

5100e359ae ▾

...

tensorflow / tensorflow / core / kernels / count\_ops.cc

 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 {
```

```

30 // TODO(momernick): Extend this function to work with outputs of rank > 2.
31 template <class T>
32 Status OutputSparse(const BatchedMap<T>& per_batch_counts, int num_values,
33                    bool is_1d, OpKernelContext* context) {
34     int total_values = 0;
35     int num_batches = per_batch_counts.size();
36     for (const auto& per_batch_count : per_batch_counts) {
37         total_values += per_batch_count.size();
38     }
39
40     Tensor* indices;
41     int inner_dim = is_1d ? 1 : 2;
42     TF_RETURN_IF_ERROR(context->allocate_output(
43         0, TensorShape({total_values, inner_dim}), &indices));
44
45     Tensor* values;
46     TF_RETURN_IF_ERROR(
47         context->allocate_output(1, TensorShape({total_values}), &values));
48
49     auto output_indices = indices->matrix<int64_t>();
50     auto output_values = values->flat<T>();
51     int64_t value_loc = 0;
52     for (int b = 0; b < num_batches; ++b) {
53         const auto& per_batch_count = per_batch_counts[b];
54         std::vector<std::pair<int, T>> pairs(per_batch_count.begin(),
55                                           per_batch_count.end());
56         std::sort(pairs.begin(), pairs.end());
57         for (const auto& x : pairs) {
58             if (is_1d) {
59                 output_indices(value_loc, 0) = x.first;
60             } else {
61                 output_indices(value_loc, 0) = b;
62                 output_indices(value_loc, 1) = x.first;
63             }
64             output_values(value_loc) = x.second;
65             ++value_loc;
66         }
67     }
68     Tensor* dense_shape;
69     if (is_1d) {
70         TF_RETURN_IF_ERROR(
71             context->allocate_output(2, TensorShape({1}), &dense_shape));
72         dense_shape->flat<int64_t>().data()[0] = num_values;
73     } else {
74         TF_RETURN_IF_ERROR(
75             context->allocate_output(2, TensorShape({2}), &dense_shape));
76         dense_shape->flat<int64_t>().data()[0] = num_batches;
77         dense_shape->flat<int64_t>().data()[1] = num_values;
78     }

```

```

79
80     return Status::OK();
81 }
82
83 int GetOutputSize(int max_seen, int max_length, int min_length) {
84     return max_length > 0 ? max_length : std::max((max_seen + 1), min_length);
85 }
86
87 } // namespace
88
89 template <class T, class W>
90 class DenseCount : public OpKernel {
91 public:
92     explicit DenseCount(OpKernelConstruction* context) : OpKernel(context) {
93         OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
94         OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
95         OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
96     }
97
98     void Compute(OpKernelContext* context) override {
99         const Tensor& data = context->input(0);
100         const Tensor& weights = context->input(1);
101         bool use_weights = weights.NumElements() > 0;
102
103         OP_REQUIRES(context,
104             TensorShapeUtils::IsVector(data.shape()) ||
105             TensorShapeUtils::IsMatrix(data.shape()),
106             errors::InvalidArgument(
107                 "Input must be a 1 or 2-dimensional tensor. Got: ",
108                 data.shape().DebugString()));
109
110         if (use_weights) {
111             OP_REQUIRES(
112                 context, weights.shape() == data.shape(),
113                 errors::InvalidArgument(
114                     "Weights and data must have the same shape. Weight shape: ",
115                     weights.shape().DebugString(),
116                     "; data shape: ", data.shape().DebugString()));
117         }
118
119         bool is_1d = TensorShapeUtils::IsVector(data.shape());
120         int negative_valued_axis = -1;
121         int num_batch_dimensions = (data.shape().dims() + negative_valued_axis);
122
123         int num_batch_elements = 1;
124         for (int i = 0; i < num_batch_dimensions; ++i) {
125             OP_REQUIRES(context, data.shape().dim_size(i) != 0,
126                 errors::InvalidArgument(
127                     "Invalid input: Shapes dimension cannot be 0."));

```

```

128     num_batch_elements *= data.shape().dim_size(i);
129 }
130 int num_value_elements = data.shape().num_elements() / num_batch_elements;
131 auto per_batch_counts = BatchedMap<W>(num_batch_elements);
132
133 T max_value = 0;
134
135 const auto data_values = data.flat<T>();
136 const auto weight_values = weights.flat<W>();
137 int i = 0;
138 for (int b = 0; b < num_batch_elements; ++b) {
139     for (int v = 0; v < num_value_elements; ++v) {
140         const auto& value = data_values(i);
141         if (value >= 0 && (maxlength_ <= 0 || value < maxlength_)) {
142             if (binary_output_) {
143                 per_batch_counts[b][value] = 1;
144             } else if (use_weights) {
145                 per_batch_counts[b][value] += weight_values(i);
146             } else {
147                 per_batch_counts[b][value]++;
148             }
149             if (value > max_value) {
150                 max_value = value;
151             }
152         }
153         ++i;
154     }
155 }
156
157 int num_output_values = GetOutputSize(max_value, maxlength_, minlength_);
158 OP_REQUIRES_OK(context, OutputSparse<W>(per_batch_counts, num_output_values,
159                                         is_1d, context));
160 }
161
162 private:
163     int maxlength_;
164     int minlength_;
165     bool binary_output_;
166 };
167
168 ...
168 template <class T, class W>
169 class SparseCount : public OpKernel {
170 public:
171     explicit SparseCount(OpKernelConstruction* context) : OpKernel(context) {
172         OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
173         OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
174         OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
175     }
176

```

