#include "tensorflow/core/framework/op.h"

#include "tensorflow/core/framework/shape\_inference.h"

#include "tensorflow/core/framework/op\_kernel.h"

#include "tensorflow/core/util/work\_sharder.h"

#include <twml.h>

#include "tensorflow\_utils.h"

using namespace tensorflow;

void ComputeHashingDiscretizer(

OpKernelContext\*,

int64\_t,

const twml::Map<int64\_t, int64\_t> &,

int64\_t,

int64\_t,

int64\_t);

REGISTER\_OP("HashingDiscretizer")

.Attr("T: {float, double}")

.Input("input\_ids: int64")

.Input("input\_vals: T")

.Input("bin\_vals: T")

.Attr("feature\_ids: tensor = { dtype: DT\_INT64 }")

.Attr("n\_bin: int")

.Attr("output\_bits: int")

.Attr("cost\_per\_unit: int")

.Attr("options: int")

.Output("new\_keys: int64")

.Output("new\_vals: T")

.SetShapeFn(

[](::tensorflow::shape\_inference::InferenceContext\* c) {

c->set\_output(0, c->input(0));

c->set\_output(1, c->input(1));

return Status::OK();

}

)

.Doc(R"doc(

This operation discretizes a tensor containing continuous features (if calibrated).

- note - choice of float or double should be consistent among inputs/output

Input

input\_ids(int64): A tensor containing input feature ids (direct from data record).

input\_vals(float/double): A tensor containing input values at corresponding feature ids.

- i.e. input\_ids[i] <-> input\_vals[i] for each i

bin\_vals(float/double): A tensor containing the bin boundaries for values of a given feature.

- float or double, matching input\_vals

feature\_ids(int64 attr): 1D TensorProto of feature IDs seen during calibration

-> hint: look up make\_tensor\_proto:

proto\_init = np.array(values, dtype=np.int64)

tensor\_attr = tf.make\_tensor\_proto(proto\_init)

n\_bin(int): The number of bin boundary values per feature

-> hence, n\_bin + 1 buckets for each feature

output\_bits(int): The maximum number of bits to use for the output IDs.

cost\_per\_unit(int): An estimate of the number of CPU cycles (or nanoseconds

if not CPU-bound) to complete a unit of work. Overestimating creates too

many shards and CPU time will be dominated by per-shard overhead, such as

Context creation. Underestimating may not fully make use of the specified

parallelism.

options(int): selects behavior of the op.

0x00 in bits{1:0} for std::lower\_bound bucket search.

0x01 in bits{1:0} for linear bucket search

0x02 in bits{1:0} for std::upper\_bound bucket search

0x00 in bits{4:2} for integer\_multiplicative\_hashing

0x01 in bits{4:2} for integer64\_multiplicative\_hashing

higher bits/other values are reserved for future extensions

Outputs

new\_keys(int64): The discretized feature ids with same shape and size as keys.

new\_vals(float or double): The discretized values with the same shape and size as vals.

Operation

Note that the discretization operation maps observation vectors to higher dimensional

observation vectors. Here, we describe this mapping.

Let a calibrated feature observation be given by (F,x), where F is the ID of the

feature, and x is some real value (i.e., continuous feature). This kind of

representation is useful for the representation of sparse vectors, where there

are many zeros.

For example, for a dense feature vector [1.2, 2.4, 3.6], we might have

(0, 1.2) (1, 2.4) and (2, 3.6), with feature IDs indicating the 0th, 1st, and 2nd

elements of the vector.

The disretizer performs the following operation:

(F,x) -> (map(x|F),1).

Hence, we have that map(x|F) is a new feature ID, and the value observed for that

feature is 1. We might read map(x|F) as 'the map of x for feature F'.

For each feature F, we associate a (discrete, finite) set of new feature IDs, newIDs(F).

We will then have that map(x|F) is in the set newIDs(F) for any value of x. Each

set member of newIDs(F) is associated with a 'bin', as defined by the bin

boundaries given in the bin\_vals input array. For any two different feature IDs F

and G, we would ideally have that INTERSECT(newIDs(F),newIDs(G)) is the empty set.

However, this is not guaranteed for this discretizer.

In the case of this hashing discretizer, map(x|F) can actually be written as follows:

let bucket = bucket(x|F) be the the bucket index for x, according to the

calibration on F. (This is an integer value in [0,n\_bin], inclusive)

F is an integer ID. Here, we have that map(x|F) = hash\_fn(F,bucket). This has

the desirable property that the new ID depends only on the calibration data

supplied for feature F, and not on any other features in the dataset (e.g.,

number of other features present in the calibration data, or order of features

in the dataset). Note that PercentileDiscretizer does NOT have this property.

This comes at the expense of the possibility of output ID collisions, which

we try to minimize through the design of hash\_fn.

Example - consider input vector with a single element, i.e. [x].

Let's Discretize to one of 2 values, as follows:

Let F=0 for the ID of the single feature in the vector.

Let the bin boundary of feature F=0 be BNDRY(F) = BNDRY(0) since F=0

bucket = bucket(x|F=0) = 0 if x<=BNDRY(0) else 1

Let map(x|F) = hash\_fn(F=0,bucket=0) if x<=BNDRY(0) else hash\_fn(F=0,bucket=1)

If we had another element y in the vector, i.e. [x, y], then we might additionally

Let F=1 for element y.

