import os

import re

import time

from collections import OrderedDict

from absl import logging

import numpy as np

import tensorflow.compat.v1 as tf

from tensorflow.python.ops.lookup\_ops import index\_table\_from\_tensor

import twml

# Padding is 0, UNK is 1:

PAD\_WORD\_ID = 0

OOV\_WORD\_ID = 1

def load\_initializers\_from\_csv(

embedding\_path, vocab\_size=-1, embedding\_size=None, separator=None, vocab=None

):

"""

Loads embeddings saved in the `glove format <https://nlp.stanford.edu/projects/glove/>`\_.

The glove format is a txt file separated by spaces.

Each line looks like: "word 0.00001 0.2334 ...".

Arguments:

embedding\_path:

path to the embeddings file on HDFS (hdfs://default/...)

or its local\_path (/path/to/...).

The embedding\_path may also specify a pattern. In which case, the embeddings

are read in the lexical order of the filenames that match the order.

vocab\_size:

the maximum size of the vocabulary. The top ``vocab\_size`` words in the file

are included in the vocabulary. If you specify a positive vocab\_size,

the words are expected to be in descending order of frequency.

This allows the embeddings to be easily filtered to top vocab\_size words.

Reducing the vocab\_size acts as a regularizer, preventing the model to overfit on rarer words.

A negative vocab\_size loads all embeddings.

Reducing the vocab\_size may also help with memory issues,

allowing the embedding initializers to fit inside the graph.

embedding\_size:

Defaults to None. If None, the embedding size is infered from the file name.

For example, ``glove.300d.txt`` and ``glove300d200.txt`` will both infrered

as ``embedding\_size=300``. If this can't be done, the ``embedding\_size`` is

inferred from the first line in the file. If ``embedding\_size`` is provided,

only the last ``embedding\_size`` values of each line are considered. This

allows the line parser to recover from partial word parsing errors.

separator:

Specifies the separator to use when splitting each line into values.

Default value is a whitespace (same as glove format).

vocab:

OrderedDict mapping words to np.array embedding vectors. Initializes the vocabulary.

Duplicate words found in the file are ignored.

Defaults to a vocabulary of two words::

vocab = OrderedDict()

vocab[''] = np.random.randn(embedding\_size)

vocab['<UNK>'] = np.random.randn(embedding\_size)

Returns:

tuple of (vocab\_initializer, weight\_initializer, shape)

vocab\_initializer:

A tf.constant\_initializer containing a vector of word strings of size vocab\_size.

weight\_initializer:

A twml.contrib.initializers.partition\_constant\_initializer containing

the weight matrix of embeddings of size vocab\_size x embedding\_size.

shape:

A tuple containing of (vocab\_size, embedding\_size).

"""

start = time.time()

embedding\_path = twml.util.sanitize\_hdfs\_path(embedding\_path)

is\_user\_vocab = True

if vocab is None:

vocab = OrderedDict()

vocab[''] = True

vocab['<UNK>'] = True

is\_user\_vocab = False

elif not isinstance(vocab, OrderedDict):

raise RuntimeError(

"Expecting vocab argument of type OrderedDict or None. "

"Got type %s instead." % type(vocab).\_\_name\_\_

)

if embedding\_size is None:

embedding\_file = os.path.basename(embedding\_path)

match = re.search(r"[^\d]([\d]+)d", embedding\_file)

if match is not None:

embedding\_size = int(match.group(1))

if embedding\_size is not None and not isinstance(embedding\_size, int):

raise RuntimeError(

"Expecting embedding\_size argument of type int or None. "

"Got type %s, instead." % type(embedding\_size).\_\_name\_\_

)

embedding\_paths = sorted(tf.io.gfile.glob(embedding\_path))

if len(embedding\_paths) > 1:

raise ValueError(

"You are most likely using a the wrong --embedding.path"

)

embedding\_path = embedding\_paths[0]

logging.info("Reading embeddings file from path %s.." % embedding\_path)

with tf.io.gfile.GFile(embedding\_path) as f:

lines = f.readlines()

logging.info("Done reading embeddings file from path %s." % embedding\_path)

logging.info("Parsing vocbulary and embeddings...")

for line in lines:

# Word and weights separated by space

values = line.strip().split(separator)

# Word is first symbol on each line

word = values[0]

if word not in vocab:

if embedding\_size is None or embedding\_size <= 0:

# get all elements after the first one.

word\_weights = values[1:]

embedding\_size = len(word\_weights)

else:

# get the last embedding\_size elements

word\_weights = values[-min(embedding\_size, len(values) - 1) :]

try:

if len(word\_weights) != embedding\_size:

raise ValueError

word\_weights = np.asarray(word\_weights, dtype=np.float32)

vocab[word] = word\_weights

except ValueError:

logging.info("Wasn't able to load embeddings for word '%s'. Ignoring it" % word)

vocab\_len = len(vocab)

if vocab\_size > 0 and vocab\_len == vocab\_size:

