# pylint: disable=missing-docstring, bare-except, pointless-statement,

# pointless-string-statement, redundant-unittest-assert, no-else-return,

# no-member, old-style-class, dangerous-default-value, protected-access,

# too-few-public-methods

import os

import numpy as np

import yaml

"""

Utility to load tensors serialized by Deepbird V1.

Note that Deepbird V1 serialize tensor names as \"weight\".\'1\'.

For user-friendliness, the quotes are removed from the tensor names.

"""

# helper class used to assist hierarchical key access by remembering intermediate keys.

class \_KeyRecorder(object):

def \_\_init\_\_(self, tensorio, keys=[]):

self.tensorio = tensorio

self.keys = keys

def \_\_getitem\_\_(self, k):

new\_keys = self.keys + [str(k)]

prefix = ".".join(new\_keys)

key\_list = self.tensorio.list\_tensors()

# if we have a complete key, load the tensor.

if prefix in key\_list:

return self.tensorio.\_load(prefix)

# we don't have a complete key yet, but at least one tensor should start with this prefix.

for k\_value in key\_list:

if k\_value.startswith(prefix):

return \_KeyRecorder(self.tensorio, new\_keys)

# if no key starts with the prefix, this \_key\_recorder is not valid.

raise ValueError("Key not found: " + prefix)

# convert tensorio tensor type to numpy data type.

# also returns element size in bytes.

def \_get\_data\_type(data\_type):

if data\_type == 'Double':

return (np.float64, 8)

if data\_type == 'Float':

return (np.float32, 4)

if data\_type == 'Int':

return (np.int32, 4)

if data\_type == 'Long':

return (np.int64, 8)

if data\_type == 'Byte':

return (np.int8, 1)

raise ValueError('Unexpected tensorio data type: ' + data\_type)

class TensorIO(object):

"""

Construct a TensorIO class.

tensorio\_path: a directory containing tensors serialized using tensorio. tar file not supported.

mmap\_tensor:

By default, loaded tensors use mmap storage.

Set this to false to not use mmap. Useful when loading multiple tensors.

"""

def \_\_init\_\_(self, tensorio\_path, mmap\_tensor=True):

self.\_tensorio\_path = tensorio\_path

self.\_mmap\_tensor = mmap\_tensor

# Make sure we can locate spec.yaml.

yaml\_file = os.path.join(tensorio\_path, 'spec.yaml')

if not os.path.exists(yaml\_file):

raise ValueError('Invalid tensorio path: no spec.yaml found.')

# load spec.yaml.

with open(yaml\_file, 'r') as file\_open:

# Note that tensor names in the yaml are like this: \"weight\".\'1\'

# For user-friendliness, we remove the quotes.

\_spec = yaml.safe\_load(file\_open)

self.\_spec = {k.replace("'", '').replace('"', ''): v for (k, v) in \_spec.items()}

def list\_tensors(self):

"""

Returns a list of tensors saved in the given path.

"""

return self.\_spec.keys()

def \_load\_tensor(self, name):

"""

Load Tensor with the given name.

Raise value error if the named tensor is not found.

Returns a numpy array if the named tensor is found.

"""

tensor\_info = self.\_spec[name]

if tensor\_info['type'] != 'tensor':

raise ValueError('Trying to load a tensor of unknown type: ' + tensor\_info['type'])

filename = os.path.join(self.\_tensorio\_path, tensor\_info['filename'])

(data\_type, element\_size) = \_get\_data\_type(tensor\_info['tensorType'])

np\_array = np.memmap(

filename,

dtype=data\_type,

mode='r',

# -1 because lua offset is 1 based.

offset=(tensor\_info['offset'] - 1) \* element\_size,

shape=tuple(tensor\_info['size']),

order='C',

)

return np\_array if self.\_mmap\_tensor else np\_array[:].copy()

def \_load\_nontensor\_data(self, name):

"""

Load non-tensor data with the given name.

Returns a python string.

"""

tensor\_info = self.\_spec[name]

return tensor\_info['data']

def \_load(self, name):

"""

Load data serialized under the given name, it could be a tensor or regular data.

"""

if name not in self.\_spec:

raise ValueError('The specified key {} is not found in {}'.format(name, self.\_tensorio\_path))

data\_type = self.\_spec[name]['type']

if data\_type == 'tensor':

return self.\_load\_tensor(name)

else:

return self.\_load\_nontensor\_data(name)

def load\_all(self):

"""

Load all tensors stored in the tensorio directory.

Returns a dictionary from tensor name to numpy arrays.

"""

return {k: self.\_load(k) for k in self.\_spec}

###########################################

# The below are utilities for convenience #

###########################################

def \_\_getitem\_\_(self, k):

"""

Shorthand for \_load\_tensor, but also supports hierarchical access like: tensorio['a']['b']['1']

"""

if k in self.\_spec:

# We have a full tensor name, directly load it.

return self.\_load\_tensor(k)

else:

return \_KeyRecorder(self)[k]