Let the bin boundary be BNDRY(F) = BNDRY(1) since F=1

bucket = bucket(x|F=1) = 0 if x<=BNDRY(1) else 1

Let map(x|F) = hash\_fn(F=1,bucket=0) if x<=BNDRY(1) else hash\_fn(F=1,bucket=1)

Note how the construction of map(x|F=1) does not depend on whether map(x|F=0)

was constructed.

)doc");

template<typename T>

class HashingDiscretizer : public OpKernel {

public:

explicit HashingDiscretizer(OpKernelConstruction\* context) : OpKernel(context) {

OP\_REQUIRES\_OK(context,

context->GetAttr("n\_bin", &n\_bin\_));

OP\_REQUIRES(context,

n\_bin\_ > 0,

errors::InvalidArgument("Must have n\_bin\_ > 0."));

OP\_REQUIRES\_OK(context,

context->GetAttr("output\_bits", &output\_bits\_));

OP\_REQUIRES(context,

output\_bits\_ > 0,

errors::InvalidArgument("Must have output\_bits\_ > 0."));

OP\_REQUIRES\_OK(context,

context->GetAttr("cost\_per\_unit", &cost\_per\_unit\_));

OP\_REQUIRES(context,

cost\_per\_unit\_ >= 0,

errors::InvalidArgument("Must have cost\_per\_unit >= 0."));

OP\_REQUIRES\_OK(context,

context->GetAttr("options", &options\_));

// construct the ID\_to\_index hash map

Tensor feature\_IDs;

// extract the tensors

OP\_REQUIRES\_OK(context,

context->GetAttr("feature\_ids", &feature\_IDs));

// for access to the data

// int64\_t data type is set in to\_layer function of the calibrator objects in Python

auto feature\_IDs\_flat = feature\_IDs.flat<int64>();

// verify proper dimension constraints

OP\_REQUIRES(context,

feature\_IDs.shape().dims() == 1,

errors::InvalidArgument("feature\_ids must be 1D."));

// reserve space in the hash map and fill in the values

int64\_t num\_features = feature\_IDs.shape().dim\_size(0);

#ifdef USE\_DENSE\_HASH

ID\_to\_index\_.set\_empty\_key(0);

ID\_to\_index\_.resize(num\_features);

#else

ID\_to\_index\_.reserve(num\_features);

#endif // USE\_DENSE\_HASH

for (int64\_t i = 0 ; i < num\_features ; i++) {

ID\_to\_index\_[feature\_IDs\_flat(i)] = i;

}

}

void Compute(OpKernelContext\* context) override {

ComputeHashingDiscretizer(

context,

output\_bits\_,

ID\_to\_index\_,

n\_bin\_,

cost\_per\_unit\_,

options\_);

}

private:

twml::Map<int64\_t, int64\_t> ID\_to\_index\_;

int n\_bin\_;

int output\_bits\_;

int cost\_per\_unit\_;

int options\_;

};

#define REGISTER(Type) \

REGISTER\_KERNEL\_BUILDER( \

Name("HashingDiscretizer") \

.Device(DEVICE\_CPU) \

.TypeConstraint<Type>("T"), \

HashingDiscretizer<Type>); \

REGISTER(float);

REGISTER(double);

void ComputeHashingDiscretizer(

OpKernelContext\* context,

int64\_t output\_bits,

const twml::Map<int64\_t, int64\_t> &ID\_to\_index,

int64\_t n\_bin,

int64\_t cost\_per\_unit,

int64\_t options) {

const Tensor& keys = context->input(0);

const Tensor& vals = context->input(1);

const Tensor& bin\_vals = context->input(2);

const int64 output\_size = keys.dim\_size(0);

TensorShape output\_shape;

OP\_REQUIRES\_OK(context, TensorShapeUtils::MakeShape(&output\_size, 1, &output\_shape));

Tensor\* new\_keys = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, output\_shape, &new\_keys));

Tensor\* new\_vals = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(1, output\_shape, &new\_vals));

try {

twml::Tensor out\_keys\_ = TFTensor\_to\_twml\_tensor(\*new\_keys);

twml::Tensor out\_vals\_ = TFTensor\_to\_twml\_tensor(\*new\_vals);

const twml::Tensor in\_keys\_ = TFTensor\_to\_twml\_tensor(keys);

const twml::Tensor in\_vals\_ = TFTensor\_to\_twml\_tensor(vals);

const twml::Tensor bin\_vals\_ = TFTensor\_to\_twml\_tensor(bin\_vals);

// retrieve the thread pool from the op context

auto worker\_threads = \*(context->device()->tensorflow\_cpu\_worker\_threads());

// Definition of the computation thread

auto task = [&](int64 start, int64 limit) {

twml::hashDiscretizerInfer(out\_keys\_, out\_vals\_,

in\_keys\_, in\_vals\_,

n\_bin,

bin\_vals\_,

output\_bits,

ID\_to\_index,

start, limit,

options);

};

// let Tensorflow split up the work as it sees fit

Shard(worker\_threads.num\_threads,

worker\_threads.workers,

output\_size,

static\_cast<int64>(cost\_per\_unit),

task);

} catch (const std::exception &e) {

context->CtxFailureWithWarning(errors::InvalidArgument(e.what()));

}

}