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

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

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

#include <twml.h>

#include <twml/functions.h>

#include <twml/utilities.h>

#include "tensorflow\_utils.h"

#include "resource\_utils.h"

#include <algorithm>

using std::string;

REGISTER\_OP("DecodeDataRecord")

.Attr("InputType: {uint8, string}")

.Attr("keep\_features: list(int)")

.Attr("keep\_codes: list(int)")

.Attr("label\_features: list(int)")

.Attr("weight\_features: list(int) = []")

.Input("input\_bytes: InputType")

.Output("data\_record\_handle: resource")

.SetShapeFn(shape\_inference::ScalarShape)

.Doc(R"doc(

A tensorflow OP that creates a handle for the datarecord.

Attr

keep\_features: a list of int ids to keep.

keep\_codes: their corresponding code.

label\_features: list of feature ids representing the labels.

weight\_features: list of feature ids representing the weights. Defaults to empty list.

shared\_name: name used by the resource handle inside the resource manager.

container: name used by the container of the resources.

shared\_name and container are required when inheriting from ResourceOpKernel.

Input

input\_bytes: Input tensor containing the serialized batch of HashedDataRecords.

Outputs

data\_record\_handle: A resource handle to the DataRecord struct.

)doc");

template<typename InputType>

class DecodeDataRecord : public OpKernel {

public:

explicit DecodeDataRecord(OpKernelConstruction\* context)

: OpKernel(context) {

std::vector<int64> keep\_features;

std::vector<int64> keep\_codes;

std::vector<int64> label\_features;

std::vector<int64> weight\_features;

OP\_REQUIRES\_OK(context, context->GetAttr("keep\_features", &keep\_features));

OP\_REQUIRES\_OK(context, context->GetAttr("keep\_codes", &keep\_codes));

OP\_REQUIRES\_OK(context, context->GetAttr("label\_features", &label\_features));

OP\_REQUIRES\_OK(context, context->GetAttr("weight\_features", &weight\_features));

OP\_REQUIRES(context, keep\_features.size() == keep\_codes.size(),

errors::InvalidArgument("keep keys and values must have same size."));

#ifdef USE\_DENSE\_HASH

m\_keep\_map.set\_empty\_key(0);

m\_labels\_map.set\_empty\_key(0);

m\_weights\_map.set\_empty\_key(0);

#endif // USE\_DENSE\_HASH

for (uint64\_t i = 0; i < keep\_features.size(); i++) {

m\_keep\_map[keep\_features[i]] = keep\_codes[i];

}

for (uint64\_t i = 0; i < label\_features.size(); i++) {

m\_labels\_map[label\_features[i]] = i;

}

for (uint64\_t i = 0; i < weight\_features.size(); i++) {

m\_weights\_map[weight\_features[i]] = i;

}

}

private:

twml::Map<int64\_t, int64\_t> m\_keep\_map;

twml::Map<int64\_t, int64\_t> m\_labels\_map;

twml::Map<int64\_t, int64\_t> m\_weights\_map;

void Compute(OpKernelContext\* context) override {

try {

DataRecordResource \*resource = nullptr;

OP\_REQUIRES\_OK(context, makeResourceHandle<DataRecordResource>(context, 0, &resource));

// Store the input bytes in the resource so it isnt freed before the resource.

// This is necessary because we are not copying the contents for tensors.

resource->input = context->input(0);

int batch\_size = getBatchSize<InputType>(resource->input);

int num\_labels = static\_cast<int>(m\_labels\_map.size());

int num\_weights = static\_cast<int>(m\_weights\_map.size());

twml::DataRecordReader reader;

reader.setKeepMap(&m\_keep\_map);

reader.setLabelsMap(&m\_labels\_map);

// Do not set weight map if it is empty. This will take a faster path.

if (num\_weights != 0) {

reader.setWeightsMap(&m\_weights\_map);

}

resource->records.clear();

resource->records.reserve(batch\_size);

for (int i = 0; i < batch\_size; i++) {

resource->records.emplace\_back(num\_labels, num\_weights);

}

for (int64 id = 0; id < batch\_size; id++) {

const uint8\_t \*input\_bytes = getInputBytes<InputType>(resource->input, id);

reader.setBuffer(input\_bytes);

// decode the reader

resource->records[id].decode(reader);

}

// This should be fine because m\_keep\_map should never go out of scope.

resource->keep\_map = &m\_keep\_map;

resource->num\_weights = num\_weights;

resource->num\_labels = num\_labels;

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

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

}

}

};

int64\_t count\_if\_exists(const twml::DataRecord::BinaryFeatures &set,

const twml::Map<int64\_t, int64\_t> \*const keep\_map) {

int64\_t count = 0;

for (const auto &key : set) {

if (keep\_map->find(key) == keep\_map->end()) continue;

count++;

}

return count;

}

// This works for continuous, discrete, and string features

template<typename V>

int64\_t count\_if\_exists(const twml::Map<int64\_t, V> &map,

const twml::Map<int64\_t, int64\_t> \*const keep\_map) {

int64\_t count = 0;

for (const auto &elem : map) {

if (keep\_map->find(elem.first) == keep\_map->end()) continue;

count++;

}

return count;

}

int64\_t count\_if\_exists(const twml::DataRecord::SparseBinaryFeatures &map,

const twml::Map<int64\_t, int64\_t> \*const keep\_map) {

int64\_t count = 0;

for (const auto &elem : map) {

if (keep\_map->find(elem.first) == keep\_map->end()) continue;

count += elem.second.size();

}

return count;

}

int64\_t count\_if\_exists(const twml::DataRecord::SparseContinuousFeatures &map,

const twml::Map<int64\_t, int64\_t> \*const keep\_map) {

int64\_t count = 0;

for (const auto &elem : map) {

if (keep\_map->find(elem.first) == keep\_map->end()) continue;

count += elem.second.size();

}

return count;

}

REGISTER\_OP("GetBinaryFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads binary features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

values: always set to 1 (float)

