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

"""

Implementing MDL Layer

"""

from .layer import Layer

from .partition import Partition

from .stitch import Stitch

import libtwml

import numpy as np

import tensorflow.compat.v1 as tf

import twml

class MDL(Layer): # noqa: T000

"""

MDL layer is constructed by MDLCalibrator after accumulating data

and performing minimum description length (MDL) calibration.

MDL takes sparse continuous features and converts then to sparse

binary features. Each binary output feature is associated to an MDL bin.

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

Each MDL calibration tries to find bin delimiters such that the number of features values

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

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

"""

def \_\_init\_\_(

self,

n\_feature, n\_bin, out\_bits,

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

bin\_ids=None, feature\_offsets=None, \*\*kwargs):

"""

Creates a non-initialized `MDL` 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 MDL 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 MDL bins used for MDL 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 MDL discretizes and knows about.

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

1. divide inputs into two feature spaces: MDL vs non-MDL

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

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

hash\_values:

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

bin\_ids:

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

unique IDs to which the MDL 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')).

"""

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

tf.logging.warning("MDL will be deprecated. Please use PercentileDiscretizer instead")

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

self.\_n\_feature = n\_feature

self.\_n\_bin = n\_bin

self.\_hash\_keys\_initializer = tf.constant\_initializer(

hash\_keys if hash\_keys is not None

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

dtype=np.int64

)

self.\_hash\_values\_initializer = tf.constant\_initializer(

hash\_values if hash\_values is not None

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

dtype=np.int64

)

self.\_bin\_ids\_initializer = tf.constant\_initializer(

bin\_ids if bin\_ids is not None

else np.empty(max\_mdl\_feature, dtype=np.int64),

dtype=np.int64

)

self.\_bin\_values\_initializer = tf.constant\_initializer(

bin\_values if bin\_values is not None

else np.empty(max\_mdl\_feature, dtype=np.float32),

dtype=np.float32

)

self.\_feature\_offsets\_initializer = tf.constant\_initializer(

feature\_offsets if feature\_offsets is not None

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

dtype=np.int64

)

# note that calling build here is an exception as typically \_\_call\_\_ would call build().

# We call it here because we need to initialize hash\_map.

# Also note that the variable\_scope is set by add\_variable in build()

if not self.built:

self.build(input\_shape=None)

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

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()

# build variables

hash\_keys = self.add\_variable(

'hash\_keys',

initializer=self.\_hash\_keys\_initializer,

shape=[self.\_n\_feature],

dtype=tf.int64,

trainable=False)

hash\_values = self.add\_variable(

'hash\_values',

initializer=self.\_hash\_values\_initializer,

shape=[self.\_n\_feature],

dtype=tf.int64,

trainable=False)

# hashmap converts known features into range [0, n\_feature)

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

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

self.bin\_ids = self.add\_variable(

'bin\_ids',

initializer=self.\_bin\_ids\_initializer,

shape=[self.\_n\_feature \* (self.\_n\_bin + 1)],

dtype=tf.int64,

trainable=False)

self.bin\_values = self.add\_variable(

'bin\_values',

initializer=self.\_bin\_values\_initializer,

shape=[self.\_n\_feature \* (self.\_n\_bin + 1)],

dtype=tf.float32,

trainable=False)

self.feature\_offsets = self.add\_variable(

'feature\_offsets',

initializer=self.\_feature\_offsets\_initializer,

shape=[self.\_n\_feature],

dtype=tf.int64,

trainable=False)

# make sure this is last

self.built = True

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

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

Implements MDL inference where inputs are intersected with a hash\_map.

Part of the inputs are discretized using twml.mdl to produce a mdl\_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 MDL 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

# get intersect(keys, hash\_map)

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

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

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

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

non\_mdl\_keys, mdl\_in\_keys = key

non\_mdl\_vals, mdl\_in\_vals = vals

self.non\_mdl\_keys = non\_mdl\_keys

# run MDL on the keys/values it knows about

mdl\_keys, mdl\_vals = libtwml.ops.mdl(mdl\_in\_keys, mdl\_in\_vals, self.bin\_ids, self.bin\_values,

self.feature\_offsets)

# handle output ID conflicts

mdl\_size = tf.size(self.bin\_ids, out\_type=tf.int64)

non\_mdl\_size = tf.subtract(self.output\_size, mdl\_size)

non\_mdl\_keys = tf.add(tf.floormod(non\_mdl\_keys, non\_mdl\_size), mdl\_size)

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

# of the Stitch Layer

# out for inference checking

self.mdl\_out\_keys = mdl\_keys

concat\_data = self.stitch([non\_mdl\_vals, mdl\_vals],

[non\_mdl\_keys, mdl\_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()

def compute\_output\_shape(self, input\_shape):

"""Computes the output shape of the layer given the input shape.

Args:

input\_shape: A (possibly nested tuple of) `TensorShape`. It need not

be fully defined (e.g. the batch size may be unknown).

Raises NotImplementedError.

"""

raise NotImplementedError