# pylint: disable=no-member, attribute-defined-outside-init, too-many-instance-attributes

"""

Implementing HashingDiscretizer Layer

"""

import libtwml

import tensorflow.compat.v1 as tf

import twml

from twml.constants import HashingDiscretizerOptions

from twml.layers.layer import Layer

class HashingDiscretizer(Layer):

"""A layer that discretizes continuous features, with hashed feature assignments

HashingDiscretizer converts sparse continuous features into sparse

binary features. Each binary output feature indicates the presence of a

value in a HashingDiscretizer bin.

Each calibrated HashingDiscretizer input feature is converted to n\_bin+1 bins.

- n\_bin bin boundaries for each feature (i.e. len(bin\_vals[id])==n\_bin) defines n\_bin+1 bins

- bin assignment = sum(bin\_vals<val)

The difference between this layer and PercentileDiscretizer is that the

HashingDiscretizer always assigns the same output id in the

SparseTensor to the same input (feature id, bin) pair. This is useful if you

want to user transfer learning on pre-trained sparse to dense embedding

layers, but re-calibrate your discretizer on newer data.

If there are no calibrated features, then the discretizer will only apply

twml.util.limit\_bits to the the feature keys (aka "feature\_ids"). Essentially,

the discretizer will be a "no-operation", other than obeying `out\_bits`

Typically, a HashingDiscretizer layer will be generated by calling the

to\_layer() method of the HashingDiscretizerCalibrator

"""

def \_\_init\_\_(self, feature\_ids, bin\_vals, n\_bin, out\_bits,

cost\_per\_unit=500, options=None, \*\*kwargs):

"""

Creates a non-initialized `HashingDiscretizer` object.

Parent class args:

see [tf.layers.Layer](https://www.tensorflow.org/api\_docs/python/tf/layers/Layer)

for documentation of parent class arguments.

Required args:

feature\_ids (1D int64 numpy array):

- list of feature IDs that have been calibrated and have corresponding

bin boundary values in the bin\_vals array

- bin values for feature feature\_ids[i] live at bin\_vals[i\*n\_bin:(i+1)\*n\_bin]

bin\_vals (1D float numpy array):

- These are the bin boundary values for each calibrated feature

- len(bin\_vals) = n\_bin\*len(feature\_ids)

n\_bin (int):

- number of HashingDiscretizer bins is actually n\_bin + 1

- \*\*\*Note\*\*\* that if a value N is passed for the value of n\_bin to

HashingDiscretizerCalibrator, then HashingDiscretizerCalibrator

will generate N+1 bin boundaries for each feature, and hence there

will actually be N+2 potential bins for each feature

out\_bits (int):

Determines the maximum value for output feature IDs.

The dense\_shape of the SparseTensor returned by lookup(x)

will be [x.shape[0], 1 << output\_bits].

Optional args:

cost\_per\_unit (int):

- heuristic for intra op multithreading. approximate nanoseconds per input value.

options (int or None for default):

- Selects behavior of the op. Default is lower\_bound and integer\_multiplicative\_hashing.

- Use values in twml.constants.HashingDiscretizerOptions to select options as follows

choose exactly one of HashingDiscretizerOptions.{SEARCH\_LOWER\_BOUND, SEARCH\_LINEAR, SEARCH\_UPPER\_BOUND}

choose exactly one of HashingDiscretizerOptions.{HASH\_32BIT, HASH\_64BIT}

Bitwise OR these together to construct the options input.

For example, `options=(HashingDiscretizerOptions.SEARCH\_UPPER\_BOUND | HashingDiscretizerOptions.HASH\_64BIT)`

"""

super(HashingDiscretizer, self).\_\_init\_\_(\*\*kwargs)

self.\_feature\_ids = feature\_ids

self.\_bin\_vals = bin\_vals

self.\_n\_bin = n\_bin

self.\_out\_bits = out\_bits

self.cost\_per\_unit = cost\_per\_unit

if options is None:

options = HashingDiscretizerOptions.SEARCH\_LOWER\_BOUND | HashingDiscretizerOptions.HASH\_32BIT

self.\_options = options

if not self.built:

self.build(input\_shape=None)

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

"""

Creates the variables of the layer

"""

# make sure this is last

self.built = True

def call(self, inputs, \*\*kwargs):

"""

Implements HashingDiscretizer inference on a twml.SparseTensor.

Alternatively, accepts a tf.SparseTensor that can be converted

to twml.SparseTensor.

Performs discretization of input values.

i.e. bucket\_val = bucket(val | feature\_id)

This bucket mapping depends on the calibration (i.e. the bin boundaries).

However, (feature\_id, bucket\_val) pairs are mapped to new\_feature\_id in

a way that is independent of the calibration procedure

Args:

inputs: A 2D SparseTensor that is input to HashingDiscretizer for

discretization. It has a dense\_shape of [batch\_size, input\_size]

name: A name for the operation (optional).

Returns:

A tf.SparseTensor, created from twml.SparseTensor.to\_tf()

Its dense\_shape is [shape\_input.dense\_shape[0], 1 << output\_bits].

"""

if isinstance(inputs, tf.SparseTensor):

inputs = twml.SparseTensor.from\_tf(inputs)

assert(isinstance(inputs, twml.SparseTensor))

# sparse column indices

ids = inputs.ids

# sparse row indices

keys = inputs.indices

# sparse values

vals = inputs.values

if len(self.\_feature\_ids) > 0:

# pass all inputs to the c++ op

# the op determines whether to discretize (when a feature is calibrated),

# or whether to simply limit bits and pass through (when not calibrated)

# NOTE - Hashing is done in C++

discretizer\_keys, discretizer\_vals = libtwml.ops.hashing\_discretizer(

input\_ids=keys, # Input

input\_vals=vals, # Input

bin\_vals=self.\_bin\_vals, # Input

feature\_ids=tf.make\_tensor\_proto(self.\_feature\_ids), # Attr

n\_bin=self.\_n\_bin, # Attr

output\_bits=self.\_out\_bits, # Attr

cost\_per\_unit=self.cost\_per\_unit, # Attr

options=self.\_options, # Attr

)

else:

discretizer\_keys = twml.util.limit\_bits(keys, self.\_out\_bits)

discretizer\_vals = vals

batch\_size = tf.to\_int64(inputs.dense\_shape[0])

output\_size = tf.convert\_to\_tensor(1 << self.\_out\_bits, tf.int64)

output\_shape = [batch\_size, output\_size]

return twml.SparseTensor(ids, discretizer\_keys, discretizer\_vals, output\_shape).to\_tf()