tensorflower-gardener [tf.data] Change Cardinality() implementation to read c... ... History

8 contributors

305 lines (266 sloc) | 11.8 KB ...

```
1  /* Copyright 2017 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 #include <numeric>
16
17 #include "tensorflow/core/framework/dataset.h"
18 #include "tensorflow/core/framework/partial_tensor_shape.h"
19 #include "tensorflow/core/framework/register_types.h"
20 #include "tensorflow/core/framework/tensor.h"
21 #include "tensorflow/core/util/sparse/sparse_tensor.h"
22
23 namespace tensorflow {
24 namespace data {
25 namespace {
26
27 // See documentation in ../../ops/dataset_ops.cc for a high-level
28 // description of the following op.
29
```

```
template <typename T>
class Dataset : public DatasetBase {
 public:
  explicit Dataset(OpKernelContext* ctx,
                   const sparse::SparseTensor& sparse_tensor)
      : DatasetBase(DatasetContext(ctx)),
        sparse_tensor_(sparse_tensor),
        dtypes_({DT_INT64, sparse_tensor.dtype(), DT_INT64}),
        shapes_({{-1, sparse_tensor.dims() - 1},
                 {-1},
                 {sparse_tensor.dims() - 1}}) {}

  std::unique_ptr<IteratorBase> MakeIteratorInternal(
      const string& prefix) const override {
    return absl::make_unique<Iterator>(typename Iterator::Params{
        this, strings::StrCat(prefix, "::SparseTensorSlice")});
  }

  const DataTypeVector& output_dtypes() const override { return dtypes_; }
  const std::vector<PartialTensorShape>& output_shapes() const override {
    return shapes_;
  }

  string DebugString() const override {
    return "SparseTensorSliceDatasetOp::Dataset";
  }

  int64_t CardinalityInternal() const override {
    return sparse_tensor_.shape()[0];
  }

  Status InputDatasets(std::vector<const DatasetBase*>* inputs) const override {
    return Status::OK();
  }

  Status CheckExternalState() const override { return Status::OK(); }

 protected:
  Status AsGraphDefInternal(SerializationContext* ctx,
                            DatasetGraphDefBuilder* b,
                            Node** output) const override {
    Node* indices_node;
    TF_RETURN_IF_ERROR(b->AddTensor(sparse_tensor_.indices(), &indices_node));
    Node* value_node;
    TF_RETURN_IF_ERROR(b->AddTensor(sparse_tensor_.values(), &value_node));
    Node* dense_shape_node;
    std::vector<int64_t> dense_shape;
    dense_shape.reserve(sparse_tensor_.shape().size());
    for (int i = 0; i < sparse_tensor_.shape().size(); i++)
```

```
      dense_shape.emplace_back(sparse_tensor_.shape()[i]);
    TF_RETURN_IF_ERROR(b->AddVector(dense_shape, &dense_shape_node));
    AttrValue val_dtype;
    b->BuildAttrValue(sparse_tensor_.dtype(), &val_dtype);
    TF_RETURN_IF_ERROR(
        b->AddDataset(this, {indices_node, value_node, dense_shape_node},
                      {{"Tvalues", val_dtype}}, output));
    return Status::OK();
  }

 private:
  class Iterator : public DatasetIterator<Dataset<T>> {
   public:
    explicit Iterator(const typename Iterator::Params& params)
        : DatasetIterator<Dataset<T>>(params),
          num_elements_(params.dataset->sparse_tensor_.shape()[0]),
          dense_shape_(DT_INT64, {params.dataset->sparse_tensor_.dims() - 1}),
          group_iterable_(params.dataset->sparse_tensor_.group({0})),
          iter_(group_iterable_.begin()) {
      for (size_t i = 0; i < dense_shape_.NumElements(); ++i) {
        dense_shape_.vec<int64_t>()(i) =
            params.dataset->sparse_tensor_.shape()[i + 1];
      }
    }

    Status GetNextInternal(IteratorContext* ctx,
                           std::vector<Tensor>* out_tensors,
                           bool* end_of_sequence) override {
      mutex_lock l(mu_);
      if (i_ == num_elements_) {
        *end_of_sequence = true;
        return Status::OK();
      }

      out_tensors->clear();
      out_tensors->reserve(3);
      const int rank = Iterator::dataset()->sparse_tensor_.dims();

      if (i_ > next_non_empty_i_ && iter_ != group_iterable_.end()) {
        // We still have elements to consume from `group_iterable_`
        // and we have emitted all elements up to and including the
        // current position.
