tensorflow / **tensorflow** Public

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

5100e359ae

tensorflow / tensorflow / core / kernels / **fractional_avg_pool_op.cc**

jpienaar Rename to underlying type rather than alias ... ✓ **History**

7 contributors

374 lines (333 sloc) | 15.8 KB

```
/* Copyright 2016 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.
==============================================================================*/
#define EIGEN_USE_THREADS

#include <algorithm>
#include <cmath>
#include <random>
#include <vector>

#include "tensorflow/core/kernels/fractional_pool_common.h"

#include "third_party/eigen3/unsupported/Eigen/CXX11/Tensor"
#include "tensorflow/core/framework/numeric_op.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/lib/random/random.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/platform/mutex.h"
```

```cpp
#include "tensorflow/core/util/guarded_philox_random.h"

namespace tensorflow {
typedef Eigen::ThreadPoolDevice CPUDevice;

template <typename T>
class FractionalAvgPoolOp : public OpKernel {
 public:
  explicit FractionalAvgPoolOp(OpKernelConstruction* context)
      : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("pooling_ratio", &pooling_ratio_));
    OP_REQUIRES_OK(context, context->GetAttr("pseudo_random", &pseudo_random_));
    OP_REQUIRES_OK(context, context->GetAttr("overlapping", &overlapping_));
    OP_REQUIRES(context, pooling_ratio_.size() == 4,
                errors::InvalidArgument(
                    "pooling_ratio field must specify 4 dimensions"));
    OP_REQUIRES(
        context, pooling_ratio_[0] == 1 || pooling_ratio_[3] == 1,
        errors::Unimplemented("Fractional average pooling is not yet "
                              "supported on the batch nor channel dimension."));
    OP_REQUIRES_OK(context, context->GetAttr("deterministic", &deterministic_));
    OP_REQUIRES_OK(context, context->GetAttr("seed", &seed_));
    OP_REQUIRES_OK(context, context->GetAttr("seed2", &seed2_));
    if (deterministic_) {
      // If both seeds are not set when deterministic_ is true, force set seeds.
      if ((seed_ == 0) && (seed2_ == 0)) {
        seed_ = random::New64();
        seed2_ = random::New64();
      }
    } else {
      OP_REQUIRES(
          context, (seed_ == 0) && (seed2_ == 0),
          errors::InvalidArgument(
              "Both seed and seed2 should be 0 if deterministic is false."));
    }
  }

  void Compute(OpKernelContext* context) override {
    typedef Eigen::Map<const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        ConstEigenMatrixMap;
    typedef Eigen::Map<Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        EigenMatrixMap;

    constexpr int tensor_in_and_out_dims = 4;

    const Tensor& tensor_in = context->input(0);
    OP_REQUIRES(context, tensor_in.dims() == tensor_in_and_out_dims,
                errors::InvalidArgument("tensor_in must be 4-dimensional"));

