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

3 contributors 

384 lines (332 sloc) | 14.3 KB ...

```
1  /* Copyright 2020 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 ==============================================================================*/
15
16 #include "absl/container/flat_hash_map.h"
17 #include "tensorflow/core/framework/op_kernel.h"
18 #include "tensorflow/core/framework/op_requires.h"
19 #include "tensorflow/core/framework/register_types.h"
20 #include "tensorflow/core/framework/tensor.h"
21 #include "tensorflow/core/platform/errors.h"
22 #include "tensorflow/core/platform/types.h"
23
24 namespace tensorflow {
25
26 template <class T>
27 using BatchedMap = std::vector<absl::flat_hash_map<int64_t, T>>;
28
29 namespace {
```

```
// TODO(momernick): Extend this function to work with outputs of rank > 2.
template <class T>
Status OutputSparse(const BatchedMap<T>& per_batch_counts, int num_values,
                    bool is_1d, OpKernelContext* context) {
  int total_values = 0;
  int num_batches = per_batch_counts.size();
  for (const auto& per_batch_count : per_batch_counts) {
    total_values += per_batch_count.size();
  }

  Tensor* indices;
  int inner_dim = is_1d ? 1 : 2;
  TF_RETURN_IF_ERROR(context->allocate_output(
      0, TensorShape({total_values, inner_dim}), &indices));

  Tensor* values;
  TF_RETURN_IF_ERROR(
      context->allocate_output(1, TensorShape({total_values}), &values));

  auto output_indices = indices->matrix<int64_t>();
  auto output_values = values->flat<T>();
  int64_t value_loc = 0;
  for (int b = 0; b < num_batches; ++b) {
    const auto& per_batch_count = per_batch_counts[b];
    std::vector<std::pair<int, T>> pairs(per_batch_count.begin(),
                                         per_batch_count.end());
    std::sort(pairs.begin(), pairs.end());
    for (const auto& x : pairs) {
      if (is_1d) {
        output_indices(value_loc, 0) = x.first;
      } else {
        output_indices(value_loc, 0) = b;
        output_indices(value_loc, 1) = x.first;
      }
      output_values(value_loc) = x.second;
      ++value_loc;
    }
  }
  Tensor* dense_shape;
  if (is_1d) {
    TF_RETURN_IF_ERROR(
        context->allocate_output(2, TensorShape({1}), &dense_shape));
    dense_shape->flat<int64_t>().data()[0] = num_values;
  } else {
    TF_RETURN_IF_ERROR(
        context->allocate_output(2, TensorShape({2}), &dense_shape));
    dense_shape->flat<int64_t>().data()[0] = num_batches;
    dense_shape->flat<int64_t>().data()[1] = num_values;
  }
```

```
  return Status::OK();
}

int GetOutputSize(int max_seen, int max_length, int min_length) {
  return max_length > 0 ? max_length : std::max((max_seen + 1), min_length);
}

}  // namespace

template <class T, class W>
class DenseCount : public OpKernel {
 public:
  explicit DenseCount(OpKernelConstruction* context) : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
    OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
    OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
  }

  void Compute(OpKernelContext* context) override {
    const Tensor& data = context->input(0);
    const Tensor& weights = context->input(1);
    bool use_weights = weights.NumElements() > 0;

    OP_REQUIRES(context,
                TensorShapeUtils::IsVector(data.shape()) ||
                    TensorShapeUtils::IsMatrix(data.shape()),
                errors::InvalidArgument(
                    "Input must be a 1 or 2-dimensional tensor. Got: ",
                    data.shape().DebugString()));

    if (use_weights) {
      OP_REQUIRES(
          context, weights.shape() == data.shape(),
          errors::InvalidArgument(
              "Weights and data must have the same shape. Weight shape: ",
              weights.shape().DebugString(),
              "; data shape: ", data.shape().DebugString()));
    }

    bool is_1d = TensorShapeUtils::IsVector(data.shape());
    int negative_valued_axis = -1;
    int num_batch_dimensions = (data.shape().dims() + negative_valued_axis);

    int num_batch_elements = 1;
    for (int i = 0; i < num_batch_dimensions; ++i) {
      OP_REQUIRES(context, data.shape().dim_size(i) != 0,
                  errors::InvalidArgument(
                      "Invalid input: Shapes dimension cannot be 0."));
```

```
      num_batch_elements *= data.shape().dim_size(i);
    }
    int num_value_elements = data.shape().num_elements() / num_batch_elements;
    auto per_batch_counts = BatchedMap<W>(num_batch_elements);

    T max_value = 0;

    const auto data_values = data.flat<T>();
    const auto weight_values = weights.flat<W>();
    int i = 0;
    for (int b = 0; b < num_batch_elements; ++b) {
      for (int v = 0; v < num_value_elements; ++v) {
        const auto& value = data_values(i);
        if (value >= 0 && (maxlength_ <= 0 || value < maxlength_)) {