)doc");

class GetBinaryFeatures : public OpKernel {

public:

explicit GetBinaryFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_binary\_size = count\_if\_exists(common.getBinary(), handle->keep\_map);

int64 total\_binary\_size = records.size() \* common\_binary\_size;

for (int id = 0; id < records.size(); id++) {

total\_binary\_size += count\_if\_exists(handle->records[id].getBinary(), handle->keep\_map);

}

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

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* ids = nullptr;

Tensor\* values = nullptr;

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto ids\_flat = ids->flat<int64>();

auto values\_flat = values->flat<float>();

for (int64 id = 0; id < records.size(); id++) {

for (const auto &it : common.getBinary()) {

if (handle->keep\_map->find(it) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it;

offset++;

}

for (const auto &it : records[id].getBinary()) {

if (handle->keep\_map->find(it) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it;

offset++;

}

}

// All the values for binary features are 1.

std::fill(values\_flat.data(), values\_flat.data() + total\_size, 1);

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

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

}

}

};

REGISTER\_OP("GetContinuousFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads continuous features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: Datarecord keys (int64)

values: Datarecord values(float)

)doc");

class GetContinuousFeatures : public OpKernel {

public:

explicit GetContinuousFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_continuous\_size = count\_if\_exists(common.getContinuous(), handle->keep\_map);

int64 total\_continuous\_size = records.size() \* common\_continuous\_size;

for (int id = 0; id < records.size(); id++) {

total\_continuous\_size += count\_if\_exists(handle->records[id].getContinuous(),

handle->keep\_map);

}

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

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* values = nullptr;

Tensor\* ids = nullptr;

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto values\_flat = values->flat<float>();

auto ids\_flat = ids->flat<int64>();

for (int64 id = 0; id < records.size(); id++) {

for (const auto &it : common.getContinuous()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

values\_flat(offset) = it.second;

offset++;

}

for (const auto &it : records[id].getContinuous()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

values\_flat(offset) = it.second;

offset++;

}

}

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

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

}

}

};

REGISTER\_OP("GetDiscreteFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: int64")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads discrete features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

values: DataRecord values(int64)

)doc");

class GetDiscreteFeatures : public OpKernel {

public:

explicit GetDiscreteFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_discrete\_size = count\_if\_exists(common.getDiscrete(), handle->keep\_map);

int64 total\_discrete\_size = records.size() \* common\_discrete\_size;

for (int id = 0; id < records.size(); id++) {

total\_discrete\_size += count\_if\_exists(handle->records[id].getDiscrete(),

handle->keep\_map);

}

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

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* values = nullptr;

Tensor\* ids = nullptr;

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto values\_flat = values->flat<int64>();

auto ids\_flat = ids->flat<int64>();

for (int64 id = 0; id < records.size(); id++) {

for (const auto &it : common.getDiscrete()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

values\_flat(offset) = it.second;

offset++;

}

for (const auto &it : records[id].getDiscrete()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

values\_flat(offset) = it.second;

offset++;

}

}

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

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

}

}

};

REGISTER\_OP("GetStringFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("names: string")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads string features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

names: DataRecord values(string)

values: always set to 1 (float)

)doc");

class GetStringFeatures : public OpKernel {

public:

explicit GetStringFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_string\_size = count\_if\_exists(common.getString(), handle->keep\_map);

int64 total\_string\_size = records.size() \* common\_string\_size;

for (int id = 0; id < records.size(); id++) {

total\_string\_size += count\_if\_exists(handle->records[id].getString(),

handle->keep\_map);

}

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

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* names = nullptr;

Tensor\* ids = nullptr;

Tensor\*values = nullptr;

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

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto names\_flat = names->flat<string>();

auto ids\_flat = ids->flat<int64>();

auto values\_flat = values->flat<float>();

std::fill(values\_flat.data(), values\_flat.data() + total\_size, 1);

for (int64 id = 0; id < records.size(); id++) {

for (const auto &it : common.getString()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it.second;

offset++;

}

for (const auto &it : records[id].getString()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it.second;

offset++;

}

}

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

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

}

}

};

REGISTER\_OP("GetSparseBinaryFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("names: string")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads sparse binary features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

names: DataRecord values(string)

values: always set to 1 (float)

)doc");

class GetSparseBinaryFeatures : public OpKernel {

public:

explicit GetSparseBinaryFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_sparse\_binary\_size = count\_if\_exists(common.getSparseBinary(), handle->keep\_map);

int64 total\_sparse\_binary\_size = records.size() \* common\_sparse\_binary\_size;

for (int id = 0; id < records.size(); id++) {

total\_sparse\_binary\_size += count\_if\_exists(handle->records[id].getSparseBinary(),

handle->keep\_map);

}

const int total\_size = static\_cast<int>(total\_sparse\_binary\_size);

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* names = nullptr;

Tensor\* ids = nullptr;

Tensor\* values = nullptr;

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

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto names\_flat = names->flat<string>();

auto ids\_flat = ids->flat<int64>();

auto values\_flat = values->flat<float>();

// All the values for sparse binary features are 1.

std::fill(values\_flat.data(), values\_flat.data() + total\_size, 1);

for (int64 id = 0; id < records.size(); id++) {

for (const auto &it : common.getSparseBinary()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

for (const auto &it\_inner : it.second) {

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it\_inner;

offset++;

}

}

for (const auto &it : records[id].getSparseBinary()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

for (const auto &it\_inner : it.second) {

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it\_inner;

offset++;

}

}

}

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

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

}

}

};

REGISTER\_OP("GetSparseContinuousFeatures")

.Input("data\_record\_handle: resource")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: float")

.Output("names: string")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that reads sparse continuous features

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

values: DataRecord values(float)

names: DataRecord values(string)

