#include "tensorflow/core/framework/op.h"

#include "tensorflow/core/framework/shape\_inference.h"

#include "tensorflow/core/framework/op\_kernel.h"

using namespace tensorflow;

REGISTER\_OP("SparseMaxNorm")

.Attr("epsilon: float")

.Input("max\_values: Ref(float)")

.Input("indices: int64")

.Input("values: float")

.Input("is\_training: bool")

.Output("updated\_max\_values: Ref(float)")

.Output("normalized\_values: float")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that normalizes a batch of sparse inputs based on the current maximum value.

Input

max\_values: float tensor variable representing the max values seen so far.

indices: int64 tensor representing indices representing a feature.

values: float tensor representing values for the current batch.

is\_training: bool tensor specifying if the op should be run in training mode or not.

Outputs

updated\_max\_values: max\_values updated with the current batch.

normalized\_values: Input values normalized by the max value seen so far.

The pseudo code for normalization can be seen below:

# During training / inference

for i, idx in enumerate(indices):

updated\_max\_values[idx] = max(max\_values[idx], abs(values[i]))

normalized\_values[i] = values[i] / updated\_max\_values[idx]

)doc");

class SparseMaxNorm : public OpKernel {

private:

float epsilon\_;

public:

explicit SparseMaxNorm(OpKernelConstruction \*context) : OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("epsilon", &epsilon\_));

}

void Compute(OpKernelContext \*context) override {

// We always return the input ref.

context->forward\_ref\_input\_to\_ref\_output(0, 0);

Tensor max\_values\_tensor = context->mutable\_input(0, false);

OP\_REQUIRES(context, max\_values\_tensor.IsInitialized(),

errors::FailedPrecondition("Attempting to use uninitialized "

"parameters: ",

requested\_input(0)));

const Tensor &indices\_tensor = context->input(1);

const Tensor &values\_tensor = context->input(2);

const Tensor &is\_training\_tensor = context->input(3);

const auto indices = indices\_tensor.flat<int64>();

const auto values = values\_tensor.flat<float>();

const bool is\_training = is\_training\_tensor.scalar<bool>()();

auto max\_values = max\_values\_tensor.flat<float>();

Tensor \*normalized\_values\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(1, values\_tensor.shape(),

&normalized\_values\_tensor));

auto normalized\_values = normalized\_values\_tensor->flat<float>();

const int64 N = indices.size();

for (int64 i = 0; i < N; i++) {

int64 idx = indices(i);

float value = values(i);

float max\_value = std::max(max\_values(idx), std::abs(value));

// Guaranteed to be between [-1, 1].

normalized\_values(i) = value / std::max(max\_value, epsilon\_);

if (is\_training) {

max\_values(idx) = max\_value;

}

}

}

};

REGISTER\_OP("SparseBatchNorm")

.Attr("input\_size: int")

.Attr("epsilon: float")

.Input("means: Ref(float)")

.Input("variances: Ref(float)")

.Input("indices: int64")

.Input("values: float")

.Input("is\_training: bool")

.Output("updated\_means: Ref(float)")

.Output("updated\_vars: Ref(float)")

.Output("normalized\_values: float")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that performs batch normalization.

Attr

input\_size: Size of the inputs.

epsilon: The minimum value of the variance.

Input

mean: float tensor variable representing the running mean seen so far.

variances: float tensor variable representing the running variance seen so far.

indices: int64 tensor representing indices representing a feature.

values: float tensor representing values for the current batch.

is\_training: bool tensor specifying if the op should be run in training mode or not.

Outputs

updated\_means: mean updated with the current batch.

updated\_vars: variances updated with the current batch.

normalized\_values: Input values normalized by the max value seen so far.