        sparse::Group group = *iter_;
        const auto indices = group.indices();
        const auto values = group.values<T>();
        const int64_t num_entries = values.size();
        next_non_empty_i_ = indices(0, 0);

        next_indices_ = Tensor(DT_INT64, {num_entries, rank - 1});
```

```
128         next_values_ = Tensor(DataTypeToEnum<T>::value, {num_entries});
129
130         auto next_indices_t = next_indices_.matrix<int64_t>();
131         auto next_values_t = next_values_.vec<T>();
132
133         for (int64_t i = 0; i < num_entries; ++i) {
134           for (int d = 1; d < rank; ++d) {
135             next_indices_t(i, d - 1) = indices(i, d);
136           }
137           next_values_t(i) = values(i);
138         }
139
140         ++iter_;
141       }
142       if (i_ == next_non_empty_i_) {
143         // The current position is non-empty in the input
144         // `SparseTensor`, and we have already read the value from the
145         // `GroupIterable`.
146         out_tensors->push_back(std::move(next_indices_));
147         out_tensors->push_back(std::move(next_values_));
148         out_tensors->push_back(dense_shape_);
149         next_non_empty_i_ = kNextNonEmptyUnknown;
150       } else {
151         DCHECK(i_ < next_non_empty_i_ || iter_ == group_iterable_.end());
152         // The current position is empty in the input `SparseTensor`,
153         // so emit empty indices and values.
154         out_tensors->push_back(Tensor(DT_INT64, TensorShape({0, rank - 1})));
155         out_tensors->push_back(Tensor(DataTypeToEnum<T>::value, {0}));
156         out_tensors->push_back(dense_shape_);
157       }
158
159       ++i_;
160       *end_of_sequence = false;
161       return Status::OK();
162     }
163
164   protected:
165     std::shared_ptr<model::Node> CreateNode(
166         IteratorContext* ctx, model::Node::Args args) const override {
167       return model::MakeSourceNode(std::move(args));
168     }
169
170     Status SaveInternal(SerializationContext* ctx,
171                         IteratorStateWriter* writer) override {
172       mutex_lock l(mu_);
173       TF_RETURN_IF_ERROR(writer->WriteScalar(Iterator::full_name("i"), i_));
174       TF_RETURN_IF_ERROR(
175           writer->WriteScalar(Iterator::full_name("iter_loc"), iter_.loc()));
176       TF_RETURN_IF_ERROR(writer->WriteScalar(
```

```
177           Iterator::full_name("next_non_empty_i_"), next_non_empty_i_));
178       if (i_ <= next_non_empty_i_) {
179         TF_RETURN_IF_ERROR(writer->WriteTensor(
180             Iterator::full_name("next_indices_"), next_indices_));
181         TF_RETURN_IF_ERROR(writer->WriteTensor(
182             Iterator::full_name("next_values_"), next_values_));
183       }
184       return Status::OK();
185     }
186
187     Status RestoreInternal(IteratorContext* ctx,
188                            IteratorStateReader* reader) override {
189       mutex_lock l(mu_);
190       TF_RETURN_IF_ERROR(reader->ReadScalar(Iterator::full_name("i"), &i_));
191       int64_t iter_loc;
192       TF_RETURN_IF_ERROR(
193           reader->ReadScalar(Iterator::full_name("iter_loc"), &iter_loc));
194       iter_ = group_iterable_.at(iter_loc);
195       TF_RETURN_IF_ERROR(reader->ReadScalar(
196           Iterator::full_name("next_non_empty_i_"), &next_non_empty_i_));
197       if (i_ <= next_non_empty_i_) {
198         TF_RETURN_IF_ERROR(reader->ReadTensor(
199             Iterator::full_name("next_indices_"), &next_indices_));
200         TF_RETURN_IF_ERROR(reader->ReadTensor(
201             Iterator::full_name("next_values_"), &next_values_));
202       }
203       return Status::OK();
204     }
205
206    private:
207     const int64_t num_elements_;
208
209     Tensor dense_shape_;
210
211     mutex mu_;
212     sparse::GroupIterable group_iterable_ TF_GUARDED_BY(mu_);
213     sparse::GroupIterable::IteratorStep iter_ TF_GUARDED_BY(mu_);
214     int64_t i_ TF_GUARDED_BY(mu_) = 0;
215     const int64_t kNextNonEmptyUnknown = -1;
216     int64_t next_non_empty_i_ TF_GUARDED_BY(mu_) = kNextNonEmptyUnknown;
217     Tensor next_indices_ TF_GUARDED_BY(mu_);
218     Tensor next_values_ TF_GUARDED_BY(mu_);
219   };
220
221   const sparse::SparseTensor sparse_tensor_;
222   const DataTypeVector dtypes_;
223   const std::vector<PartialTensorShape> shapes_;
224 };
225
```

```
template <typename T>
class SparseTensorSliceDatasetOp : public DatasetOpKernel {
 public:
  explicit SparseTensorSliceDatasetOp(OpKernelConstruction* ctx)
      : DatasetOpKernel(ctx) {}

  void MakeDataset(OpKernelContext* ctx, DatasetBase** output) override {
    // Create a new SparseTensorSliceDatasetOp::Dataset, insert it in
    // the step container, and return it as the output.