)doc");

class GetSparseContinuousFeatures : public OpKernel {

public:

explicit GetSparseContinuousFeatures(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const auto &common = handle->common;

int64 common\_sparse\_continuous\_size = count\_if\_exists(common.getSparseContinuous(),

handle->keep\_map);

int64 total\_sparse\_continuous\_size = records.size() \* common\_sparse\_continuous\_size;

for (int id = 0; id < records.size(); id++) {

total\_sparse\_continuous\_size += count\_if\_exists(handle->records[id].getSparseContinuous(),

handle->keep\_map);

}

const int total\_size = static\_cast<int>(total\_sparse\_continuous\_size);

TensorShape shape = {total\_size};

Tensor\* keys = nullptr;

Tensor\* values = nullptr;

Tensor\* names = nullptr;

Tensor\* ids = nullptr;

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

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

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

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

uint64\_t offset = 0;

auto keys\_flat = keys->flat<int64>();

auto values\_flat = values->flat<float>();

auto names\_flat = names->flat<string>();

auto ids\_flat = ids->flat<int64>();

for (int64 id = 0; id < records.size(); id++) {

// copying the contents of the maps of maps

for (const auto &it : common.getSparseContinuous()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

// for each id; iterate through the number of maps corresponding to that id

for (const auto &it\_inner : it.second) {

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it\_inner.first;

values\_flat(offset) = it\_inner.second;

offset++;

}

}

// copying the contents of the maps of maps

for (const auto &it : records[id].getSparseContinuous()) {

if (handle->keep\_map->find(it.first) == handle->keep\_map->end()) continue;

// for each id; iterate through the number of maps corresponding to that id

for (const auto &it\_inner : it.second) {

ids\_flat(offset) = id;

keys\_flat(offset) = it.first;

names\_flat(offset) = it\_inner.first;

values\_flat(offset) = it\_inner.second;

offset++;

}

}

}

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

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

}

}

};

REGISTER\_OP("GetBatchSizeFromDataRecord")

.Input("data\_record\_handle: resource")

.Output("batch\_size: int64")

.SetShapeFn(shape\_inference::ScalarShape)

.Doc(R"doc(

A tensorflow OP that returns batch size from the data record.

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

batch\_size: Number of records held in the handle.

)doc");

class GetBatchSizeFromDataRecord : public OpKernel {

public:

explicit GetBatchSizeFromDataRecord(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

Tensor \*output;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, TensorShape({}), &output));

output->scalar<int64>()() = handle->records.size();

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

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

}

}

};

REGISTER\_OP("GetLabelsFromDataRecord")

.Input("data\_record\_handle: resource")

.Output("labels: float")

.Attr("default\_label: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns labels from the data record.

Attr

default\_label: The value used when a label is absent in a data record.

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

labels: A 2D tensor of size [batch\_size, num\_labels] containing the label values.

)doc");

class GetLabelsFromDataRecord : public OpKernel {

private:

float default\_label;

public:

explicit GetLabelsFromDataRecord(OpKernelConstruction\* context)

: OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("default\_label", &default\_label));

}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const int num\_labels = static\_cast<int>(handle->num\_labels);

TensorShape shape = {static\_cast<int64>(handle->records.size()), num\_labels};

Tensor \*labels;

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

// The default value of label is not present in data record is std::nanf

// For continuous labels, change that to a default\_label or label.

auto func = [this](float label) -> float {

return std::isnan(label) ? default\_label : label;

};

auto labels\_data = labels->flat<float>().data();

for (const auto &record : records) {

const auto& rec\_labels = record.labels();

labels\_data = std::transform(rec\_labels.begin(), rec\_labels.end(), labels\_data, func);

}

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

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

}

}

};

REGISTER\_OP("GetWeightsFromDataRecord")

.Input("data\_record\_handle: resource")

.Output("weights: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns weights from the data record.

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

weights: A 2D tensor of size [batch\_size, num\_weights] containing the weight values.

)doc");

class GetWeightsFromDataRecord : public OpKernel {

public:

explicit GetWeightsFromDataRecord(OpKernelConstruction\* context)

: OpKernel(context) {}

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

const int num\_weights = static\_cast<int>(handle->num\_weights);

TensorShape shape = {static\_cast<int64>(handle->records.size()), num\_weights};

Tensor \*weights;

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

auto weights\_data = weights->flat<float>().data();

for (const auto &record : records) {

const auto& rec\_weights = record.weights();

weights\_data = std::copy(rec\_weights.begin(), rec\_weights.end(), weights\_data);

}

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

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

}

}

};

template<typename ValueType, typename FeatureType, typename TensorType>

void SetValueGroup(

const FeatureType& type,

const int64& feature\_id,

const int64& id,

const ValueType& default\_value,

TensorType values\_flat) {

auto it = type.find(feature\_id);

values\_flat(id) = (it == type.end()) ? default\_value : it->second;

}

template<typename ValueType, typename TensorType>

// overloading for BinaryFeatures; as it needs to set a value of 1

void SetValueGroup(

const twml::DataRecord::BinaryFeatures& type,

const int64& feature\_id,

const int64& id,

const ValueType& default\_value,

TensorType values\_flat) {

auto it = type.find(feature\_id);

values\_flat(id) = (it == type.end()) ? default\_value : 1;

}

// Helper for Group Extraction of Dense Features

template<typename ValueType, typename FeatureType>

void ComputeHelperGroupFeaturesAsTensors(

OpKernelContext\* context,

const std::vector<int64>& feature\_ids,

ValueType& default\_value,

std::function<const FeatureType&(const twml::DataRecord&)> f) {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

// Output shape is 2D; where the first dimension corresponds to the batch\_size

// and the second corresponds to the number of features passed to the TF Op.

const int batch\_size = static\_cast<int64>(handle->records.size());

const int num\_feature\_ids = static\_cast<int>(feature\_ids.size());

TensorShape shape = {batch\_size, num\_feature\_ids};

// Define the output

Tensor\* values = nullptr;

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

auto values\_flat = values->flat<ValueType>();

for (int64 id = 0; id < records.size(); id++) {

const auto &type = f(records[id]);

const auto id\_offset = id \* feature\_ids.size();

for (int64 fid = 0; fid < feature\_ids.size(); fid++) {

auto feature\_id = feature\_ids[fid];