The pseudo code for normalization can be seen below:

if is\_training:

means, variances = update\_metrics(means, variances, values)

normalized\_values = (values - means) / sqrt(variances + epsilon)

return normalized\_values \* gamma + beta

)doc");

class SparseBatchNorm : public OpKernel {

private:

std::vector<int64> counts\_;

std::vector<float> m2s\_;

float epsilon\_;

public:

explicit SparseBatchNorm(OpKernelConstruction \*context) : OpKernel(context) {

int64 input\_size;

OP\_REQUIRES\_OK(context, context->GetAttr("input\_size", &input\_size));

OP\_REQUIRES\_OK(context, context->GetAttr("epsilon", &epsilon\_));

counts\_.resize(input\_size);

m2s\_.resize(input\_size);

}

void Compute(OpKernelContext \*context) override {

// We always return the input ref.

context->forward\_ref\_input\_to\_ref\_output(0, 0);

context->forward\_ref\_input\_to\_ref\_output(1, 1);

Tensor means\_tensor = context->mutable\_input(0, true);

Tensor variances\_tensor = context->mutable\_input(1, true);

OP\_REQUIRES(context, means\_tensor.IsInitialized(),

errors::FailedPrecondition("Attempting to use uninitialized "

"parameters: ",

requested\_input(0)));

OP\_REQUIRES(context, variances\_tensor.IsInitialized(),

errors::FailedPrecondition("Attempting to use uninitialized "

"parameters: ",

requested\_input(1)));

const Tensor &indices\_tensor = context->input(2);

const Tensor &values\_tensor = context->input(3);

const Tensor &is\_training\_tensor = context->input(4);

const auto indices = indices\_tensor.flat<int64>();

const auto values = values\_tensor.flat<float>();

const bool is\_training = is\_training\_tensor.scalar<bool>()();

auto means = means\_tensor.flat<float>();

auto variances = variances\_tensor.flat<float>();

Tensor \*normalized\_values\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(2, values\_tensor.shape(),

&normalized\_values\_tensor));

auto normalized\_values = normalized\_values\_tensor->flat<float>();

const int64 N = indices.size();

if (is\_training) {

// Accumulate, mean, count, sum of squared differences.

// Reference wiki:

// https://en.wikipedia.org/wiki/Algorithms\_for\_calculating\_variance#Online\_algorithm

// Reference paper:

// https://www.jstor.org/stable/1266577?seq=1#page\_scan\_tab\_contents

for (int64 i = 0; i < N; i++) {

int64 idx = indices(i);

int64 count = counts\_[idx] + 1;

float value = values(i);

float old\_mean = means(idx);

float old\_delta = value - old\_mean;

float new\_mean = old\_mean + old\_delta / count;

float new\_delta = value - new\_mean;

counts\_[idx] = count;

m2s\_[idx] += new\_delta \* old\_delta;

means(idx) = new\_mean;

variances(idx) = m2s\_[idx] / count;

}

}

// Normalize the values

for (int64 i = 0; i < N; i++) {

int64 idx = indices(i);

float stdev = std::sqrt(variances(idx) + epsilon\_);

normalized\_values(i) = (values(i) - means(idx)) / stdev;

}

}

};

REGISTER\_OP("SparseMaxNormInference")

.Attr("epsilon: float")

.Input("max\_values: float")

.Input("indices: int64")

.Input("values: float")

.Output("normalized\_values: float")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that normalizes a batch of sparse inputs based on the current maximum value.

This is the inference OP.

Input

max\_values: float tensor representing the max values seen so far.

indices: int64 tensor representing indices representing a feature.

values: float tensor representing values for the current batch.

Outputs

normalized\_values: Input values normalized by the max value seen so far.

The pseudo code for normalization can be seen below:

# During inference

for i, idx in enumerate(indices):

updated\_max\_values[idx] = max(max\_values[idx], abs(values[i]))

normalized\_values[i] = values[i] / updated\_max\_values[idx]

)doc");

class SparseMaxNormInference : public OpKernel {

private:

float epsilon\_;

public:

explicit SparseMaxNormInference(OpKernelConstruction \*context) : OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("epsilon", &epsilon\_));

