251 lines (225 sloc) | 10.1 KB

```
/* Copyright 2019 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.
==============================================================================*/

#include <locale>
#include <string>

#include "absl/strings/ascii.h"
#include "absl/strings/str_cat.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/platform/errors.h"

namespace tensorflow {
namespace text {

namespace {
template <typename SPLITS_TYPE>
class StringNGramsOp : public tensorflow::OpKernel {
```

```
public:
 explicit StringNGramsOp(tensorflow::OpKernelConstruction* context)
     : tensorflow::OpKernel(context) {
   OP_REQUIRES_OK(context, context->GetAttr("separator", &separator_));
   OP_REQUIRES_OK(context, context->GetAttr("ngram_widths", &ngram_widths_));
   OP_REQUIRES_OK(context, context->GetAttr("left_pad", &left_pad_));
   OP_REQUIRES_OK(context, context->GetAttr("right_pad", &right_pad_));
   OP_REQUIRES_OK(context, context->GetAttr("pad_width", &pad_width_));
   OP_REQUIRES_OK(context, context->GetAttr("preserve_short_sequences",
                                            &preserve_short_));
 }

 int get_pad_width(const int ngram_width) const {
   // Ngrams can be padded with either a fixed pad width or a dynamic pad
   // width depending on the 'pad_width' arg, but in no case should the padding
   // ever be wider than 'ngram_width' - 1.
   return std::min(pad_width_ < 0 ? ngram_width - 1 : pad_width_,
                   ngram_width - 1);
 }

 int get_num_ngrams(const int length, const int ngram_width) const {
   int pad_width = get_pad_width(ngram_width);
   return std::max(0, ((length + 2 * pad_width) - ngram_width) + 1);
 }

 void Compute(tensorflow::OpKernelContext* context) override {
   for (int ngram_width : ngram_widths_) {
     OP_REQUIRES(
         context, ngram_width > 0,
         errors::InvalidArgument("ngram_widths must contain positive values"));
   }

   const tensorflow::Tensor* data;
   OP_REQUIRES_OK(context, context->input("data", &data));
   const auto& input_data = data->flat<tstring>().data();

   const tensorflow::Tensor* splits;
   OP_REQUIRES_OK(context, context->input("data_splits", &splits));
   const auto& splits_vec = splits->flat<SPLITS_TYPE>();

   // Validate that the splits are valid indices into data, only if there are
   // splits specified.
   const int input_data_size = data->flat<tstring>().size();
   const int splits_vec_size = splits_vec.size();
   if (splits_vec_size > 0) {
     int prev_split = splits_vec(0);
     OP_REQUIRES(context, prev_split == 0,
                 errors::InvalidArgument("First split value must be 0, got ",
                                         prev_split));
```

```
      for (int i = 1; i < splits_vec_size; ++i) {
        bool valid_splits = splits_vec(i) >= prev_split;
        valid_splits = valid_splits && (splits_vec(i) <= input_data_size);
        OP_REQUIRES(context, valid_splits,
                    errors::InvalidArgument(
                        "Invalid split value ", splits_vec(i), ", must be in [",
                        prev_split, ", ", input_data_size, "]"));
        prev_split = splits_vec(i);
      }
      OP_REQUIRES(context, prev_split == input_data_size,
                  errors::InvalidArgument(
                      "Last split value must be data size. Expected ",
                      input_data_size, ", got ", prev_split));
    }

    int num_batch_items = splits_vec.size() - 1;
    tensorflow::Tensor* ngrams_splits;
    OP_REQUIRES_OK(
        context, context->allocate_output(1, splits->shape(), &ngrams_splits));
    auto ngrams_splits_data = ngrams_splits->flat<SPLITS_TYPE>().data();

    // If there is no data or size, return an empty RT.
    if (data->flat<tstring>().size() == 0 || splits_vec.size() == 0) {
      tensorflow::Tensor* empty;
      OP_REQUIRES_OK(context,
                     context->allocate_output(0, data->shape(), &empty));
      for (int i = 0; i <= num_batch_items; ++i) {
        ngrams_splits_data[i] = 0;
      }
      return;
    }

    ngrams_splits_data[0] = 0;
    for (int i = 1; i <= num_batch_items; ++i) {
      int length = splits_vec(i) - splits_vec(i - 1);
      int num_ngrams = 0;
      for (int ngram_width : ngram_widths_)
        num_ngrams += get_num_ngrams(length, ngram_width);
      if (preserve_short_ && length > 0 && num_ngrams == 0) {
        num_ngrams = 1;
      }
      ngrams_splits_data[i] = ngrams_splits_data[i - 1] + num_ngrams;
    }

    tensorflow::Tensor* ngrams;
    OP_REQUIRES_OK(
        context,
        context->allocate_output(
            0, TensorShape({ngrams_splits_data[num_batch_items]}), &ngrams));
```

```
    auto ngrams_data = ngrams->flat<tstring>().data();

    for (int i = 0; i < num_batch_items; ++i) {
      auto data_start = &input_data[splits_vec(i)];
      int output_start_idx = ngrams_splits_data[i];
      for (int ngram_width : ngram_widths_) {
        auto output_start = &ngrams_data[output_start_idx];
        int length = splits_vec(i + 1) - splits_vec(i);
        int num_ngrams = get_num_ngrams(length, ngram_width);
        CreateNgrams(data_start, output_start, num_ngrams, ngram_width);
        output_start_idx += num_ngrams;
      }
      // If we're preserving short sequences, check to see if no sequence was
      // generated by comparing the current output start idx to the original
      // one (ngram_splits_data). If no ngrams were generated, then they will
      // be equal (since we increment output_start_idx by num_ngrams every
      // time we create a set of ngrams.)