// The value is set to default if it does not exist in the current DataRecord

SetValueGroup(type, feature\_id, id\_offset + fid, default\_value, values\_flat);

}

}

}

// Helper for Single Extraction of Dense Features

template<typename ValueType, typename FeatureType>

void ComputeHelperFeaturesAsTensors(

OpKernelContext\* context,

ValueType& default\_value,

int64 feature\_id,

std::function<const FeatureType&(const twml::DataRecord&)> f) {

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

// Output shape is 2D; where the first dimension corresponds to the batch\_size

// and the second corresponds to the number of features passed to the TF Op.

const int total\_size = static\_cast<int64>(handle->records.size());

TensorShape shape = {total\_size};

// Define the output

Tensor\* values = nullptr;

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

auto values\_flat = values->flat<ValueType>();

for (int64 id = 0; id < records.size(); id++) {

const auto &type = f(records[id]);

SetValueGroup(type, feature\_id, id, default\_value, values\_flat);

}

}

REGISTER\_OP("GetBinaryAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Attr("default\_value: float")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the value of the feature\_id across multiple DataRecords

)doc");

class GetBinaryAsTensor : public OpKernel {

private:

int64 feature\_id;

float default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::BinaryFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::BinaryFeatures& { return record.getBinary(); };

ComputeHelperFeaturesAsTensors(context, default\_value, feature\_id, f);

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

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

}

}

};

REGISTER\_OP("GetContinuousAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Attr("default\_value: float")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the value of the feature\_id across multiple DataRecords

)doc");

class GetContinuousAsTensor : public OpKernel {

private:

int64 feature\_id;

float default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::ContinuousFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::ContinuousFeatures& { return record.getContinuous(); };

ComputeHelperFeaturesAsTensors(context, default\_value, feature\_id, f);

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

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

}

}

};

REGISTER\_OP("GetDiscreteAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Attr("default\_value: int")

.Output("values: int64")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the value of the feature\_id across multiple DataRecords

)doc");

class GetDiscreteAsTensor : public OpKernel {

private:

int64 feature\_id;

int64 default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::DiscreteFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::DiscreteFeatures& { return record.getDiscrete(); };

ComputeHelperFeaturesAsTensors(context, default\_value, feature\_id, f);

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

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

}

}

};

REGISTER\_OP("GetStringAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Attr("default\_value: string")

.Output("names: string")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

names: A Tensor corresponding to the value of the feature\_id across multiple DataRecords

)doc");

class GetStringAsTensor : public OpKernel {

private:

int64 feature\_id;

string default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::StringFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::StringFeatures& { return record.getString(); };

ComputeHelperFeaturesAsTensors(context, default\_value, feature\_id, f);

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

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

}

}

};

REGISTER\_OP("GetBinaryGroupAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_ids: list(int)")

.Attr("default\_value: float")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_ids: List of ids representing the features whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the values of the feature\_ids across multiple DataRecords

)doc");

class GetBinaryGroupAsTensor : public OpKernel {

private:

float default\_value;

std::vector<int64> feature\_ids;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_ids", &feature\_ids));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::BinaryFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::BinaryFeatures& { return record.getBinary(); };

ComputeHelperGroupFeaturesAsTensors(context, feature\_ids, default\_value, f);

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

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

}

}

};

REGISTER\_OP("GetContinuousGroupAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_ids: list(int)")

.Attr("default\_value: float")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_ids: List of ids representing the features whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the values of the feature\_ids across multiple DataRecords

)doc");

class GetContinuousGroupAsTensor : public OpKernel {

private:

float default\_value;

std::vector<int64> feature\_ids;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_ids", &feature\_ids));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::ContinuousFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::ContinuousFeatures& { return record.getContinuous(); };

ComputeHelperGroupFeaturesAsTensors(context, feature\_ids, default\_value, f);

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

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

}

}

};

REGISTER\_OP("GetDiscreteGroupAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_ids: list(int)")

.Attr("default\_value: int")

.Output("values: int64")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_ids: List of ids representing the features whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

values: A Tensor corresponding to the values of the feature\_ids across multiple DataRecords

)doc");

class GetDiscreteGroupAsTensor : public OpKernel {

private:

std::vector<int64> feature\_ids;

int64 default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_ids", &feature\_ids));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::DiscreteFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::DiscreteFeatures& { return record.getDiscrete(); };

ComputeHelperGroupFeaturesAsTensors(context, feature\_ids, default\_value, f);

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

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

}

}

};

REGISTER\_OP("GetStringGroupAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_ids: list(int)")

.Attr("default\_value: string")

.Output("names: string")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns a Dense Tensor with the values of a particular feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_ids: List of ids representing the features whose values will be extracted.

default\_value: default\_value to be inputted if the values are missing from the current DataRecord.

Outputs

names: A Tensor corresponding to the values of the feature\_ids across multiple DataRecords

)doc");

class GetStringGroupAsTensor : public OpKernel {

private:

std::vector<int64> feature\_ids;

string default\_value;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_ids", &feature\_ids));

OP\_REQUIRES\_OK(context, context->GetAttr("default\_value", &default\_value));

}

void Compute(OpKernelContext\* context) override {

try {

std::function<const twml::DataRecord::StringFeatures &(const twml::DataRecord &)> f =

[](const twml::DataRecord& record) ->const twml::DataRecord::StringFeatures& { return record.getString(); };

ComputeHelperGroupFeaturesAsTensors(context, feature\_ids, default\_value, f);

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

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

}

}

};

REGISTER\_OP("GetSparseBinaryAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Output("ids: int64")

.Output("keys: int64")

.Output("names: string")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns tensors corresponding to the ids, keys and names of a particular

feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

names: DataRecord values(string)

)doc");

class GetSparseBinaryAsTensor : public OpKernel {

private:

int64 feature\_id;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

}

void Compute(OpKernelContext\* context) override {

try {

// We need two passes to the data:

// 1 to compute the output size of the tensor

// 2 to copy the values to the tensor

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

// Creating a vector we increment every time a key is found

std::vector<std::string> temp\_names;

std::vector<int64> temp\_ids;

for (int64 id = 0; id < records.size(); id++) {

const auto &sparse\_binary = records[id].getSparseBinary();

auto it = sparse\_binary.find(feature\_id);

// Find all instances of key in DataRecord

if (it != sparse\_binary.end()) {

// insert to temp\_names all the values in the dictionary value

temp\_names.insert(temp\_names.end(), it->second.begin(), it->second.end());

temp\_ids.insert(temp\_ids.end(), it->second.size(), id);

}

}

// The total\_size will be the that of the saved vector

const int total\_size = static\_cast<int64>(temp\_names.size());

TensorShape shape = {total\_size};

Tensor\* ids = nullptr;

Tensor\* keys = nullptr;

Tensor\* names = nullptr;

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

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

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

auto keys\_flat = keys->flat<int64>();

auto names\_flat = names->flat<string>();

auto ids\_flat = ids->flat<int64>();

// The feature id value will always be the same

std::fill(keys\_flat.data(), keys\_flat.data() + total\_size, feature\_id);

std::copy(temp\_names.begin(), temp\_names.end(), names\_flat.data());

std::copy(temp\_ids.begin(), temp\_ids.end(), ids\_flat.data());

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

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

}

}

};

REGISTER\_OP("GetSparseContinuousAsTensor")

.Input("data\_record\_handle: resource")

.Attr("feature\_id: int")

.Output("ids: int64")

.Output("keys: int64")

.Output("names: string")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that returns tensors corresponding to the ids, keys, names and values of a particular

feature\_id.

Input

data\_record\_handle: Resource handle to DataRecord

Attr

feature\_id: Id representing the feature whose values will be extracted.

Outputs

ids: ids specifies the index of the records[id] in the batch (int64)

keys: DataRecord keys (int64)

names: DataRecord values(string)

values: DataRecord values(float)

)doc");

class GetSparseContinuousAsTensor : public OpKernel {

private:

int64 feature\_id;

public:

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

OP\_REQUIRES\_OK(context, context->GetAttr("feature\_id", &feature\_id));

}

void Compute(OpKernelContext\* context) override {

try {

// We need two passes to the data:

// 1 to compute the output size of the tensor

// 2 to copy the values to the tensor

auto handle = getHandle<DataRecordResource>(context, 0);

const auto &records = handle->records;

// Creating a vector we increment every time a key is found

std::vector<std::string> temp\_names;

std::vector<float> temp\_values;

std::vector<int64> temp\_ids;

for (int64 id = 0; id < records.size(); id++) {

const auto &sparse\_continuous = records[id].getSparseContinuous();

auto it = sparse\_continuous.find(feature\_id);

// Find all instances of key in DataRecord

if (it != sparse\_continuous.end()) {

// insert to temp\_names all the values in the dictionary value

auto value\_map = it->second;

for (auto& elem : value\_map) {

temp\_names.push\_back(elem.first);

temp\_values.push\_back(elem.second);

temp\_ids.push\_back(id);

}

}

}

// The total\_size will be the that of the saved vector

const int total\_size = static\_cast<int64>(temp\_names.size());

TensorShape shape = {total\_size};

Tensor\* ids = nullptr;

Tensor\* keys = nullptr;

Tensor\* names = nullptr;

Tensor\* values = nullptr;

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

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

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

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

auto keys\_flat = keys->flat<int64>();

auto names\_flat = names->flat<string>();

auto ids\_flat = ids->flat<int64>();

auto values\_flat = values->flat<float>();

// The feature id value will always be the same

std::fill(keys\_flat.data(), keys\_flat.data() + total\_size, feature\_id);

std::copy(temp\_names.begin(), temp\_names.end(), names\_flat.data());

std::copy(temp\_ids.begin(), temp\_ids.end(), ids\_flat.data());

std::copy(temp\_values.begin(), temp\_values.end(), values\_flat.data());

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

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

}

}

};

// Helper function to add ids, keys and values to common vector

inline void addIdsKeysValuesToVectors(

const int64 id,

const int64 key,

const double value,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

ids.push\_back(id);

keys.push\_back(key);

values.push\_back(value);

}

struct KeepFeatures {

KeepFeatures() : vec(), set() {}

template<typename ContainerType>

KeepFeatures(const std::vector<int64> &keep\_features,

const ContainerType \*const container) {

vec.reserve(keep\_features.size());

#ifdef USE\_DENSE\_HASH

set.resize(keep\_features.size());

set.set\_empty\_key(0);

#else

set.reserve(keep\_features.size());

#endif // USE\_DENSE\_HASH

set.max\_load\_factor(0.5);

for (const auto &elem : keep\_features) {

if (container->find(elem) == container->end()) continue;

vec.push\_back(elem);

set.insert(elem);

}

}

size\_t size() const {

return vec.size();

}

std::vector<int64> vec;

twml::Set<int64> set;

};

// Helper Function to Filter and Hash Feature for Binary Features

void filterAndHashFeature(

const twml::DataRecord::BinaryFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

addIdsKeysValuesToVectors(current\_id, \*iter, 1, ids, keys, values);

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem) == keep\_features.set.end()) continue;

addIdsKeysValuesToVectors(current\_id, elem, 1, ids, keys, values);

}

}

}

// Helper Function to Filter and Hash Feature for Continuous Features

void filterAndHashFeature(

const twml::DataRecord::ContinuousFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

addIdsKeysValuesToVectors(current\_id, iter->first, iter->second, ids, keys, values);

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

addIdsKeysValuesToVectors(current\_id, elem.first, elem.second, ids, keys, values);

}

}

}

// Helper Function to Filter and Hash Feature for Discrete Features

void filterAndHashFeature(

const twml::DataRecord::DiscreteFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

int64\_t key = twml::mixDiscreteIdAndValue(iter->first, iter->second);

