#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 CombinedComputeDiscretizers(

OpKernelContext\*,

int64\_t,

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

int64\_t);

REGISTER\_OP("PercentileDiscretizerV2")

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

.Input("input\_ids: int64")

.Input("input\_vals: T")

.Input("bin\_ids: int64")

.Input("bin\_vals: T")

.Input("feature\_offsets: int64")

.Input("start\_compute: int64")

.Input("end\_compute: int64")

.Attr("output\_bits: int")

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

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

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

.Output("new\_keys: int64")

.Output("new\_vals: T")

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

// TODO: check sizes

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

c->set\_output(1, c->input(0));

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: A tensor containing input values at corresponding feature ids.

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

- float or double

bin\_ids(int64): A tensor containing the discretized feature id for each bin.

bin\_vals: A tensor containing the bin boundaries for values of a given feature.

- float or double

feature\_offsets(int64): Specifies the starting location of bins for a given feature id.

start\_compute(int64 scalar tensor): which index to start the computation at

end\_compute(int64 scalar tensor): which index to end the computation right before

-> for example, (start\_compute,end\_compute)=(0,10) would compute on 0 thru 9

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

-> 2\*\*out\_bits must be greater than bin\_ids.size

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

feature\_indices(int64): 1D TensorProto of feature indices corresponding with feature\_IDs

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

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

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

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.

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 F~(x) 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 have that

INTERSECT(newIDs(F),newIDs(G)) is the empty set

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

Let newIDs(F) = newIDs(0) = {0,1}

Let map(x|F) = map(x|0) = 0 if x<=BNDRY else 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

Let newIDs(F) = newIDs(1) = {2,3} (so as to have empty intersect with newIDs(0))

Let map(x|F) = map(x|1) = 2 if x<=BNDRY else 3

Consider vector observation [-0.1, 0.2]. We then represent this as [(0, -0.1), (1, 0.2)]

Let BNDRY(0) = BNDRY(1) = 0. When we discretize the vector, we get:

(0, -0.1) -> (map(-0.1|0), 1) = (0, 1)

(1, 0.2) -> (map( 0.2|1), 1) = (3, 1)

Our output vector is then represented sparsely as [(0, 1), (3, 1)], and the dense

representation of this could be [1, 0, 0, 1]

)doc");

template<typename T>

class PercentileDiscretizerV2 : public OpKernel {

public:

explicit PercentileDiscretizerV2(OpKernelConstruction\* context) : OpKernel(context) {

// get the number of output bits

// for use with features that have not been calibrated

OP\_REQUIRES\_OK(context,

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

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."));

// construct the ID\_to\_index hash map

Tensor feature\_IDs;

Tensor feature\_indices;

// extract the tensors

OP\_REQUIRES\_OK(context,

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

OP\_REQUIRES\_OK(context,

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

// 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>();

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

// verify proper dimension constraints

OP\_REQUIRES(context, feature\_IDs.shape() == feature\_indices.shape(),

errors::InvalidArgument("feature\_ids and feature\_indices must be identical shape."));

OP\_REQUIRES(context, feature\_IDs.shape().dims() == 1,

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

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

int 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 (int i = 0 ; i < num\_features ; i++) {

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

}

}

void Compute(OpKernelContext\* context) override {

CombinedComputeDiscretizers(

context,

output\_bits\_,

ID\_to\_index\_,

cost\_per\_unit\_);

}

private:

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

int output\_bits\_;

int cost\_per\_unit\_;

};

#define REGISTER(Type) \

REGISTER\_KERNEL\_BUILDER( \

Name("PercentileDiscretizerV2") \

.Device(DEVICE\_CPU) \

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

PercentileDiscretizerV2<Type>); \

REGISTER(float);

REGISTER(double);

void CombinedComputeDiscretizers(

OpKernelContext\* context,

int64\_t output\_bits,

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

int64\_t cost\_per\_unit) {

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

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

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

const Tensor& bin\_vals = context->input(3);

const Tensor& feature\_offsets = context->input(4);

uint64 full\_size = keys.dim\_size(0);

const int total\_size = static\_cast<int64>(full\_size);

TensorShape output\_shape = {total\_size};

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\_ids\_ = TFTensor\_to\_twml\_tensor(bin\_ids);

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

const twml::Tensor feature\_offsets\_ = TFTensor\_to\_twml\_tensor(feature\_offsets);

// 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::discretizerInfer(out\_keys\_, out\_vals\_,

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

bin\_ids\_, bin\_vals\_,

feature\_offsets\_, output\_bits,

ID\_to\_index,

start, limit,

start);

};

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

Shard(worker\_threads.num\_threads,

worker\_threads.workers,

full\_size,

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

task);

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

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

}

}