          if (binary_output_) {
            per_batch_counts[b][value] = 1;
          } else if (use_weights) {
            per_batch_counts[b][value] += weight_values(i);
          } else {
            per_batch_counts[b][value]++;
          }
          if (value > max_value) {
            max_value = value;
          }
        }
        ++i;
      }
    }

    int num_output_values = GetOutputSize(max_value, maxlength_, minlength_);
    OP_REQUIRES_OK(context, OutputSparse<W>(per_batch_counts, num_output_values,
                                            is_1d, context));
  }

 private:
  int maxlength_;
  int minlength_;
  bool binary_output_;
};

template <class T, class W>
class SparseCount : public OpKernel {
 public:
  explicit SparseCount(OpKernelConstruction* context) : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
    OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
    OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
  }

```

```
177   void Compute(OpKernelContext* context) override {
178     const Tensor& indices = context->input(0);
179     const Tensor& values = context->input(1);
180     const Tensor& shape = context->input(2);
181     const Tensor& weights = context->input(3);
182     bool use_weights = weights.NumElements() > 0;
183
184     OP_REQUIRES(context, TensorShapeUtils::IsMatrix(indices.shape()),
185                 errors::InvalidArgument(
186                     "Input indices must be a 2-dimensional tensor. Got: ",
187                     indices.shape().DebugString()));
188
189     if (use_weights) {
190       OP_REQUIRES(
191           context, weights.shape() == values.shape(),
192           errors::InvalidArgument(
193               "Weights and values must have the same shape. Weight shape: ",
194               weights.shape().DebugString(),
195               "; values shape: ", values.shape().DebugString()));
196     }
197
198     OP_REQUIRES(context, shape.NumElements() != 0,
199                 errors::InvalidArgument(
200                     "The shape argument requires at least one element."));
201
202     bool is_1d = shape.NumElements() == 1;
203     auto shape_vector = shape.flat<int64_t>();
204     int num_batches = is_1d ? 1 : shape_vector(0);
205     int num_values = values.NumElements();
206
207     for (int b = 0; b < shape_vector.size(); b++) {
208       OP_REQUIRES(context, shape_vector(b) >= 0,
209                   errors::InvalidArgument(
210                       "Elements in dense_shape must be >= 0. Instead got:",
211                       shape.DebugString()));
212     }
213
214     OP_REQUIRES(context, num_values == indices.shape().dim_size(0),
215                 errors::InvalidArgument(
216                     "Number of values must match first dimension of indices.",
217                     "Got ", num_values,
218                     " values, indices shape: ", indices.shape().DebugString()));
219
220     const auto indices_values = indices.matrix<int64_t>();
221     const auto values_values = values.flat<T>();
222     const auto weight_values = weights.flat<W>();
223
224     auto per_batch_counts = BatchedMap<W>(num_batches);
225
```

```
226     T max_value = 0;
227
228     OP_REQUIRES(context, num_values <= indices.shape().dim_size(0),
229                 errors::InvalidArgument(
230                     "The first dimension of indices must be equal to or "
231                     "greather than number of values. ( ",
232                     indices.shape().dim_size(0), " vs. ", num_values, " )"));
233     OP_REQUIRES(context, indices.shape().dim_size(1) > 0,
234                 errors::InvalidArgument("The second dimension of indices must "
235                                         "be greater than 0. Received: ",
236                                         indices.shape().dim_size(1)));
237
238     for (int idx = 0; idx < num_values; ++idx) {
239       int batch = is_1d ? 0 : indices_values(idx, 0);
240       if (batch >= num_batches) {
241         OP_REQUIRES(context, batch < num_batches,
242                     errors::InvalidArgument(
243                         "Indices value along the first dimension must be ",
244                         "lower than the first index of the shape.", "Got ",
245                         batch, " as batch and ", num_batches,
246                         " as the first dimension of the shape."));
247       }
248       const auto& value = values_values(idx);
249       if (value >= 0 && (maxlength_ <= 0 || value < maxlength_)) {
250         if (binary_output_) {
251           per_batch_counts[batch][value] = 1;
252         } else if (use_weights) {
253           per_batch_counts[batch][value] += weight_values(idx);
254         } else {
255           per_batch_counts[batch][value]++;
256         }