}

void Compute(OpKernelContext \*context) override {

const Tensor &max\_values\_tensor = context->input(0);

const Tensor &indices\_tensor = context->input(1);

const Tensor &values\_tensor = context->input(2);

const auto max\_values = max\_values\_tensor.flat<float>();

const auto indices = indices\_tensor.flat<int64>();

const auto values = values\_tensor.flat<float>();

Tensor \*normalized\_values\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, values\_tensor.shape(),

&normalized\_values\_tensor));

auto normalized\_values = normalized\_values\_tensor->flat<float>();

const int64 N = indices.size();

for (int64 i = 0; i < N; i++) {

int64 idx = indices(i);

float value = values(i);

float max\_value = std::max(max\_values(idx), std::abs(value));

// Guaranteed to be between [-1, 1].

normalized\_values(i) = value / std::max(max\_value, epsilon\_);

}

}

};

REGISTER\_OP("SparseMaxNormTraining")

.Attr("epsilon: float")

.Input("max\_values: float")

.Input("indices: int64")

.Input("values: float")

.Output("updated\_max\_values: float")

.Output("normalized\_values: float")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that normalizes a batch of sparse inputs based on the current maximum value.

This is the training OP.

Input

max\_values: float tensor variable representing the max values seen so far.

indices: int64 tensor representing indices representing a feature.

values: float tensor representing values for the current batch.

Outputs

updated\_max\_values: max\_values updated with the current batch.

normalized\_values: Input values normalized by the max value seen so far.

The pseudo code for normalization can be seen below:

# During training

for i, idx in enumerate(indices):

updated\_max\_values[idx] = max(max\_values[idx], abs(values[i]))

normalized\_values[i] = values[i] / updated\_max\_values[idx]

)doc");

class SparseMaxNormTraining : public OpKernel {

private:

float epsilon\_;

public:

explicit SparseMaxNormTraining(OpKernelConstruction \*context) : OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("epsilon", &epsilon\_));

}

void Compute(OpKernelContext \*context) override {

const Tensor &max\_values\_tensor = context->input(0);

const Tensor &indices\_tensor = context->input(1);

const Tensor &values\_tensor = context->input(2);

const auto max\_values = max\_values\_tensor.flat<float>();

const auto indices = indices\_tensor.flat<int64>();

const auto values = values\_tensor.flat<float>();

Tensor \*updated\_max\_values\_tensor = nullptr;

Tensor \*normalized\_values\_tensor = nullptr;

OP\_REQUIRES\_OK(context, context->allocate\_output(0, max\_values\_tensor.shape(),

&updated\_max\_values\_tensor));

OP\_REQUIRES\_OK(context, context->allocate\_output(1, values\_tensor.shape(),

&normalized\_values\_tensor));

auto updated\_max\_values = updated\_max\_values\_tensor->flat<float>();

auto normalized\_values = normalized\_values\_tensor->flat<float>();

const int64 N = indices.size();

// This copy is needed because the values of updated\_max\_values are originally garbage.

// Also note that N is not the same as max\_values.size()

std::copy(max\_values.data(), max\_values.data() + max\_values.size(), updated\_max\_values.data());

for (int64 i = 0; i < N; i++) {

int64 idx = indices(i);

float value = values(i);

float updated\_max\_value = std::max(updated\_max\_values(idx), std::abs(value));

// Guaranteed to be between [-1, 1].

normalized\_values(i) = value / std::max(updated\_max\_value, epsilon\_);

// Saving the updated\_max\_values

updated\_max\_values(idx) = updated\_max\_value;

}

}

};

REGISTER\_KERNEL\_BUILDER(

Name("SparseMaxNorm")

.Device(DEVICE\_CPU),

SparseMaxNorm);

REGISTER\_KERNEL\_BUILDER(

Name("SparseBatchNorm")

.Device(DEVICE\_CPU),

SparseBatchNorm);

REGISTER\_KERNEL\_BUILDER(

Name("SparseMaxNormInference")

.Device(DEVICE\_CPU),

SparseMaxNormInference);

REGISTER\_KERNEL\_BUILDER(

Name("SparseMaxNormTraining")

.Device(DEVICE\_CPU),

SparseMaxNormTraining);