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

int64\_t key = twml::mixDiscreteIdAndValue(elem.first, elem.second);

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

}

}

// Helper Function to Filter and Hash Feature for String Features

void filterAndHashFeature(

const twml::DataRecord::StringFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

int64\_t key = twml::mixStringIdAndValue(

iter->first,

iter->second.size(),

reinterpret\_cast<const uint8\_t\*>(iter->second.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

int64\_t key = twml::mixStringIdAndValue(

elem.first,

elem.second.size(),

reinterpret\_cast<const uint8\_t\*>(elem.second.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

}

}

// Helper Function to Filter and Hash Feature for Sparse Binary Features

void filterAndHashFeature(

const twml::DataRecord::SparseBinaryFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

for (const auto &name : iter->second) {

int64\_t key = twml::mixStringIdAndValue(iter->first, name.size(),

reinterpret\_cast<const uint8\_t\*>(name.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

for (const auto &name : elem.second) {

int64\_t key = twml::mixStringIdAndValue(elem.first, name.size(),

reinterpret\_cast<const uint8\_t\*>(name.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, 1, ids, keys, values);

}

}

}

}

// Helper Function to Filter and Hash Feature for Sparse Continuous Features

void filterAndHashFeature(

const twml::DataRecord::SparseContinuousFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

for (const auto &map : iter->second) {

int64\_t key = twml::mixStringIdAndValue(

iter->first,

map.first.size(),

reinterpret\_cast<const uint8\_t\*>(map.first.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, map.second, ids, keys, values);

}

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

for (const auto &map : elem.second) {

int64\_t key = twml::mixStringIdAndValue(

elem.first,

map.first.size(),

reinterpret\_cast<const uint8\_t\*>(map.first.c\_str()));

addIdsKeysValuesToVectors(current\_id, key, map.second, ids, keys, values);

}

}

}

}

// Helper Function to Filter and Hash Feature for Sparse Continuous Features

void filterAndHashFeatureCompat(

const twml::DataRecord::SparseContinuousFeatures& features,

const int64 current\_id,

const KeepFeatures &keep\_features,

std::vector<int64>& ids,

std::vector<int64>& keys,

std::vector<float>& values) {

if (keep\_features.size() < 2 \* features.size()) {

for (const auto &f : keep\_features.vec) {

const auto &iter = features.find(f);

if (iter == features.end()) continue;

for (const auto &map : iter->second) {

int64\_t key = twml::featureId(map.first);

addIdsKeysValuesToVectors(current\_id, key, map.second, ids, keys, values);

}

}

} else {

for (const auto &elem : features) {

if (keep\_features.set.find(elem.first) == keep\_features.set.end()) continue;

for (const auto &map : elem.second) {

int64\_t key = twml::featureId(map.first);

addIdsKeysValuesToVectors(current\_id, key, map.second, ids, keys, values);

}

}

}

}

void copy\_if\_exists(std::vector<int64>& out,

const std::vector<int64>& in,

const twml::Map<int64\_t, int64\_t> \*const map) {

out.reserve(in.size());

for (const auto &elem : in) {

if (map->find(elem) == map->end()) continue;

out.push\_back(elem);

}

}

void ComputeHashedFeaturesAsTensor(OpKernelContext\* context,

const DataRecordResource \*const handle,

const KeepFeatures &binary\_keep\_features,

const KeepFeatures &continuous\_keep\_features,

const KeepFeatures &discrete\_keep\_features,

const KeepFeatures &string\_keep\_features,

const KeepFeatures &sparse\_binary\_keep\_features,

const KeepFeatures &sparse\_continuous\_keep\_features,

bool sparse\_continuous\_compatibility) {

const auto &records = handle->records;

uint64\_t estimated\_size = (binary\_keep\_features.size() + continuous\_keep\_features.size() +

discrete\_keep\_features.size() + string\_keep\_features.size() +

sparse\_binary\_keep\_features.size() +

sparse\_continuous\_keep\_features.size());

// Construct temporary vectors for common features

std::vector<int64> common\_ids, common\_keys, temp\_ids, temp\_keys;

std::vector<float> common\_values, temp\_values;

common\_ids.reserve(estimated\_size);

common\_keys.reserve(estimated\_size);

common\_values.reserve(estimated\_size);

const auto &common\_binary = handle->common.getBinary();

const auto &common\_continuous = handle->common.getContinuous();

const auto &common\_discrete = handle->common.getDiscrete();

const auto &common\_string = handle->common.getString();

const auto &common\_sparse\_binary = handle->common.getSparseBinary();

const auto &common\_sparse\_continuous = handle->common.getSparseContinuous();

filterAndHashFeature(common\_binary, 0, binary\_keep\_features,

common\_ids, common\_keys, common\_values);

filterAndHashFeature(common\_continuous, 0, continuous\_keep\_features,

common\_ids, common\_keys, common\_values);

filterAndHashFeature(common\_discrete, 0, discrete\_keep\_features,

common\_ids, common\_keys, common\_values);

filterAndHashFeature(common\_string, 0, string\_keep\_features,

common\_ids, common\_keys, common\_values);

filterAndHashFeature(common\_sparse\_binary, 0, sparse\_binary\_keep\_features,

common\_ids, common\_keys, common\_values);

if (sparse\_continuous\_compatibility) {

filterAndHashFeatureCompat(common\_sparse\_continuous, 0, sparse\_continuous\_keep\_features,

common\_ids, common\_keys, common\_values);

} else {

filterAndHashFeature(common\_sparse\_continuous, 0, sparse\_continuous\_keep\_features,

common\_ids, common\_keys, common\_values);

}

common\_ids.clear();