257         if (value > max_value) {
258           max_value = value;
259         }
260       }
261     }
262
263     int num_output_values = GetOutputSize(max_value, maxlength_, minlength_);
264     OP_REQUIRES_OK(context, OutputSparse<W>(per_batch_counts, num_output_values,
265                                             is_1d, context));
266   }
267
268  private:
269   int maxlength_;
270   int minlength_;
271   bool binary_output_;
272   bool validate_;
273 };
274
```

```
template <class T, class W>
class RaggedCount : public OpKernel {
 public:
  explicit RaggedCount(OpKernelConstruction* context) : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
    OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
    OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
  }

  void Compute(OpKernelContext* context) override {
    const Tensor& splits = context->input(0);
    const Tensor& values = context->input(1);
    const Tensor& weights = context->input(2);
    bool use_weights = weights.NumElements() > 0;
    bool is_1d = false;

    if (use_weights) {
      OP_REQUIRES(
          context, weights.shape() == values.shape(),
          errors::InvalidArgument(
              "Weights and values must have the same shape. Weight shape: ",
              weights.shape().DebugString(),
              "; values shape: ", values.shape().DebugString()));
    }

    const auto splits_values = splits.flat<int64_t>();
    const auto values_values = values.flat<T>();
    const auto weight_values = weights.flat<W>();
    int num_batches = splits.NumElements() - 1;
    int num_values = values.NumElements();

    OP_REQUIRES(
        context, num_batches > 0,
        errors::InvalidArgument(
            "Must provide at least 2 elements for the splits argument"));
    OP_REQUIRES(context, splits_values(0) == 0,
                errors::InvalidArgument("Splits must start with 0, not with ",
                                        splits_values(0)));
    OP_REQUIRES(context, splits_values(num_batches) == num_values,
                errors::InvalidArgument(
                    "Splits must end with the number of values, got ",
                    splits_values(num_batches), " instead of ", num_values));

    auto per_batch_counts = BatchedMap<W>(num_batches);
    T max_value = 0;
    int batch_idx = 0;

    for (int idx = 0; idx < num_values; ++idx) {
      while (idx >= splits_values(batch_idx)) {
```

```
        batch_idx++;
      }
      const auto& value = values_values(idx);
      if (value >= 0 && (maxlength_ <= 0 || value < maxlength_)) {
        if (binary_output_) {
          per_batch_counts[batch_idx - 1][value] = 1;
        } else if (use_weights) {
          per_batch_counts[batch_idx - 1][value] += weight_values(idx);
        } else {
          per_batch_counts[batch_idx - 1][value]++;
        }
        if (value > max_value) {
          max_value = value;
        }
      }
    }

    int num_output_values = GetOutputSize(max_value, maxlength_, minlength_);
    OP_REQUIRES_OK(context, OutputSparse<W>(per_batch_counts, num_output_values,
                                            is_1d, context));
  }

 private:
  int maxlength_;
  int minlength_;
  bool binary_output_;
  bool validate_;
};

#define REGISTER_W(W_TYPE) \
  REGISTER(int32, W_TYPE)  \
  REGISTER(int64_t, W_TYPE)

#define REGISTER(I_TYPE, W_TYPE)                                    \
                                                                    \
  REGISTER_KERNEL_BUILDER(Name("DenseCountSparseOutput")            \
                              .TypeConstraint<I_TYPE>("T")          \
                              .TypeConstraint<W_TYPE>("output_type") \
                              .Device(DEVICE_CPU),                  \
                          DenseCount<I_TYPE, W_TYPE>)               \
                                                                    \
  REGISTER_KERNEL_BUILDER(Name("SparseCountSparseOutput")           \
                              .TypeConstraint<I_TYPE>("T")          \
                              .TypeConstraint<W_TYPE>("output_type") \
                              .Device(DEVICE_CPU),                  \
                          SparseCount<I_TYPE, W_TYPE>)              \
                                                                    \
  REGISTER_KERNEL_BUILDER(Name("RaggedCountSparseOutput")           \
                              .TypeConstraint<I_TYPE>("T")          \
```

```
                                .TypeConstraint<W_TYPE>("output_type") \
                                .Device(DEVICE_CPU),                   \
                            RaggedCount<I_TYPE, W_TYPE>)

TF_CALL_INTEGRAL_TYPES(REGISTER_W);
TF_CALL_float(REGISTER_W);
TF_CALL_double(REGISTER_W);

#undef REGISTER_W
#undef REGISTER

}  // namespace tensorflow
```