      if (preserve_short_ && output_start_idx == ngrams_splits_data[i]) {
        int data_length = splits_vec(i + 1) - splits_vec(i);
        // One legitimate reason to not have any ngrams when preserve_short_
        // is true is if the sequence itself is empty. In that case, move on.
        if (data_length == 0) {
          continue;
        }
        // We don't have to worry about dynamic padding sizes here: if padding
        // was dynamic, every sequence would have had sufficient padding to
        // generate at least one ngram.
        int ngram_width = data_length + 2 * pad_width_;
        auto output_start = &ngrams_data[output_start_idx];
        int num_ngrams = 1;
        CreateNgrams(data_start, output_start, num_ngrams, ngram_width);
      }
    }
  }

  void CreateNgrams(const tstring* data, tstring* output, int num_ngrams,
                    int ngram_width) const {
    for (int ngram_index = 0; ngram_index < num_ngrams; ++ngram_index) {
      int pad_width = get_pad_width(ngram_width);
      int left_padding = std::max(0, pad_width - ngram_index);
      int right_padding =
          std::max(0, pad_width - (num_ngrams - (ngram_index + 1)));
      int num_tokens = ngram_width - (left_padding + right_padding);
      int data_start_index = left_padding > 0 ? 0 : ngram_index - pad_width;

      // Calculate the total expected size of the ngram so we can reserve the
      // correct amount of space in the string.
      int ngram_size = 0;
      // Size of the left padding.
```

```
      ngram_size += left_padding * left_pad_.length();
      // Size of the tokens.
      for (int n = 0; n < num_tokens; ++n) {
        ngram_size += data[data_start_index + n].length();
      }
      // Size of the right padding.
      ngram_size += right_padding * right_pad_.length();
      // Size of the separators.
      int num_separators = left_padding + right_padding + num_tokens - 1;
      ngram_size += num_separators * separator_.length();

      // Build the ngram.
      tstring* ngram = &output[ngram_index];
      ngram->reserve(ngram_size);
      for (int n = 0; n < left_padding; ++n) {
        ngram->append(left_pad_);
        ngram->append(separator_);
      }
      // Only output first num_tokens - 1 pairs of data and separator
      for (int n = 0; n < num_tokens - 1; ++n) {
        ngram->append(data[data_start_index + n]);
        ngram->append(separator_);
      }
      // Handle case when there are no tokens or no right padding as these can
      // result in consecutive separators.
      if (num_tokens > 0) {
        // If we have tokens, then output last and then pair each separator with
        // the right padding that follows, to ensure ngram ends either with the
        // token or with the right pad.
        ngram->append(data[data_start_index + num_tokens - 1]);
        for (int n = 0; n < right_padding; ++n) {
          ngram->append(separator_);
          ngram->append(right_pad_);
        }
      } else {
        // If we don't have tokens, then the last item inserted into the ngram
        // has been the separator from the left padding loop above. Hence,
        // output right pad and separator and make sure to finish with a
        // padding, not a separator.
        for (int n = 0; n < right_padding - 1; ++n) {
          ngram->append(right_pad_);
          ngram->append(separator_);
        }
        ngram->append(right_pad_);
      }

      // In debug mode only: validate that we've reserved enough space for the
      // ngram.
      DCHECK_EQ(ngram_size, ngram->size());
```

```
    }
  }

  string separator_;
  string left_pad_;
  string right_pad_;
  bool use_pad_;
  bool extend_pad_;
  bool preserve_short_;

  std::vector<int> ngram_widths_;
  int pad_width_;
};

}  // namespace
REGISTER_KERNEL_BUILDER(Name("StringNGrams")
                            .Device(tensorflow::DEVICE_CPU)
                            .TypeConstraint<int32>("Tsplits"),
                        StringNGramsOp<int32>);
REGISTER_KERNEL_BUILDER(Name("StringNGrams")
                            .Device(tensorflow::DEVICE_CPU)
                            .TypeConstraint<int64_t>("Tsplits"),
                        StringNGramsOp<int64_t>);

}  // namespace text
}  // namespace tensorflow
```