tensorflow / **tensorflow** Public

Code | Issues 2.1k | Pull requests 283 | Actions | Projects 1 | ...

ca6f96b62a

tensorflow / tensorflow / lite / kernels / **embedding_lookup_sparse.cc**

mihaimaruseac Fix a dangerous integer overflow and a malloc of negative size. ... ✕ History

6 contributors

269 lines (232 sloc) | 9.8 KB

```
/* Copyright 2017 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

// Op that looks up items from a sparse tensor in an embedding matrix.
// The sparse lookup tensor is represented by three individual tensors: lookup,
// indices, and dense_shape. The representation assume that the corresponding
// dense tensor would satisfy:
//   * dense.shape = dense_shape
//   * dense[tuple(indices[i])] = lookup[i]
//
// By convention, indices should be sorted.
//
// Options:
//   combiner: The reduction op (SUM, MEAN, SQRTN).
//     * SUM computes the weighted sum of the embedding results.
//     * MEAN is the weighted sum divided by the total weight.
//     * SQRTN is the weighted sum divided by the square root of the sum of the
//       squares of the weights.
//
// Input:
//     Tensor[0]: Ids to lookup, dim.size == 1, int32.
//     Tensor[1]: Indices, int32.
//     Tensor[2]: Dense shape, int32.
//     Tensor[3]: Weights to use for aggregation, float.
//     Tensor[4]: Params, a matrix of multi-dimensional items,
//                dim.size >= 2, float.
//
// Output:
//   A (dense) tensor representing the combined embeddings for the sparse ids.
//   For each row in the sparse tensor represented by (lookup, indices, shape)
//   the op looks up the embeddings for all ids in that row, multiplies them by
//   the corresponding weight, and combines these embeddings as specified in the
//   last dimension.
//
//   Output.dim = [l0, ... , ln-1, e1, ..., em]
//   Where dense_shape == [l0, ..., ln] and Tensor[4].dim == [e0, e1, ..., em]
//
//   For instance, if params is a 10x20 matrix and ids, weights are:
//
//   [0, 0]: id 1, weight 2.0
//   [0, 1]: id 3, weight 0.5
//   [1, 0]: id 0, weight 1.0
//   [2, 3]: id 1, weight 3.0
//
//   with combiner=MEAN, then the output will be a (3, 20) tensor where:
//
//   output[0, :] = (params[1, :] * 2.0 + params[3, :] * 0.5) / (2.0 + 0.5)
//   output[1, :] = (params[0, :] * 1.0) / 1.0
//   output[2, :] = (params[1, :] * 3.0) / 3.0
//
//   When indices are out of bound, the op will not succeed.

#include <stdint.h>

#include <algorithm>
#include <cmath>

#include "tensorflow/lite/c/builtin_op_data.h"
#include "tensorflow/lite/c/common.h"
#include "tensorflow/lite/kernels/internal/tensor_ctypes.h"
#include "tensorflow/lite/kernels/internal/tensor_utils.h"
#include "tensorflow/lite/kernels/kernel_util.h"

namespace tflite {
namespace ops {
namespace builtin {
```

```

namespace {

TfLiteStatus Prepare(TfLiteContext* context, TfLiteNode* node) {
  TF_LITE_ENSURE_EQ(context, NumInputs(node), 5);
  TF_LITE_ENSURE_EQ(context, NumOutputs(node), 1);

  const TfLiteTensor* ids;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 0, &ids));
  TF_LITE_ENSURE_EQ(context, NumDimensions(ids), 1);
  TF_LITE_ENSURE_EQ(context, ids->type, kTfLiteInt32);

  const TfLiteTensor* indices;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 1, &indices));
  TF_LITE_ENSURE_EQ(context, NumDimensions(indices), 2);
  TF_LITE_ENSURE_EQ(context, indices->type, kTfLiteInt32);

  const TfLiteTensor* shape;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 2, &shape));
  TF_LITE_ENSURE_EQ(context, NumDimensions(shape), 1);
  TF_LITE_ENSURE_EQ(context, shape->type, kTfLiteInt32);

  const TfLiteTensor* weights;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 3, &weights));
  TF_LITE_ENSURE_EQ(context, NumDimensions(weights), 1);
  TF_LITE_ENSURE_EQ(context, weights->type, kTfLiteFloat32);

  TF_LITE_ENSURE_EQ(context, SizeOfDimension(indices, 0),
                    SizeOfDimension(ids, 0));
  TF_LITE_ENSURE_EQ(context, SizeOfDimension(indices, 0),
                    SizeOfDimension(weights, 0));

  const TfLiteTensor* value;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 4, &value));
  TF_LITE_ENSURE(context, NumDimensions(value) >= 2);

  // Mark the output as a dynamic tensor.
  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context, GetOutputSafe(context, node, 0, &output));
  TF_LITE_ENSURE_TYPES_EQ(context, output->type, kTfLiteFloat32);
  output->allocation_type = kTfLiteDynamic;

  return kTfLiteOk;
}

void FinalizeAggregation(TfLiteCombinerType combiner, int num_elements,
                         float current_total_weight,
                         float current_squares_weight, int embedding_size,
                         float* output) {
```

```cpp
  if (combiner != kTfLiteCombinerTypeSum && num_elements > 0) {
    float multiplier = 1.0;
    switch (combiner) {
      case kTfLiteCombinerTypeMean:
        multiplier = current_total_weight;
