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

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

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

#include <cstdint>

#include <twml.h>

#include "tensorflow\_utils.h"

#include "resource\_utils.h"

#include <iterator>

template<typename InputType, typename RecordType>

class DecodeBatchPredictionRequestKernel : public OpKernel {

public:

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

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;

}

}

protected:

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;

int64 m\_decode\_mode;

template<typename ResourceType>

void Decode(OpKernelContext\* context, ResourceType \*resource) {

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

const uint8\_t \*input\_bytes = getInputBytes<InputType>(resource->input, 0);

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

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

typename RecordType::Reader reader;

twml::GenericBatchPredictionRequest<RecordType> bpr(num\_labels, num\_weights);

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

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

reader.setBuffer(input\_bytes);

reader.setDecodeMode(m\_decode\_mode);

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

if (num\_weights != 0) {

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

}

bpr.decode(reader);

resource->common = std::move(bpr.common());

resource->records = std::move(bpr.requests());

resource->num\_labels = num\_labels;

resource->num\_weights = num\_weights;

}

};

REGISTER\_OP("DecodeAndHashBatchPredictionRequestV2")

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

.Input("input\_bytes: InputType")

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

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

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

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

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

.Output("hashed\_data\_record\_handle: resource")

.SetShapeFn(shape\_inference::ScalarShape)

.Doc(R"doc(

A tensorflow OP that decodes a list/batch of data records and creates a handle to the batch of hashed data records.

Compared to DecodeAndHashBatchPredictionRequest, DecodeAndHashBatchPredictionRequestV2 is used for training instead

of serving. Thus label\_features and weight\_features[optional] must be passed, and labels and weights are extracted in

the output.

DecodeAndHashBatchPredictionRequestV2 controls what DataRecords we want to process together in a batch in training.

For instance, we can put all instances for a query in the same batch when training a ranking model.

Notice that this OP was added separately to make sure we would not break the API for DecodeAndHashBatchPredictionRequest.

It requires some discussions if we merge the two ops into a single .cpp file in a future API revision.

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.

decode\_mode: integer, indicates which decoding method to use. Let a sparse continuous

have a feature\_name and a dict of {name: value}. 0 indicates feature\_ids are computed

as hash(name). 1 indicates feature\_ids are computed as hash(feature\_name, name)

Input

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

Outputs

hashed\_data\_record\_handle: A resource handle to the HashedDataRecordResource containing batch of HashedDataRecords.

)doc");

template<typename InputType>

class DecodeAndHashBatchPredictionRequestV2 :

public DecodeBatchPredictionRequestKernel<InputType, twml::HashedDataRecord> {

public:

DecodeAndHashBatchPredictionRequestV2(OpKernelConstruction \*context)

: DecodeBatchPredictionRequestKernel<InputType, twml::HashedDataRecord>(context) {

}

private:

void Compute(OpKernelContext\* context) override {

try {

HashedDataRecordResource \*resource = nullptr;

OP\_REQUIRES\_OK(

context,

makeResourceHandle<HashedDataRecordResource>(context, 0, &resource));

this->Decode(context, resource);

// Each datarecord has a copy of common features.

// Initialize total\_size by common\_size \* num\_records

int64 common\_size = static\_cast<int64>(resource->common.totalSize());

int64 num\_records = static\_cast<int64>(resource->records.size());

int64 total\_size = common\_size \* num\_records;

for (const auto &record : resource->records) {

total\_size += static\_cast<int64>(record.totalSize());

}

resource->total\_size = total\_size;

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

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

}

}

};

REGISTER\_OP("DecodeBatchPredictionRequestV2")

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

.Input("input\_bytes: InputType")

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

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

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

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

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

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

.SetShapeFn(shape\_inference::ScalarShape)

.Doc(R"doc(

A tensorflow OP that decodes batch prediction request and creates a handle to the batch of data records.

Attr

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

keep\_codes: their corresponding code.

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

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

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

decode\_mode: reserved, do not use.

Input

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

Outputs

data\_record\_handle: A resource handle to the DataRecordResource containing batch of DataRecords.

)doc");

template<typename InputType>

class DecodeBatchPredictionRequestV2 :

public DecodeBatchPredictionRequestKernel<InputType, twml::DataRecord> {

public:

DecodeBatchPredictionRequestV2(OpKernelConstruction \*context)

: DecodeBatchPredictionRequestKernel<InputType, twml::DataRecord>(context) {

}

private:

void Compute(OpKernelContext\* context) override {

try {

DataRecordResource \*resource = nullptr;

OP\_REQUIRES\_OK(

context,

makeResourceHandle<DataRecordResource>(context, 0, &resource));

this->Decode(context, resource);

resource->keep\_map = &(this->m\_keep\_map);

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

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

}

}

};

#define REGISTER\_DECODE\_OPS(InputType) \

REGISTER\_KERNEL\_BUILDER( \

Name("DecodeAndHashBatchPredictionRequestV2") \

.Device(DEVICE\_CPU) \

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

DecodeAndHashBatchPredictionRequestV2<InputType>); \

REGISTER\_KERNEL\_BUILDER( \

Name("DecodeBatchPredictionRequestV2") \

.Device(DEVICE\_CPU) \

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

DecodeBatchPredictionRequestV2<InputType>); \

REGISTER\_DECODE\_OPS(uint8)

REGISTER\_DECODE\_OPS(string)