#include "internal/endianutils.h"

#include "internal/error.h"

#include "internal/thrift.h"

#include <twml/Tensor.h>

#include <twml/BatchPredictionResponse.h>

#include <twml/DataRecord.h>

#include <twml/ThriftWriter.h>

#include <twml/DataRecordWriter.h>

#include <inttypes.h>

#include <stdint.h>

#include <unistd.h>

#include <string.h>

#include <algorithm>

// When the number of predictions is very high, as some cases that Ads wants, the generic thrift

// encoder becomes super expensive because we have to deal with lua tables.

// This function is a special operation to efficiently write a batch prediction responses based on

// tensors.

namespace twml {

BatchPredictionResponse::BatchPredictionResponse(

const Tensor &keys, const Tensor &values,

const Tensor &dense\_keys, const std::vector<RawTensor> &dense\_values

) : keys\_(keys), values\_(values), dense\_keys\_(dense\_keys), dense\_values\_(dense\_values) {

// determine batch size

if (values\_.getNumDims() > 0) {

batch\_size\_ = values\_.getDim(0);

} else if (dense\_keys\_.getNumElements() < 1) {

throw twml::Error(TWML\_ERR\_TYPE, "Continuous values and dense tensors are both empty");

} else if (dense\_keys\_.getNumElements() != dense\_values\_.size()) {

throw twml::Error(TWML\_ERR\_TYPE, "Number of tensors not equal to number of keys");

} else {

// dim 0 for each tensor indexes batch elements

std::vector<uint64\_t> batch\_sizes;

batch\_sizes.reserve(dense\_values\_.size());

for (int i = 0; i < dense\_values\_.size(); i++)

batch\_sizes.push\_back(dense\_values\_.at(i).getDim(0));

if (std::adjacent\_find(

batch\_sizes.begin(),

batch\_sizes.end(),

std::not\_equal\_to<uint64\_t>()) != batch\_sizes.end())

throw twml::Error(TWML\_ERR\_TYPE, "Batch size (dim 0) for all tensors must be the same");

batch\_size\_ = dense\_values.at(0).getDim(0);

}

}

void BatchPredictionResponse::encode(twml::ThriftWriter &thrift\_writer) {

if (hasContinuous()) {

switch (values\_.getType()) {

case TWML\_TYPE\_FLOAT:

serializePredictions<float>(thrift\_writer);

break;

case TWML\_TYPE\_DOUBLE:

serializePredictions<double>(thrift\_writer);

break;

default:

throw twml::Error(TWML\_ERR\_TYPE, "Predictions must be float or double.");

}

} else {

// dense tensor predictions

serializePredictions<double>(thrift\_writer);

}

}

template <typename T>

void BatchPredictionResponse::serializePredictions(twml::ThriftWriter &thrift\_writer) {

twml::DataRecordWriter record\_writer = twml::DataRecordWriter(thrift\_writer);

// start BatchPredictionResponse

thrift\_writer.writeStructFieldHeader(TTYPE\_LIST, BPR\_PREDICTIONS);

thrift\_writer.writeListHeader(TTYPE\_STRUCT, getBatchSize());

for (int i = 0; i < getBatchSize(); i++) {

twml::DataRecord record = twml::DataRecord();

if (hasContinuous()) {

const T \*values = values\_.getData<T>();

const int64\_t \*local\_keys = keys\_.getData<int64\_t>();

const T \*local\_values = values + (i \* getPredictionSize());

record.addContinuous(local\_keys, getPredictionSize(), local\_values);

}

if (hasDenseTensors()) {

const int64\_t \*local\_dense\_keys = dense\_keys\_.getData<int64\_t>();

for (int j = 0; j < dense\_keys\_.getNumElements(); j++) {

const RawTensor &dense\_value = dense\_values\_.at(j).getSlice(i);

record.addRawTensor(local\_dense\_keys[j], dense\_value);

}

}

record\_writer.write(record);

}

// end BatchPredictionResponse

thrift\_writer.writeStructStop();

}

// calculate expected binary Thrift size (no memory is copied)

uint64\_t BatchPredictionResponse::encodedSize() {

bool dry\_mode = true;

twml::ThriftWriter dry\_writer = twml::ThriftWriter(nullptr, 0, dry\_mode);

encode(dry\_writer);

return dry\_writer.getBytesWritten();

}

void BatchPredictionResponse::write(Tensor &result) {

size\_t result\_size = result.getNumElements();

uint8\_t \*result\_data = result.getData<uint8\_t>();

if (result\_size != this->encodedSize()) {

throw twml::Error(TWML\_ERR\_SIZE, "Sizes do not match");

}

twml::ThriftWriter writer = twml::ThriftWriter(result\_data, result\_size);

encode(writer);

}

} // namespace twml