"""

A custom optimizer to implement Deep Gradient Compression. The general idea of

gradient compression is to compress the gradients exchanged across machines,

in order to reduce the communication overhead of distributing computing efforts.

More details in https://arxiv.org/abs/1712.01887

"""

# TODO: Test how much communication overhead this DeepGradientCompressionOptimizer can reduce under

# multi-GPU and distributed setting.

import tensorflow.compat.v1 as tf

def compute\_threshold(grad, density):

"""

A utility function to compute the threshold for gradient sparsification, given the gradient

tensor and the density.

Args:

grad(tf.Tensor):

Gradient tensor for some variable.

density(float):

Density degree when sparsifying gradients.

Returns(float):

Threshold for gradient sparsification.

"""

flat\_grad = tf.reshape(grad, [-1])

abs\_flat\_grad = tf.abs(flat\_grad)

size = tf.shape(abs\_flat\_grad)[0]

k = tf.maximum(tf.constant(1),

tf.cast(tf.scalar\_mul(density, tf.cast(size, tf.float32)), tf.int32))

topk, \_ = tf.nn.top\_k(abs\_flat\_grad, k, False)

return topk[-1]

def get\_top\_row\_indices(values, density):

"""

A utility function to get indices of most significant rows, given the density degree.

Args:

values(tf.Tensor):

Gradient or locally accumulated gradient for some variable.

density(float):

Density degree when filtering out rows.

Returns(list(int)):

Indices of most significant rows.

"""

abs\_values = tf.abs(values)

try:

row\_num = tf.shape(abs\_values)[0]

k = tf.maximum(tf.constant(1),

tf.cast(tf.scalar\_mul(density, tf.cast(row\_num, tf.float32)), tf.int32))

row\_sums = tf.squeeze(tf.reduce\_sum(values, axis=1, keepdims=True))

\_, top\_row\_indices = tf.nn.top\_k(row\_sums, k=k, sorted=False)

# print "abs\_values", abs\_values, "row\_sums", row\_sums

return top\_row\_indices

# return tf.range(row\_num)

except ValueError: # if the tensor is 0-D or 1-D

return None

class DeepGradientCompressionOptimizer(tf.train.GradientDescentOptimizer):

"""

A custom optimizer to implement Deep Gradient Compression (https://arxiv.org/abs/1712.01887).

"""

def \_\_init\_\_(self, learning\_rate, use\_locking=False, name="Sparse",

density=1.0,

density\_decay=False,

density\_decay\_steps=10000,

density\_decay\_rate=0.5,

min\_density=0.1,

accumulation=False):

super(DeepGradientCompressionOptimizer, self).\_\_init\_\_(learning\_rate, use\_locking, name)

self.\_initial\_density\_t = tf.convert\_to\_tensor(density)

self.\_density\_decay = density\_decay

dtype = self.\_initial\_density\_t.dtype

self.\_density\_decay\_steps\_t = tf.convert\_to\_tensor(density\_decay\_steps, dtype)

self.\_density\_decay\_rate\_t = tf.convert\_to\_tensor(density\_decay\_rate, dtype)

self.\_min\_density\_t = tf.convert\_to\_tensor(min\_density, dtype)

self.\_accumulation = accumulation

def \_prepare(self):

super(DeepGradientCompressionOptimizer, self).\_prepare()

if not self.\_density\_decay:

self.\_density\_t = self.\_initial\_density\_t

else:

dtype = self.\_initial\_density\_t.dtype

global\_step = tf.cast(tf.train.get\_global\_step(), dtype)

p = tf.floor(tf.divide(global\_step, self.\_density\_decay\_steps\_t))

decayed\_density = tf.multiply(self.\_initial\_density\_t,

tf.pow(self.\_density\_decay\_rate\_t, p))

self.\_density\_t = tf.maximum(self.\_min\_density\_t, decayed\_density)

def \_create\_slots(self, var\_list):

"""

Create a slot variable to accumulate gradients locally for each variable in `var\_list`.

Args:

var\_list(list(tf.Variable)):

List of variables to accumulate gradients locally for.

"""

for var in var\_list:

self.\_zeros\_slot(var, "g\_buffer", self.\_name)

def \_apply\_dense(self, grad, var):

if not self.\_accumulation:

top\_row\_indices = get\_top\_row\_indices(grad, self.\_density\_t)

if top\_row\_indices is None:

return super(DeepGradientCompressionOptimizer, self).\_apply\_dense(grad, var)

sparsified\_values = tf.gather(grad, top\_row\_indices)

sparsified\_indices = top\_row\_indices

sparsified\_grad = tf.IndexedSlices(sparsified\_values, sparsified\_indices)

return super(DeepGradientCompressionOptimizer, self).\_apply\_sparse\_duplicate\_indices(

sparsified\_grad, var)

else:

g\_buffer = self.get\_slot(var, "g\_buffer")

g\_buffer = tf.assign\_add(g\_buffer, grad)

top\_row\_indices = get\_top\_row\_indices(g\_buffer, self.\_density\_t)

if top\_row\_indices is None:

return super(DeepGradientCompressionOptimizer, self).\_apply\_dense(grad, var)

sparsified\_values = tf.gather(g\_buffer, top\_row\_indices)

sparsified\_indices = top\_row\_indices

sparsified\_grad = tf.IndexedSlices(sparsified\_values, sparsified\_indices)

update\_var = super(DeepGradientCompressionOptimizer, self).\_apply\_sparse\_duplicate\_indices(

sparsified\_grad, var)

update\_g\_buffer = tf.scatter\_update(g\_buffer, sparsified\_indices, tf.zeros\_like(

sparsified\_values))

return tf.group(\*[update\_var, update\_g\_buffer])

def \_apply\_sparse\_duplicate\_indices(self, grad, var):

if not self.\_accumulation:

top\_row\_indices = get\_top\_row\_indices(grad.values, self.\_density\_t)

if top\_row\_indices is None:

return super(DeepGradientCompressionOptimizer, self).\_apply\_sparse\_duplicate\_indices(grad, var) # noqa: E501

sparsified\_values = tf.gather(grad.values, top\_row\_indices)

sparsified\_indices = tf.gather(grad.indices, top\_row\_indices)

sparsified\_grad = tf.IndexedSlices(sparsified\_values, sparsified\_indices)

return super(DeepGradientCompressionOptimizer, self).\_apply\_sparse\_duplicate\_indices(

sparsified\_grad, var)

else:

g\_buffer = self.get\_slot(var, "g\_buffer")

g\_buffer = tf.scatter\_update(g\_buffer, grad.indices, grad.values)

top\_row\_indices = get\_top\_row\_indices(g\_buffer, self.\_density\_t)

if top\_row\_indices is None:

return super(DeepGradientCompressionOptimizer,

self).\_apply\_sparse\_duplicate\_indices(grad, var)

sparsified\_values = tf.gather(g\_buffer, top\_row\_indices)

sparsified\_indices = top\_row\_indices

sparsified\_grad = tf.IndexedSlices(sparsified\_values, sparsified\_indices)

update\_var = super(DeepGradientCompressionOptimizer, self).\_apply\_sparse\_duplicate\_indices(

sparsified\_grad, var)

update\_g\_buffer = tf.scatter\_update(g\_buffer, sparsified\_indices, tf.zeros\_like(

sparsified\_values))

return tf.group(\*[update\_var, update\_g\_buffer])