// Construct temporary vectors for all features

estimated\_size = (estimated\_size + common\_keys.size()) \* records.size();

temp\_ids.reserve(estimated\_size);

temp\_keys.reserve(estimated\_size);

temp\_values.reserve(estimated\_size);

for (int64 id = 0; id < records.size(); id++) {

temp\_ids.insert(temp\_ids.end(), common\_keys.size(), id);

temp\_keys.insert(temp\_keys.end(), common\_keys.begin(), common\_keys.end());

temp\_values.insert(temp\_values.end(), common\_values.begin(), common\_values.end());

const auto &binary = records[id].getBinary();

const auto &continuous = records[id].getContinuous();

const auto &discrete = records[id].getDiscrete();

const auto &str = records[id].getString();

const auto &sparse\_binary = records[id].getSparseBinary();

const auto &sparse\_continuous = records[id].getSparseContinuous();

filterAndHashFeature(binary, id, binary\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

filterAndHashFeature(continuous, id, continuous\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

filterAndHashFeature(discrete, id, discrete\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

filterAndHashFeature(str, id, string\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

filterAndHashFeature(sparse\_binary, id, sparse\_binary\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

if (sparse\_continuous\_compatibility) {

filterAndHashFeatureCompat(sparse\_continuous, id, sparse\_continuous\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

} else {

filterAndHashFeature(sparse\_continuous, id, sparse\_continuous\_keep\_features,

temp\_ids, temp\_keys, temp\_values);

}

}

// Copy the temporary vectors into the output Tensors

TensorShape shape = {static\_cast<int64>(temp\_ids.size())};

Tensor\* ids = nullptr;

Tensor\* keys = nullptr;

Tensor\* values = nullptr;

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

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

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

auto ids\_flat = ids->flat<int64>();

auto keys\_flat = keys->flat<int64>();

auto values\_flat = values->flat<float>();

std::copy(temp\_ids.begin(), temp\_ids.end(), ids\_flat.data());

std::copy(temp\_keys.begin(), temp\_keys.end(), keys\_flat.data());

std::copy(temp\_values.begin(), temp\_values.end(), values\_flat.data());

}

REGISTER\_OP("GetHashedFeaturesAsSparseTensor")

.Input("data\_record\_handle: resource")

.Attr("binary\_keep\_features: list(int)")

.Attr("continuous\_keep\_features: list(int)")

.Attr("discrete\_keep\_features: list(int)")

.Attr("string\_keep\_features: list(int)")

.Attr("sparse\_binary\_keep\_features: list(int)")

.Attr("sparse\_continuous\_keep\_features: list(int)")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP for returning required features of different type as

a single sparse tensor. Hashing trick is applied.

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records in the batch (int64)

keys: DataRecord keys (int64)

values: DataRecord values (float)

)doc");

class GetHashedFeaturesAsSparseTensor: public OpKernel {

public:

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

// Get the list of features to keep for each feature type

OP\_REQUIRES\_OK(context, context->GetAttr("binary\_keep\_features", &binary\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("continuous\_keep\_features", &continuous\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("discrete\_keep\_features", &discrete\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("string\_keep\_features", &string\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("sparse\_binary\_keep\_features", &sparse\_binary\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("sparse\_continuous\_keep\_features", &sparse\_continuous\_keep\_features\_));

}

private:

std::vector<int64> binary\_keep\_features\_, continuous\_keep\_features\_, discrete\_keep\_features\_;

std::vector<int64> string\_keep\_features\_, sparse\_binary\_keep\_features\_, sparse\_continuous\_keep\_features\_;

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

// Create a new list of keep features based on the original keep\_set.

// This is to ensure compatibility with existing behavior such as:

// - Ensure no new features are decoded in this op.

// - Ensure labels or weights dont get included here.

// TODO: Should we return features requested by user here even if they are labels / weights?

KeepFeatures binary\_keep\_features(binary\_keep\_features\_, handle->keep\_map);

KeepFeatures continuous\_keep\_features(continuous\_keep\_features\_, handle->keep\_map);

KeepFeatures discrete\_keep\_features(discrete\_keep\_features\_, handle->keep\_map);

KeepFeatures string\_keep\_features(string\_keep\_features\_, handle->keep\_map);

KeepFeatures sparse\_binary\_keep\_features(sparse\_binary\_keep\_features\_, handle->keep\_map);

KeepFeatures sparse\_continuous\_keep\_features(sparse\_continuous\_keep\_features\_, handle->keep\_map);

ComputeHashedFeaturesAsTensor(context, handle.get(),

binary\_keep\_features,

continuous\_keep\_features,

discrete\_keep\_features,

string\_keep\_features,

sparse\_binary\_keep\_features,

sparse\_continuous\_keep\_features,

false);

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

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

}

}

};

REGISTER\_OP("GetHashedFeaturesAsSparseTensorV2")

.Input("data\_record\_handle: resource")

.Attr("binary\_keep\_features: list(int)")

.Attr("continuous\_keep\_features: list(int)")

.Attr("discrete\_keep\_features: list(int)")

.Attr("string\_keep\_features: list(int)")

.Attr("sparse\_binary\_keep\_features: list(int)")

.Attr("sparse\_continuous\_keep\_features: list(int)")

.Attr("keep\_features: list(int)")

.Attr("keep\_codes: list(int)")

.Attr("decode\_mode: int = 0")

.Output("ids: int64")

.Output("keys: int64")

.Output("values: float")

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

return Status::OK();

}).Doc(R"doc(

A tensorflow OP for returning required features of different type as

a single sparse tensor. Hashing trick is applied.