```

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                     "greater 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

```

```

275 template <class T, class W>
276 class RaggedCount : public OpKernel {
277 public:
278     explicit RaggedCount(OpKernelConstruction* context) : OpKernel(context) {
279         OP_REQUIRES_OK(context, context->GetAttr("minlength", &minlength_));
280         OP_REQUIRES_OK(context, context->GetAttr("maxlength", &maxlength_));
281         OP_REQUIRES_OK(context, context->GetAttr("binary_output", &binary_output_));
282     }
283
284     void Compute(OpKernelContext* context) override {
285         const Tensor& splits = context->input(0);
286         const Tensor& values = context->input(1);
287         const Tensor& weights = context->input(2);
288         bool use_weights = weights.NumElements() > 0;
289         bool is_1d = false;
290
291         if (use_weights) {
292             OP_REQUIRES(
293                 context, weights.shape() == values.shape(),
294                 errors::InvalidArgument(
295                     "Weights and values must have the same shape. Weight shape: ",
296                     weights.shape().DebugString(),
297                     "; values shape: ", values.shape().DebugString()));
298         }
299
300         const auto splits_values = splits.flat<int64_t>();
301         const auto values_values = values.flat<T>();
302         const auto weight_values = weights.flat<W>();
303         int num_batches = splits.NumElements() - 1;
304         int num_values = values.NumElements();
305
306         OP_REQUIRES(
307             context, num_batches > 0,
308             errors::InvalidArgument(
309                 "Must provide at least 2 elements for the splits argument"));
310         OP_REQUIRES(context, splits_values(0) == 0,
311                     errors::InvalidArgument("Splits must start with 0, not with ",
312                                             splits_values(0)));
313         OP_REQUIRES(context, splits_values(num_batches) == num_values,
314                     errors::InvalidArgument(
315                         "Splits must end with the number of values, got ",
316                         splits_values(num_batches), " instead of ", num_values));
317
318         auto per_batch_counts = BatchedMap<W>(num_batches);
319         T max_value = 0;
320         int batch_idx = 0;
321
322         for (int idx = 0; idx < num_values; ++idx) {
323             while (idx >= splits_values(batch_idx)) {

```

```

324     batch_idx++;
325 }
326 const auto& value = values_values(idx);
327 if (value >= 0 && (maxlength_ <= 0 || value < maxlength_)) {
328     if (binary_output_) {
329         per_batch_counts[batch_idx - 1][value] = 1;
330     } else if (use_weights) {
331         per_batch_counts[batch_idx - 1][value] += weight_values(idx);
332     } else {
333         per_batch_counts[batch_idx - 1][value]++;
334     }
335     if (value > max_value) {
336         max_value = value;
337     }
338 }
339 }
340
341 int num_output_values = GetOutputSize(max_value, maxlength_, minlength_);
342 OP_REQUIRES_OK(context, OutputSparse<W>(per_batch_counts, num_output_values,
343     is_1d, context));
344 }
345
346 private:
347     int maxlength_;
348     int minlength_;
349     bool binary_output_;
350     bool validate_;
351 };
352
353 #define REGISTER_W(W_TYPE) \
354     REGISTER(int32, W_TYPE) \
355     REGISTER(int64_t, W_TYPE)
356
357 #define REGISTER(I_TYPE, W_TYPE) \
358     \
359     REGISTER_KERNEL_BUILDER(Name("DenseCountSparseOutput") \
360         .TypeConstraint<I_TYPE>("T") \
361         .TypeConstraint<W_TYPE>("output_type") \
362         .Device(DEVICE_CPU), \
363         DenseCount<I_TYPE, W_TYPE>) \
364     \
365     REGISTER_KERNEL_BUILDER(Name("SparseCountSparseOutput") \
366         .TypeConstraint<I_TYPE>("T") \
367         .TypeConstraint<W_TYPE>("output_type") \
368         .Device(DEVICE_CPU), \
369         SparseCount<I_TYPE, W_TYPE>) \
370     \
371     REGISTER_KERNEL_BUILDER(Name("RaggedCountSparseOutput") \
372         .TypeConstraint<I_TYPE>("T") \

```



```
373         .TypeConstraint<W_TYPE>("output_type") \
374         .Device(DEVICE_CPU), \
375         RaggedCount<I_TYPE, W_TYPE>)
376
377     TF_CALL_INTEGRAL_TYPES(REGISTER_W);
378     TF_CALL_float(REGISTER_W);
379     TF_CALL_double(REGISTER_W);
380
381     #undef REGISTER_W
382     #undef REGISTER
383
384 } // namespace tensorflow
```