```

```cpp
    std::vector<int> input_size(tensor_in_and_out_dims);
    std::vector<int> output_size(tensor_in_and_out_dims);
    for (int i = 0; i < tensor_in_and_out_dims; ++i) {
      input_size[i] = tensor_in.dim_size(i);
      OP_REQUIRES(
          context, pooling_ratio_[i] <= input_size[i],
          errors::InvalidArgument(
              "Pooling ratio cannot be bigger than input tensor dim size."));
    }
    // Output size.
    for (int i = 0; i < tensor_in_and_out_dims; ++i) {
      output_size[i] =
          static_cast<int>(std::floor(input_size[i] / pooling_ratio_[i]));
      DCHECK_GT(output_size[i], 0);
    }

    // Generate pooling sequence.
    std::vector<int64_t> row_cum_seq;
    std::vector<int64_t> col_cum_seq;
    GuardedPhiloxRandom generator;
    generator.Init(seed_, seed2_);
    row_cum_seq = GeneratePoolingSequence(input_size[1], output_size[1],
                                          &generator, pseudo_random_);
    col_cum_seq = GeneratePoolingSequence(input_size[2], output_size[2],
                                          &generator, pseudo_random_);

    // Prepare output.
    Tensor* output_tensor = nullptr;
    OP_REQUIRES_OK(context, context->allocate_output(
                                0,
                                TensorShape({output_size[0], output_size[1],
                                             output_size[2], output_size[3]}),
                                &output_tensor));
    Tensor* output_row_seq_tensor = nullptr;
    OP_REQUIRES_OK(
        context, context->allocate_output(
                     1, TensorShape({static_cast<int64_t>(row_cum_seq.size())}),
                     &output_row_seq_tensor));
    Tensor* output_col_seq_tensor = nullptr;
    OP_REQUIRES_OK(
        context, context->allocate_output(
                     2, TensorShape({static_cast<int64_t>(col_cum_seq.size())}),
                     &output_col_seq_tensor));

    ConstEigenMatrixMap in_mat(tensor_in.flat<T>().data(), input_size[3],
                               input_size[2] * input_size[1] * input_size[0]);

    EigenMatrixMap out_mat(output_tensor->flat<T>().data(), output_size[3],
                           output_size[2] * output_size[1] * output_size[0]);
```

```
    // out_count corresponds to number of elements in each pooling cell.
    Eigen::Matrix<T, Eigen::Dynamic, 1> out_count(out_mat.cols());

    // Initializes the output tensor and out_count with 0.
    out_mat.setZero();
    out_count.setZero();

    auto output_row_seq_flat = output_row_seq_tensor->flat<int64_t>();
    auto output_col_seq_flat = output_col_seq_tensor->flat<int64_t>();

    // Set output tensors.
    for (int i = 0; i < row_cum_seq.size(); ++i) {
      output_row_seq_flat(i) = row_cum_seq[i];
    }

    for (int i = 0; i < col_cum_seq.size(); ++i) {
      output_col_seq_flat(i) = col_cum_seq[i];
    }

    // For both input and output,
    // 0: batch
    // 1: row / row
    // 2: col / col
    // 3: depth / channel
    const int64_t row_max = input_size[1] - 1;
    const int64_t col_max = input_size[2] - 1;
    for (int64_t b = 0; b < input_size[0]; ++b) {
      // row sequence.
      for (int64_t hs = 0; hs < row_cum_seq.size() - 1; ++hs) {
        // row start and end.
        const int64_t row_start = row_cum_seq[hs];
        int64_t row_end =
            overlapping_ ? row_cum_seq[hs + 1] : row_cum_seq[hs + 1] - 1;
        row_end = std::min(row_end, row_max);

        // col sequence.
        for (int64_t ws = 0; ws < col_cum_seq.size() - 1; ++ws) {
          const int64_t out_offset =
              (b * output_size[1] + hs) * output_size[2] + ws;
          // col start and end.
          const int64_t col_start = col_cum_seq[ws];
          int64_t col_end =
              overlapping_ ? col_cum_seq[ws + 1] : col_cum_seq[ws + 1] - 1;
          col_end = std::min(col_end, col_max);
          for (int64_t h = row_start; h <= row_end; ++h) {
            for (int64_t w = col_start; w <= col_end; ++w) {
              const int64_t in_offset =
                  (b * input_size[1] + h) * input_size[2] + w;
              out_mat.col(out_offset) += in_mat.col(in_offset);
```

```
              out_count(out_offset)++;
            }
          }
        }
      }
    }
    DCHECK_GT(out_count.minCoeff(), 0);
    out_mat.array().rowwise() /= out_count.transpose().array();
  }

 private:
  bool deterministic_;
  int64_t seed_;
  int64_t seed2_;
  std::vector<float> pooling_ratio_;
  bool pseudo_random_;
  bool overlapping_;
};

#define REGISTER_FRACTIONALAVGPOOL(type)                                      \
  REGISTER_KERNEL_BUILDER(                                                    \
      Name("FractionalAvgPool").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      FractionalAvgPoolOp<type>)

REGISTER_FRACTIONALAVGPOOL(int32);
REGISTER_FRACTIONALAVGPOOL(int64_t);
REGISTER_FRACTIONALAVGPOOL(float);
REGISTER_FRACTIONALAVGPOOL(double);

#undef REGISTER_FRACTIONALAVGPOOL

template <class T>
class FractionalAvgPoolGradOp : public OpKernel {
 public:
  explicit FractionalAvgPoolGradOp(OpKernelConstruction* context)
      : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("overlapping", &overlapping_));
  }

  void Compute(OpKernelContext* context) override {
    // Here's the basic idea:
    // Batch and depth dimension are independent from row and col dimension. And
    // because FractionalAvgPool currently only support pooling along row and
    // col, we can basically think of this 4D tensor backpropagation as
    // operation of a series of 2D planes.
    //
    // For each element of a 'slice' (2D plane) of output_backprop, we need to
    // figure out its contributors when doing FractionalAvgPool operation. This
    // can be done based on row_pooling_sequence, col_pooling_seq and
```

```
    // overlapping.
    // Once we figure out the original contributors, we just need to evenly
    // divide the value of this element among these contributors.
    //
    // Internally, we divide the out_backprop tensor and store it in a temporary
    // tensor of double type. And cast it to the corresponding type.
    typedef Eigen::Map<const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        ConstEigenMatrixMap;
    typedef Eigen::Map<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>>
        EigenDoubleMatrixMap;

    // Grab the inputs.
    const Tensor& orig_input_tensor_shape = context->input(0);
    OP_REQUIRES(context,
                orig_input_tensor_shape.dims() == 1 &&
                    orig_input_tensor_shape.NumElements() == 4,
                errors::InvalidArgument("original input tensor shape must be"
                                        "1-dimensional and 4 elements"));
    const Tensor& out_backprop = context->input(1);
    const Tensor& row_seq_tensor = context->input(2);
    const Tensor& col_seq_tensor = context->input(3);

    const int64_t out_batch = out_backprop.dim_size(0);
    const int64_t out_rows = out_backprop.dim_size(1);
    const int64_t out_cols = out_backprop.dim_size(2);
    const int64_t out_depth = out_backprop.dim_size(3);

    OP_REQUIRES(context, row_seq_tensor.NumElements() > out_rows,
                errors::InvalidArgument("Given out_backprop shape ",
                                        out_backprop.shape().DebugString(),
                                        ", row_seq_tensor must have at least ",
                                        out_rows + 1, " elements, but got ",
                                        row_seq_tensor.NumElements()));
    OP_REQUIRES(context, col_seq_tensor.NumElements() > out_cols,
                errors::InvalidArgument("Given out_backprop shape ",
                                        out_backprop.shape().DebugString(),
                                        ", col_seq_tensor must have at least ",
                                        out_cols + 1, " elements, but got ",
                                        col_seq_tensor.NumElements()));

    auto row_seq_tensor_flat = row_seq_tensor.flat<int64_t>();
    auto col_seq_tensor_flat = col_seq_tensor.flat<int64_t>();
    auto orig_input_tensor_shape_flat = orig_input_tensor_shape.flat<int64_t>();

    const int64_t in_batch = orig_input_tensor_shape_flat(0);
    const int64_t in_rows = orig_input_tensor_shape_flat(1);
    const int64_t in_cols = orig_input_tensor_shape_flat(2);
    const int64_t in_depth = orig_input_tensor_shape_flat(3);
    OP_REQUIRES(
```

```
        context, in_batch != 0,
        errors::InvalidArgument("Batch dimension of input must not be 0"));
    OP_REQUIRES(
        context, in_rows != 0,
        errors::InvalidArgument("Rows dimension of input must not be 0"));
    OP_REQUIRES(
        context, in_cols != 0,