Input

data\_record\_handle: Resource handle to DataRecord

Outputs

ids: ids specifies the index of the records in the batch (int64)

keys: DataRecord keys (int64)

values: DataRecord values (float)

)doc");

class GetHashedFeaturesAsSparseTensorV2: public OpKernel {

public:

explicit GetHashedFeaturesAsSparseTensorV2(OpKernelConstruction\* context): OpKernel(context) {

std::vector<int64> keep\_features;

std::vector<int64> keep\_codes;

std::vector<int64> binary\_keep\_features\_, continuous\_keep\_features\_, discrete\_keep\_features\_;

std::vector<int64> string\_keep\_features\_, sparse\_binary\_keep\_features\_, sparse\_continuous\_keep\_features\_;

// Get the list of features to keep for each feature type

OP\_REQUIRES\_OK(context, context->GetAttr("binary\_keep\_features", &binary\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("continuous\_keep\_features", &continuous\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("discrete\_keep\_features", &discrete\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("string\_keep\_features", &string\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("sparse\_binary\_keep\_features", &sparse\_binary\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("sparse\_continuous\_keep\_features", &sparse\_continuous\_keep\_features\_));

OP\_REQUIRES\_OK(context, context->GetAttr("keep\_features", &keep\_features));

OP\_REQUIRES\_OK(context, context->GetAttr("keep\_codes", &keep\_codes));

OP\_REQUIRES\_OK(context, context->GetAttr("decode\_mode", &m\_decode\_mode));

twml::Map<int64\_t, int64\_t> keep\_map;

#ifdef USE\_DENSE\_HASH

keep\_map.set\_empty\_key(0);

#endif // USE\_DENSE\_HASH

for (uint64\_t i = 0; i < keep\_features.size(); i++) {

keep\_map[keep\_features[i]] = keep\_codes[i];

}

binary\_keep\_features = KeepFeatures(binary\_keep\_features\_, &keep\_map);

continuous\_keep\_features = KeepFeatures(continuous\_keep\_features\_, &keep\_map);

discrete\_keep\_features = KeepFeatures(discrete\_keep\_features\_, &keep\_map);

string\_keep\_features = KeepFeatures(string\_keep\_features\_, &keep\_map);

sparse\_binary\_keep\_features = KeepFeatures(sparse\_binary\_keep\_features\_, &keep\_map);

sparse\_continuous\_keep\_features = KeepFeatures(sparse\_continuous\_keep\_features\_, &keep\_map);

}

private:

KeepFeatures binary\_keep\_features, continuous\_keep\_features, discrete\_keep\_features;

KeepFeatures string\_keep\_features, sparse\_binary\_keep\_features, sparse\_continuous\_keep\_features;

int64 m\_decode\_mode;

void Compute(OpKernelContext\* context) override {

try {

auto handle = getHandle<DataRecordResource>(context, 0);

// Create a new list of keep features based on the original keep\_set.

// This is to ensure compatibility with existing behavior such as:

// - Ensure no new features are decoded in this op.

// - Ensure labels or weights dont get included here.

// TODO: Should we return features requested by user here even if they are labels / weights?

ComputeHashedFeaturesAsTensor(context, handle.get(),

binary\_keep\_features,

continuous\_keep\_features,

discrete\_keep\_features,

string\_keep\_features,

sparse\_binary\_keep\_features,

sparse\_continuous\_keep\_features,

m\_decode\_mode == 0);

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

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

}

}

};

#define REGISTER\_DECODE\_DATA\_RECORD(InputType) \

REGISTER\_KERNEL\_BUILDER( \

Name("DecodeDataRecord") \

.Device(DEVICE\_CPU) \

.TypeConstraint<InputType>("InputType"), \

DecodeDataRecord<InputType>); \

REGISTER\_DECODE\_DATA\_RECORD(uint8)

REGISTER\_DECODE\_DATA\_RECORD(string)

#define REGISTER\_GETTER(FIELD) \

REGISTER\_KERNEL\_BUILDER( \

Name("Get" #FIELD "Features") \

.Device(DEVICE\_CPU), \

Get##FIELD##Features); \

#define REGISTER\_GETTER\_FROM\_DR(FIELD) \

REGISTER\_KERNEL\_BUILDER( \

Name("Get" #FIELD "FromDataRecord") \

.Device(DEVICE\_CPU), \

Get##FIELD##FromDataRecord); \

#define REGISTER\_GETTER\_AS\_TENSOR(FIELD) \

REGISTER\_KERNEL\_BUILDER( \

Name("Get" #FIELD "AsTensor") \

.Device(DEVICE\_CPU), \

Get##FIELD##AsTensor); \

#define REGISTER\_GETTER\_GROUP\_AS\_TENSOR(FIELD) \

REGISTER\_KERNEL\_BUILDER( \

Name("Get" #FIELD "GroupAsTensor") \

.Device(DEVICE\_CPU), \

Get##FIELD##GroupAsTensor); \

REGISTER\_GETTER(Binary)

REGISTER\_GETTER(Continuous)

REGISTER\_GETTER(Discrete)

REGISTER\_GETTER(String)

REGISTER\_GETTER(SparseBinary)

REGISTER\_GETTER(SparseContinuous)

REGISTER\_GETTER\_FROM\_DR(BatchSize)

REGISTER\_GETTER\_FROM\_DR(Labels)

REGISTER\_GETTER\_FROM\_DR(Weights)

REGISTER\_GETTER\_AS\_TENSOR(Binary)

REGISTER\_GETTER\_AS\_TENSOR(Continuous)

REGISTER\_GETTER\_AS\_TENSOR(Discrete)

REGISTER\_GETTER\_AS\_TENSOR(String)

REGISTER\_GETTER\_AS\_TENSOR(SparseBinary)

REGISTER\_GETTER\_AS\_TENSOR(SparseContinuous)

REGISTER\_GETTER\_GROUP\_AS\_TENSOR(Binary)

REGISTER\_GETTER\_GROUP\_AS\_TENSOR(Continuous)

REGISTER\_GETTER\_GROUP\_AS\_TENSOR(Discrete)

REGISTER\_GETTER\_GROUP\_AS\_TENSOR(String)

REGISTER\_KERNEL\_BUILDER(

Name("GetHashedFeaturesAsSparseTensor")

.Device(DEVICE\_CPU),

GetHashedFeaturesAsSparseTensor);

REGISTER\_KERNEL\_BUILDER(

Name("GetHashedFeaturesAsSparseTensorV2")

.Device(DEVICE\_CPU),

GetHashedFeaturesAsSparseTensorV2);