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

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

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

#include <twml.h>

#include "tensorflow\_utils.h"

using namespace tensorflow;

void ComputeDiscretizers(OpKernelContext\* context, const bool return\_bin\_indices = false) {

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);

Tensor\* new\_keys = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, keys.shape(),

&new\_keys));

Tensor\* new\_vals = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(1, keys.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);

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

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

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

feature\_offsets\_,

return\_bin\_indices);

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

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

}

}

REGISTER\_OP("MDL")

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

.Input("keys: int64")

.Input("vals: T")

.Input("bin\_ids: int64")

.Input("bin\_vals: T")

.Input("feature\_offsets: int64")

.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.

Input

keys: A tensor containing feature ids.

vals: A tensor containing values at corresponding feature ids.

bin\_ids: A tensor containing the discretized feature id for a given bin.

bin\_vals: A tensor containing the bin boundaries for value at a given feature id.

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

Expected Sizes:

keys, vals: [N].

bin\_ids, bin\_vals: [sum\_{n=1}^{n=num\_classes} num\_bins(n)]

where

- N is the number of sparse features in the current batch.

- [0, num\_classes) represents the range each feature id can take.

- num\_bins(n) is the number of bins for a given feature id.

- If num\_bins is fixed, then xs, ys are of size [num\_classes \* num\_bins].

Expected Types:

keys, bin\_ids: int64.

vals: float or double.

bin\_vals: same as vals.

Before using MDL, you should use a hashmap to get the intersection of

input `keys` with the features that MDL knows about:

::

keys, vals # keys can be in range [0, 1 << 63)

mdl\_keys = hashmap.find(keys) # mdl\_keys are now in range [0, num\_classes\_from\_calibration)

mdl\_keys = where (mdl\_keys != -1) # Ignore keys not found

Inside MDL, the following is happening:

::

start = offsets[key[i]]

end = offsets[key[i] + 1]

idx = binary\_search for val[i] in [bin\_vals[start], bin\_vals[end]]

result\_keys[i] = bin\_ids[idx]

val[i] = 1 # binary feature value

Outputs

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

new\_vals: The discretized values with the same shape and size as vals.

)doc");

template<typename T>

class MDL : public OpKernel {

public:

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

}

void Compute(OpKernelContext\* context) override {

ComputeDiscretizers(context);

}

};

REGISTER\_OP("PercentileDiscretizer")

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

.Input("keys: int64")

.Input("vals: T")

.Input("bin\_ids: int64")

.Input("bin\_vals: T")

.Input("feature\_offsets: int64")

.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.

Input

keys: A tensor containing feature ids.

vals: A tensor containing values at corresponding feature ids.

bin\_ids: A tensor containing the discretized feature id for a given bin.

bin\_vals: A tensor containing the bin boundaries for value at a given feature id.

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

Expected Sizes:

keys, vals: [N].

bin\_ids, bin\_vals: [sum\_{n=1}^{n=num\_classes} num\_bins(n)]

where

- N is the number of sparse features in the current batch.

- [0, num\_classes) represents the range each feature id can take.

- num\_bins(n) is the number of bins for a given feature id.

- If num\_bins is fixed, then xs, ys are of size [num\_classes \* num\_bins].

Expected Types:

keys, bin\_ids: int64.

vals: float or double.

bin\_vals: same as vals.

Before using PercentileDiscretizer, you should use a hashmap to get the intersection of

input `keys` with the features that PercentileDiscretizer knows about:

::

keys, vals # keys can be in range [0, 1 << 63)

percentile\_discretizer\_keys = hashmap.find(keys) # percentile\_discretizer\_keys are now in range [0, num\_classes\_from\_calibration)

percentile\_discretizer\_keys = where (percentile\_discretizer\_keys != -1) # Ignore keys not found

Inside PercentileDiscretizer, the following is happening:

::

start = offsets[key[i]]

end = offsets[key[i] + 1]

idx = binary\_search for val[i] in [bin\_vals[start], bin\_vals[end]]

result\_keys[i] = bin\_ids[idx]

val[i] = 1 # binary feature value

Outputs

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

new\_vals: The discretized values with the same shape and size as vals.

)doc");

template<typename T>

class PercentileDiscretizer : public OpKernel {

public:

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

}

void Compute(OpKernelContext\* context) override {

ComputeDiscretizers(context);

}

};

REGISTER\_OP("PercentileDiscretizerBinIndices")

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

.Input("keys: int64")

.Input("vals: T")

.Input("bin\_ids: int64")

.Input("bin\_vals: T")

.Input("feature\_offsets: int64")

.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 the feature id and bin id of the discretized value is the same on multiple runs, they

will always be assigned to the same output key and value, regardless of the bin\_id assigned during

calibration.

Input

keys: A tensor containing feature ids.

vals: A tensor containing values at corresponding feature ids.

bin\_ids: A tensor containing the discretized feature id for a given bin.

bin\_vals: A tensor containing the bin boundaries for value at a given feature id.

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

Expected Sizes:

keys, vals: [N].

bin\_ids, bin\_vals: [sum\_{n=1}^{n=num\_classes} num\_bins(n)]

where

- N is the number of sparse features in the current batch.

- [0, num\_classes) represents the range each feature id can take.

- num\_bins(n) is the number of bins for a given feature id.

- If num\_bins is fixed, then xs, ys are of size [num\_classes \* num\_bins].

Expected Types:

keys, bin\_ids: int64.

vals: float or double.

bin\_vals: same as vals.

Before using PercentileDiscretizerBinIndices, you should use a hashmap to get the intersection of

input `keys` with the features that PercentileDiscretizerBinIndices knows about:

::

keys, vals # keys can be in range [0, 1 << 63)

percentile\_discretizer\_keys = hashmap.find(keys) # percentile\_discretizer\_keys are now in range [0, num\_classes\_from\_calibration)

percentile\_discretizer\_keys = where (percentile\_discretizer\_keys != -1) # Ignore keys not found

Inside PercentileDiscretizerBinIndices, the following is happening:

::

start = offsets[key[i]]

end = offsets[key[i] + 1]

idx = binary\_search for val[i] in [bin\_vals[start], bin\_vals[end]]

result\_keys[i] = bin\_ids[idx]

val[i] = 1 # binary feature value

Outputs

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

new\_vals: The discretized values with the same shape and size as vals.

)doc");

template<typename T>

class PercentileDiscretizerBinIndices : public OpKernel {

public:

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

}

void Compute(OpKernelContext\* context) override {

ComputeDiscretizers(context, true);

}

};

#define REGISTER(Type) \

\

REGISTER\_KERNEL\_BUILDER( \

Name("PercentileDiscretizerBinIndices") \

.Device(DEVICE\_CPU) \

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

PercentileDiscretizerBinIndices<Type>); \

\

REGISTER\_KERNEL\_BUILDER( \

Name("PercentileDiscretizer") \

.Device(DEVICE\_CPU) \

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

PercentileDiscretizer<Type>); \

\

REGISTER\_KERNEL\_BUILDER( \

Name("MDL") \

.Device(DEVICE\_CPU) \

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

MDL<Type>); \

REGISTER(float);

REGISTER(double);