        break;
      case kTfLiteCombinerTypeSqrtn:
        multiplier = std::sqrt(current_squares_weight);
        break;
      default:
        break;
    }
    for (int k = 0; k < embedding_size; k++) {
      output[k] /= multiplier;
    }
  }
}

TfLiteStatus Eval(TfLiteContext* context, TfLiteNode* node) {
  auto* params =
      reinterpret_cast<TfLiteEmbeddingLookupSparseParams*>(node->builtin_data);
  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context, GetOutputSafe(context, node, 0, &output));
  const TfLiteTensor* ids;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 0, &ids));
  const TfLiteTensor* indices;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 1, &indices));
  const TfLiteTensor* dense_shape;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 2, &dense_shape));
  const TfLiteTensor* weights;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 3, &weights));
  const TfLiteTensor* value;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 4, &value));

  const int lookup_rank = SizeOfDimension(indices, 1);
  const int embedding_rank = NumDimensions(value);
  const int num_lookups = SizeOfDimension(ids, 0);
  const int num_rows = SizeOfDimension(value, 0);

  // The last dimension gets replaced by the embedding.
  const int output_rank = (lookup_rank - 1) + (embedding_rank - 1);

  // Make sure that the actual dense shape of the sparse tensor represented by
  // (loopkup, indices, dense_shape) is consistent.
  TF_LITE_ENSURE_EQ(context, SizeOfDimension(dense_shape, 0), lookup_rank);

  // Resize output tensor.
  TfLiteIntArray* output_shape = TfLiteIntArrayCreate(output_rank);
  TF_LITE_ENSURE(context, output_shape != nullptr);
  int k = 0;
  int embedding_size = 1;
  int lookup_size = 1;
  for (int i = 0; i < lookup_rank - 1; i++, k++) {
    const int dim = dense_shape->data.i32[i];
    lookup_size *= dim;
    output_shape->data[k] = dim;
  }
  for (int i = 1; i < embedding_rank; i++, k++) {
    const int dim = SizeOfDimension(value, i);
    embedding_size *= dim;
    output_shape->data[k] = dim;
  }
  TF_LITE_ENSURE_STATUS(context->ResizeTensor(context, output, output_shape));
  const int output_size = lookup_size * embedding_size;
  TfLiteTensorRealloc(output_size * sizeof(float), output);

  float* output_ptr = GetTensorData<float>(output);
  const float* weights_ptr = GetTensorData<float>(weights);
  const float* value_ptr = GetTensorData<float>(value);

  std::fill_n(output_ptr, output_size, 0.0f);

  // Keep track of the current bucket for aggregation/combination.
  int current_output_offset = 0;
  float current_total_weight = 0.0;
  float current_squares_weight = 0.0;
  int num_elements = 0;

  for (int i = 0; i < num_lookups; i++) {
    int idx = ids->data.i32[i];
    if (idx >= num_rows || idx < 0) {
      context->ReportError(context,
                           "Embedding Lookup Sparse: index out of bounds. "
                           "Got %d, and bounds are [0, %d]",
                           idx, num_rows - 1);
      return kTfLiteError;
    }

    // Check where we need to aggregate.
    const int example_indices_offset = i * lookup_rank;
    int output_bucket = 0;
    int stride = 1;
    for (int k = (lookup_rank - 1) - 1; k >= 0; k--) {
      output_bucket += indices->data.i32[example_indices_offset + k] * stride;
      stride *= dense_shape->data.i32[k];
    }
    const int output_offset = output_bucket * embedding_size;

    // If we are in a new aggregation bucket and the combiner is not the sum,
    // go back and finalize the result of the previous bucket.
    if (output_offset != current_output_offset) {
      FinalizeAggregation(params->combiner, num_elements, current_total_weight,
                          current_squares_weight, embedding_size,
                          &output_ptr[current_output_offset]);

      // Track next bucket.
      num_elements = 0;
      current_total_weight = 0.0;
      current_squares_weight = 0.0;
      current_output_offset = output_offset;
    }

    // Add element to aggregation.
    ++num_elements;
    const int example_embedding_offset = idx * embedding_size;
    const float w = weights_ptr[i];
    current_squares_weight += w * w;
    current_total_weight += w;
    for (int k = 0; k < embedding_size; k++) {
      output_ptr[current_output_offset + k] +=
          value_ptr[example_embedding_offset + k] * w;
    }
  }

  // Finalize last bucket.
  FinalizeAggregation(params->combiner, num_elements, current_total_weight,
                      current_squares_weight, embedding_size,
                      &GetTensorData<float>(output)[current_output_offset]);

  return kTfLiteOk;
}

}  // namespace

TfLiteRegistration* Register_EMBEDDING_LOOKUP_SPARSE() {
  static TfLiteRegistration r = {nullptr, nullptr, Prepare, Eval};
  return &r;
}

}  // namespace builtin
}  // namespace ops
}  // namespace tflite
```