matching label\_scores

top\_k\_int: An int or an int `Tensor`.

Returns:

a `Tensor` that holds DCG / IDCG.

"""

sorted\_labels, predicted\_order = \_get\_ranking\_orders(

label\_scores, predicted\_scores, top\_k\_int=top\_k\_int)

predicted\_relevance = \_get\_relevance\_scores(predicted\_order)

sorted\_relevance = \_get\_relevance\_scores(sorted\_labels)

cg\_discount = \_get\_cg\_discount(top\_k\_int)

dcg = \_dcg\_idcg(predicted\_relevance, cg\_discount)

idcg = \_dcg\_idcg(sorted\_relevance, cg\_discount)

# the ndcg score of the batch

# idcg is 0 if label\_scores are all 0

ndcg = safe\_div(dcg, idcg, 'one\_ndcg')

return ndcg

def cal\_swapped\_ndcg(label\_scores, predicted\_scores, top\_k\_int):

"""

Calculate swapped NDCG score in Lambda Rank for full/top k ranking positions

Args:

label\_scores: a real `Tensor`.

predicted\_scores: a real `Tensor`, with dtype matching label\_scores

top\_k\_int: An int or an int `Tensor`.

Returns:

a `Tensor` that holds swapped NDCG by .

"""

sorted\_labels, predicted\_order = \_get\_ranking\_orders(

label\_scores, predicted\_scores, top\_k\_int=top\_k\_int)

predicted\_relevance = \_get\_relevance\_scores(predicted\_order)

sorted\_relevance = \_get\_relevance\_scores(sorted\_labels)

cg\_discount = \_get\_cg\_discount(top\_k\_int)

# cg\_discount is safe as a denominator

dcg\_k = predicted\_relevance / cg\_discount

dcg = tf.reduce\_sum(dcg\_k)

idcg\_k = sorted\_relevance / cg\_discount

idcg = tf.reduce\_sum(idcg\_k)

ndcg = safe\_div(dcg, idcg, 'ndcg\_in\_lambdarank\_training')

# remove the gain from label i then add the gain from label j

tiled\_ij = tf.tile(dcg\_k, [1, top\_k\_int])

new\_ij = (predicted\_relevance / tf.transpose(cg\_discount))

tiled\_ji = tf.tile(tf.transpose(dcg\_k), [top\_k\_int, 1])

new\_ji = tf.transpose(predicted\_relevance) / cg\_discount

# if swap i and j, remove the stale cg for i, then add the new cg for i,

# remove the stale cg for j, and then add the new cg for j

new\_dcg = dcg - tiled\_ij + new\_ij - tiled\_ji + new\_ji

new\_ndcg = safe\_div(new\_dcg, idcg, 'new\_ndcg\_in\_lambdarank\_training')

swapped\_ndcg = tf.abs(ndcg - new\_ndcg)

return swapped\_ndcg

def \_dcg\_idcg(relevance\_scores, cg\_discount):

"""

Calculate DCG scores for top\_k\_int ranking positions

Args:

relevance\_scores: a real `Tensor`.

cg\_discount: a real `Tensor`, with dtype matching relevance\_scores

Returns:

a `Tensor` that holds \\sum\_{i=1}^k \frac{relevance\_scores\_k}{cg\_discount}

"""

# cg\_discount is safe

dcg\_k = relevance\_scores / cg\_discount

return tf.reduce\_sum(dcg\_k)

def \_get\_ranking\_orders(label\_scores, predicted\_scores, top\_k\_int=1):

"""

Calculate DCG scores for top\_k\_int ranking positions

Args:

label\_scores: a real `Tensor`.

predicted\_scores: a real `Tensor`, with dtype matching label\_scores

top\_k\_int: an integer or an int `Tensor`.

Returns:

two `Tensors` that hold sorted\_labels: the ground truth relevance socres

and predicted\_order: relevance socres based on sorted predicted\_scores

"""

# sort predictions\_scores and label\_scores

# size [batch\_size/num of DataRecords, 1]

label\_scores = tf.reshape(label\_scores, [-1, 1])

predicted\_scores = tf.reshape(predicted\_scores, [-1, 1])

# sorted\_labels contians the relevance scores of the correct order

sorted\_labels, ordered\_labels\_indices = tf.nn.top\_k(

tf.transpose(label\_scores), k=top\_k\_int)

sorted\_labels = tf.transpose(sorted\_labels)

# sort predicitons and use the indices to obtain the relevance scores of the predicted order

sorted\_predictions, ordered\_predictions\_indices = tf.nn.top\_k(

tf.transpose(predicted\_scores), k=top\_k\_int)

ordered\_predictions\_indices\_for\_labels = tf.transpose(ordered\_predictions\_indices)

# predicted\_order contians the relevance scores of the predicted order

predicted\_order = tf.gather\_nd(label\_scores, ordered\_predictions\_indices\_for\_labels)

return sorted\_labels, predicted\_order

def \_get\_cg\_discount(top\_k\_int=1):

r"""

Calculate discounted gain factor for ranking position till top\_k\_int

Args:

top\_k\_int: An int or an int `Tensor`.

Returns:

a `Tensor` that holds \log\_{2}(i + 1), i \in [1, k]

"""

log\_2 = tf.log(tf.constant(2.0, dtype=tf.float32))

# top\_k\_range needs to start from 1 to top\_k\_int

top\_k\_range = tf.range(top\_k\_int) + 1

top\_k\_range = tf.reshape(top\_k\_range, [-1, 1])

# cast top\_k\_range to float

top\_k\_range = tf.cast(top\_k\_range, dtype=tf.float32)

cg\_discount = tf.log(top\_k\_range + 1.0) / log\_2

return cg\_discount

def \_get\_relevance\_scores(scores):

return 2 \*\* scores - 1

def safe\_log(raw\_scores, name=None):

"""

Calculate log of a tensor, handling cases that

raw\_scores are close to 0s

Args:

raw\_scores: An float `Tensor`.

Returns:

A float `Tensor` that hols the safe log base e of input

"""

epsilon = 1E-8

clipped\_raw\_scores = tf.maximum(raw\_scores, epsilon)

return tf.log(clipped\_raw\_scores)