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

"""

Implementing HashedPercentileDiscretizer Layer

"""

from twitter.deepbird.util.hashing import (

integer\_multiplicative\_hashing\_uniform,

integer\_multiplicative\_hashing,

) # noqa: F401

from libtwml import percentile\_discretizer\_bin\_indices

import numpy as np

import tensorflow.compat.v1 as tf

import twml

from twml.layers.layer import Layer

from twml.layers.partition import Partition

from twml.layers.stitch import Stitch

class HashedPercentileDiscretizer(Layer):

"""

HashedPercentileDiscretizer layer is constructed by PercentileDiscretizerCalibrator

after accumulating data

and performing minimum description length (PercentileDiscretizer) calibration.

HashedPercentileDiscretizer takes sparse continuous features and converts then to sparse

binary features. Each binary output feature is associated to an HashedPercentileDiscretizer

bin.

Each HashedPercentileDiscretizer input feature is converted to n\_bin bins.

Each HashedPercentileDiscretizer calibration tries to find bin delimiters such

that the number of features values

per bin is roughly equal (for each given HashedPercentileDiscretizer feature).

Note that if an input feature is rarely used, so will its associated output bin/features.

The difference between this layer and PercentileDiscretizer is that the

DeterministicPercentileDiscretize always assigns the same output id in the SparseTensor to the

same input feature id + bin. 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.

"""

def \_\_init\_\_(self, n\_feature, n\_bin, out\_bits,

bin\_values=None, hash\_keys=None, hash\_values=None,

bin\_ids=None, feature\_offsets=None,

hash\_fn=integer\_multiplicative\_hashing\_uniform, \*\*kwargs):

"""

Creates a non-initialized `HashedPercentileDiscretizer` object.

Before using the table you will have to initialize it. After initialization

the table will be immutable.

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:

n\_feature:

number of unique features accumulated during HashedPercentileDiscretizer calibration.

This is the number of features in the hash map.

Used to initialize bin\_values, hash\_keys, hash\_values,

bin\_ids, bin\_values and feature\_offsets.

n\_bin:

number of HashedPercentileDiscretizer bins used for

HashedPercentileDiscretizer calibration. Used to initialize bin\_values, hash\_keys,

hash\_values, bin\_ids, bin\_values and feature\_offsets.

out\_bits:

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:

hash\_keys:

contains the features ID that HashedPercentileDiscretizer discretizes and knows

about. The hash map (hash\_keys->hash\_values) is used for two reasons:

1. divide inputs into two feature spaces:

HashedPercentileDiscretizer vs non-HashedPercentileDiscretizer

2. transate the HashedPercentileDiscretizer features into a hash\_feature ID that

HashedPercentileDiscretizer understands.

The hash\_map is expected to contain n\_feature items.

hash\_values:

translates the feature IDs into hash\_feature IDs for HashedPercentileDiscretizer.

bin\_ids:

a 1D Tensor of size n\_feature \* n\_bin + 1 which contains

unique IDs to which the HashedPercentileDiscretizer features will be translated to.

For example, tf.Tensor(np.arange(n\_feature \* n\_bin)) would produce

the most efficient output space.

bin\_values:

a 1D Tensor aligned with bin\_ids.

For a given hash\_feature ID j, it's value bin's are indexed between

`j\*n\_bin` and `j\*n\_bin + n\_bin-1`.

As such, bin\_ids[j\*n\_bin+i] is translated from a hash\_feature ID of j

and a inputs value between

`bin\_values[j\*n\_bin + i]` and `bin\_values[j\*n\_bin+i+1]`.

feature\_offsets:

a 1D Tensor specifying the starting location of bins for a given feature id.

For example, tf.Tensor(np.arange(0, bin\_values.size, n\_bin, dtype='int64')).

hash\_fn:

a function that takes in `feature\_ids`, `bucket\_indices` and `output\_size` and

hashes the bucketed features into the `output\_size` buckets. The default uses knuth's

multiplicative hashing

"""

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

max\_discretizer\_feature = n\_feature \* (n\_bin + 1)

self.\_n\_feature = n\_feature

self.\_n\_bin = n\_bin

if not self.built:

self.build(input\_shape=None)

# build variables

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

self.\_out\_bits = out\_bits

hash\_keys = hash\_keys

if hash\_keys is None:

hash\_keys = np.empty(n\_feature, dtype=np.int64)

hash\_values = hash\_values

if hash\_values is None:

hash\_values = np.empty(n\_feature, dtype=np.int64)

initializer = tf.lookup.KeyValueTensorInitializer(hash\_keys, hash\_values)

self.hash\_map = tf.lookup.StaticHashTable(initializer, -1)

self.bin\_ids = bin\_ids

if bin\_ids is None:

bin\_ids = np.empty(max\_discretizer\_feature, dtype=np.int64)

self.bin\_values = bin\_values

if bin\_values is None:

bin\_values = np.empty(max\_discretizer\_feature, dtype=np.float32)

self.feature\_offsets = feature\_offsets

if feature\_offsets is None:

feature\_offsets = np.empty(n\_feature, dtype=np.int64)

self.hash\_fn = hash\_fn

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

"""

Creates the variables of the layer:

hash\_keys, hash\_values, bin\_ids, bin\_values, feature\_offsets and self.output\_size.

"""

# build layers

self.partition = Partition()

self.stitch = Stitch()

# make sure this is last

self.built = True

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

"""Looks up `keys` in a table, outputs the corresponding values.

Implements HashedPercentileDiscretizer inference where inputs are intersected with a

hash\_map.

Part of the inputs are discretized using twml.discretizer

to produce a discretizer\_output SparseTensor.

This SparseTensor is then joined with the original inputs SparseTensor,

but only for the inputs keys that did not get discretized.

Args:

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

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

name: A name for the operation (optional).

Returns:

A `SparseTensor` of the same type as `inputs`.

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

hashed\_keys = self.hash\_map.lookup(keys)

hashed\_keys = tf.cast(hashed\_keys, tf.int64)

found = tf.not\_equal(hashed\_keys, tf.constant(-1, tf.int64))

partition\_ids = tf.cast(found, tf.int32)

found = tf.reshape(found, [-1])

continuous\_feature\_ids = tf.boolean\_mask(keys, found)

vals, key, indices = self.partition(partition\_ids, vals, tf.where(found, hashed\_keys, keys))

non\_discretizer\_keys, discretizer\_in\_keys = key

non\_discretizer\_vals, discretizer\_in\_vals = vals

non\_discretizer\_keys = twml.util.limit\_bits(non\_discretizer\_keys, self.\_out\_bits)

self.non\_discretizer\_keys = non\_discretizer\_keys

# run HashedPercentileDiscretizer on the keys/values it knows about

output = percentile\_discretizer\_bin\_indices(discretizer\_in\_keys,

discretizer\_in\_vals,

self.bin\_ids,

self.bin\_values,

self.feature\_offsets)

discretizer\_bucket\_idxs, discretizer\_vals = output

new\_discretizer\_keys = self.hash\_fn(continuous\_feature\_ids, discretizer\_bucket\_idxs,

self.output\_size)

# Stitch the keys and values from discretizer and non discretizer indices back, with help

# of the Stitch Layer

self.discretizer\_out\_keys = new\_discretizer\_keys

concat\_data = self.stitch([non\_discretizer\_vals, discretizer\_vals],

[non\_discretizer\_keys, new\_discretizer\_keys],

indices)

concat\_vals, concat\_keys = concat\_data

# Generate output shape using \_compute\_output\_shape

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

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

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