        errors::InvalidArgument("Columns dimension of input must not be 0"));
    OP_REQUIRES(
        context, in_depth != 0,
        errors::InvalidArgument("Depth dimension of input must not be 0"));

    constexpr int tensor_in_and_out_dims = 4;
    // Transform orig_input_tensor_shape into TensorShape
    TensorShape in_shape;
    for (auto i = 0; i < tensor_in_and_out_dims; ++i) {
      in_shape.AddDim(orig_input_tensor_shape_flat(i));
    }

    // Create intermediate in_backprop.
    Tensor in_backprop_tensor_temp;
    OP_REQUIRES_OK(context, context->forward_input_or_allocate_temp(
                                {0}, DataTypeToEnum<double>::v(), in_shape,
                                &in_backprop_tensor_temp));
    in_backprop_tensor_temp.flat<double>().setZero();
    // Transform 4D tensor to 2D matrix.
    EigenDoubleMatrixMap in_backprop_tensor_temp_mat(
        in_backprop_tensor_temp.flat<double>().data(), in_depth,
        in_cols * in_rows * in_batch);
    ConstEigenMatrixMap out_backprop_mat(out_backprop.flat<T>().data(),
                                         out_depth,
                                         out_cols * out_rows * out_batch);
    // Loop through each element of out_backprop and evenly distribute the
    // element to the corresponding pooling cell.
    const int64_t in_max_row_index = in_rows - 1;
    const int64_t in_max_col_index = in_cols - 1;
    for (int64_t b = 0; b < out_batch; ++b) {
      for (int64_t r = 0; r < out_rows; ++r) {
        const int64_t in_row_start = row_seq_tensor_flat(r);
        int64_t in_row_end = overlapping_ ? row_seq_tensor_flat(r + 1)
                                          : row_seq_tensor_flat(r + 1) - 1;
        in_row_end = std::min(in_row_end, in_max_row_index);
        for (int64_t c = 0; c < out_cols; ++c) {
          const int64_t in_col_start = col_seq_tensor_flat(c);
          int64_t in_col_end = overlapping_ ? col_seq_tensor_flat(c + 1)
                                            : col_seq_tensor_flat(c + 1) - 1;
          in_col_end = std::min(in_col_end, in_max_col_index);

          const int64_t num_elements_in_pooling_cell =
```

```
                (in_row_end - in_row_start + 1) * (in_col_end - in_col_start + 1);
            const int64_t out_index = (b * out_rows + r) * out_cols + c;
            // Now we can evenly distribute out_backprop(b, h, w, *) to
            // in_backprop(b, hs:he, ws:we, *).
            for (int64_t in_r = in_row_start; in_r <= in_row_end; ++in_r) {
              for (int64_t in_c = in_col_start; in_c <= in_col_end; ++in_c) {
                const int64_t in_index = (b * in_rows + in_r) * in_cols + in_c;
                // Walk through each channel (depth).
                for (int64_t d = 0; d < out_depth; ++d) {
                  const double out_backprop_element = static_cast<double>(
                      out_backprop_mat.coeffRef(d, out_index));
                  double& in_backprop_ref =
                      in_backprop_tensor_temp_mat.coeffRef(d, in_index);
                  in_backprop_ref +=
                      out_backprop_element / num_elements_in_pooling_cell;
                }
              }
            }
          }
        }
      }

      // Depending on the type, cast double to type T.
      Tensor* in_backprop_tensor = nullptr;
      OP_REQUIRES_OK(context, context->forward_input_or_allocate_output(
                                  {0}, 0, in_shape, &in_backprop_tensor));
      auto in_backprop_tensor_flat = in_backprop_tensor->flat<T>();
      auto in_backprop_tensor_temp_flat = in_backprop_tensor_temp.flat<double>();
      for (int64_t i = 0; i < in_backprop_tensor_flat.size(); ++i) {
        in_backprop_tensor_flat(i) =
            static_cast<T>(in_backprop_tensor_temp_flat(i));
      }
    }

   private:
    bool overlapping_;
  };

  #define REGISTER_FRACTIONALAVGPOOLGRAD(type)              \
    REGISTER_KERNEL_BUILDER(Name("FractionalAvgPoolGrad")   \
                                .Device(DEVICE_CPU)         \
                                .TypeConstraint<type>("T"), \
                            FractionalAvgPoolGradOp<type>)

  REGISTER_FRACTIONALAVGPOOLGRAD(int32);
  REGISTER_FRACTIONALAVGPOOLGRAD(int64_t);
  REGISTER_FRACTIONALAVGPOOLGRAD(float);
  REGISTER_FRACTIONALAVGPOOLGRAD(double);

```

```
#undef REGISTER_FRACTIONALAVGPOOLGRAD
}  // namespace tensorflow
```