#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"

#include <map>

#include <vector>

REGISTER\_OP("FeatureExtractor")

.Attr("T: {float, double} = DT\_FLOAT")

.Input("mask\_in: bool")

.Input("ids\_in: int64")

.Input("keys\_in: int64")

.Input("values\_in: T")

.Input("codes\_in: int64")

.Input("types\_in: int8")

.Output("ids\_out: int64")

.Output("keys\_out: int64")

.Output("values\_out: T")

.Output("codes\_out: int64")

.Output("types\_out: int8")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that extracts the desired indices of a Tensor based on a mask

Input

mask\_in: boolean Tensor that determines which are the indices to be kept (bool)

ids\_in: input indices Tensor (int64)

keys\_in: input keys Tensor (int64)

values\_in: input values Tensor (float/double)

codes\_in: input codes Tensor (int64)

types\_in: input types Tensor(int8)

Outputs

ids\_out: output indices Tensor (int64)

keys\_out: output keys Tensor (int64)

values\_out: output values Tensor (float/double)

codes\_out: output codes Tensor (int64)

types\_out: output types Tensor(int8)

)doc");

template <typename T>

class FeatureExtractor : public OpKernel {

public:

explicit FeatureExtractor(OpKernelConstruction\* context)

: OpKernel(context) {}

template <typename A, typename U>

bool allequal(const A &t, const U &u) {

return t == u;

}

template <typename A, typename U, typename... Others>

bool allequal(const A &t, const U &u, Others const &... args) {

return (t == u) && allequal(u, args...);

}

void Compute(OpKernelContext\* context) override {

// Get input tensors

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

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

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

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

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

const Tensor& input\_types = context->input(5);

auto mask = input\_mask.flat<bool>();

auto ids = input\_ids.flat<int64>();

auto keys = input\_keys.flat<int64>();

auto codes = input\_codes.flat<int64>();

auto values = input\_values.flat<T>();

auto types = input\_types.flat<int8>();

// Verify that all Tensors have the same size.

OP\_REQUIRES(context, allequal(mask.size(), ids.size(), keys.size(), codes.size(), values.size(), types.size()),

errors::InvalidArgument("all input vectors must be the same size."));

// Get the size of the output vectors by counting the numbers of trues.

int total\_size = 0;

for (int i = 0; i < mask.size(); i++) {

if (mask(i))

total\_size += 1;

}

// Shape is the number of Trues in the mask Eigen::Tensor

TensorShape shape\_out = {total\_size};

// Create the output tensors

Tensor\* output\_codes = nullptr;

Tensor\* output\_ids = nullptr;

Tensor\* output\_values = nullptr;

Tensor\* output\_types = nullptr;

Tensor\* output\_keys = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, shape\_out, &output\_ids));

OP\_REQUIRES\_OK(context, context->allocate\_output(1, shape\_out, &output\_keys));

OP\_REQUIRES\_OK(context, context->allocate\_output(2, shape\_out, &output\_values));

OP\_REQUIRES\_OK(context, context->allocate\_output(3, shape\_out, &output\_codes));

OP\_REQUIRES\_OK(context, context->allocate\_output(4, shape\_out, &output\_types));

auto output\_ids\_ = output\_ids->flat<int64>();

auto output\_keys\_ = output\_keys->flat<int64>();

auto output\_codes\_ = output\_codes->flat<int64>();

auto output\_values\_ = output\_values->flat<T>();

auto output\_types\_ = output\_types->flat<int8>();

// Iterate through the mask and set values to output Eigen::Tensors

int j = 0;

for (int i = 0; i < mask.size(); i++) {

if (mask(i)) {

output\_ids\_(j) = ids(i);

output\_keys\_(j) = keys(i);

output\_values\_(j) = values(i);

output\_codes\_(j) = codes(i);

output\_types\_(j) = types(i);

++j;

}

}

}

};

#define REGISTER(Type) \

\

REGISTER\_KERNEL\_BUILDER( \

Name("FeatureExtractor") \

.Device(DEVICE\_CPU) \

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

FeatureExtractor<Type>); \

REGISTER(float);

REGISTER(double);