    const Tensor* indices;
    OP_REQUIRES_OK(ctx, ctx->input("indices", &indices));
    const Tensor* values;
    OP_REQUIRES_OK(ctx, ctx->input("values", &values));
    const Tensor* dense_shape;
    OP_REQUIRES_OK(ctx, ctx->input("dense_shape", &dense_shape));

    OP_REQUIRES(ctx, TensorShapeUtils::IsMatrix(indices->shape()),
                errors::InvalidArgument(
                    "Input indices should be a matrix but received shape ",
                    indices->shape().DebugString()));

    const auto num_indices = indices->NumElements();
    const auto num_values = values->NumElements();
    if (num_indices == 0 || num_values == 0) {
      OP_REQUIRES(ctx, num_indices == num_values,
                  errors::InvalidArgument(
                      "If indices or values are empty, the other one must also "
                      "be. Got indices of shape ",
                      indices->shape().DebugString(), " and values of shape ",
                      values->shape().DebugString()));
    }
    OP_REQUIRES(ctx, TensorShapeUtils::IsVector(values->shape()),
                errors::InvalidArgument(
                    "Input values should be a vector but received shape ",
                    indices->shape().DebugString()));
    OP_REQUIRES(ctx, TensorShapeUtils::IsVector(dense_shape->shape()),
                errors::InvalidArgument(
                    "Input shape should be a vector but received shape ",
                    dense_shape->shape().DebugString()));

    // We currently ensure that `sparse_tensor` is ordered in the
    // batch dimension.
    // TODO(mrry): Investigate ways to avoid this unconditional check
    // if we can be sure that the sparse tensor was produced in an
    // appropriate order (e.g. by `tf.parse_example()` or a Dataset
    // that batches elements into rows of a SparseTensor).
    int64_t previous_batch_index = -1;
    for (int64_t i = 0; i < indices->dim_size(0); ++i) {
      int64_t next_batch_index = indices->matrix<int64_t>()(i, 0);
```

```
      OP_REQUIRES(
          ctx, next_batch_index >= previous_batch_index,
          errors::Unimplemented("The SparseTensor must be ordered in the batch "
                                "dimension; handling arbitrarily ordered input "
                                "is not currently supported."));
      previous_batch_index = next_batch_index;
    }
    gtl::InlinedVector<int64_t, 8> std_order(dense_shape->NumElements(), 0);
    sparse::SparseTensor tensor;
    OP_REQUIRES_OK(
        ctx, sparse::SparseTensor::Create(
                 *indices, *values, TensorShape(dense_shape->vec<int64_t>()),
                 std_order, &tensor));
    *output = new Dataset<T>(ctx, std::move(tensor));
  }

 private:
};

#define REGISTER_DATASET_KERNEL(type)                           \
  REGISTER_KERNEL_BUILDER(Name("SparseTensorSliceDataset")      \
                              .Device(DEVICE_CPU)               \
                              .TypeConstraint<type>("Tvalues"), \
                          SparseTensorSliceDatasetOp<type>);

TF_CALL_DATASET_TYPES(REGISTER_DATASET_KERNEL);
#undef REGISTER_DATASET_KERNEL

}  // namespace
}  // namespace data
}  // namespace tensorflow
```