# Limit vocabulary to top terms

break

elif (vocab\_len % 1000) == 0:

logging.info("Loaded %d words into vocab" % vocab\_len)

else:

logging.info("found duplicate word: %s" % word)

if not is\_user\_vocab:

vocab[''] = np.random.randn(embedding\_size)

vocab['<UNK>'] = np.random.randn(embedding\_size)

words = list(vocab.keys())

weights = list(vocab.values())

weights = np.asarray(weights, dtype=np.float32)

assert weights.shape[0] == len(vocab)

assert weights.shape[1] == embedding\_size

vocab\_initializer = tf.constant\_initializer(words, tf.string)

weight\_initializer = twml.contrib.initializers.PartitionConstant(weights, tf.float32)

logging.info("Loaded %d embeddings in %d seconds." % (len(vocab), time.time() - start))

return vocab\_initializer, weight\_initializer, weights.shape

def add\_parser\_arguments(parser):

"""

Adds the embedding.path and embedding.vocab\_size command-line arguments to the parser.

These can be used to call an initializer loader function like

the ``load\_initializers\_from\_csv`` function.

Arguments:

parser: argparse.ArgumentParser instance obtained from Trainer.get\_trainer\_parser

Returns:

argparse.ArgumentParser instance with discretizer-specific arguments added

"""

parser.add\_argument(

"--embedding.path",

"--embedding\_path",

dest="embedding\_path",

type=str,

default=None,

help="When specified, loads glove embeddings from .txt glove file",

)

parser.add\_argument(

"--embedding.vocab\_size",

"--embedding\_vocab\_size",

dest="embedding\_vocab\_size",

type=int,

default=-1,

help="Size of vocabulary. Uses this many of the most frequent terms. Defaults to -1 (use full vocab).",

)

return parser

class EmbeddingLookup(twml.layers.Layer):

"""Layer for looking up embeddings.

Transforms a sequence of strings to a sequence of embeddings.

Arguments:

vocab\_size:

The number of word strings and embeddings in the vocabulary.

output\_size:

Long or Integer, dimensionality of the output space. The embedding vector size.

vocab\_initializer:

Initializer function for the vocabulary. Required. The initializer should

return a list of strings of size vocab\_size.

weight\_initializer:

Initializer function for the weight matrix of size vocab\_size x output\_size.

This argument defaults to zeros\_initializer().

This is valid when the EmbeddingLookup is the first layer of

parameters but should be changed otherwise.

trainable:

Boolean, if `True` adds variables to the graph collection

``GraphKeys.TRAINABLE\_VARIABLES`` (see `tf.Variable

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/Variable>`\_).

Defaults to True: trains the embeddings.

num\_oov\_buckets:

The number of buckets to use for OOV strings. These bucket ids occur after the vocab bucket

ids. Hashing is used to assign OOV strings to these buckets. If `num\_oov\_buckets` is not

specified, index `OOV\_WORD\_ID` is used for OOV strings.

name:

String, the name of the layer. Layers with the same name will

share weights, but to avoid mistakes we require ``reuse=True`` in such cases.

num\_partitions:

Number of partitions to use for the weight variable. Defaults to 1.

partition\_axis:

If num\_partitions is specified, the partition axis for the weight variable

Defaults to 0 (partition by row).

Must be 0 (row) or 1 (column, does not support yet)

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

dtype:

Defaults to tf.float32. Specifies the dtype of the weights.

use\_placeholder:

Defaults to True.

If set to `True`, the initializer is passed via a placeholder. The initializer in this case needs to be of type `keras.initializers.Constant`.

If set to `False`, the initializer becomes part of the graph. This can sometimes be beyond what protobuf clients support.

checkpoint\_dir:

Default to None.

If set to the path of a checkpoint, load embedding from the checkpoint.

convert\_to\_lowercase:

Default to True.

Converting all string inputs to lowercase.

Notes: If `use\_placeholder` is set to `True`, the feed dictionary can be accessed by calling `twml.contrib.initializers.get\_init\_feed\_dict()`.

"""

def \_\_init\_\_(

self,

vocab\_size,

output\_size,

vocab\_initializer,

weight\_initializer=None,

trainable=True,

num\_oov\_buckets=None,

oov\_word\_id=None,

name=None,

num\_partitions=1,

partition\_axis=0,

weight\_regularizer=None,

dtype=None,

use\_placeholder=True,

checkpoint\_dir=None,

convert\_to\_lowercase=True,

\*\*kwargs,

):

if dtype is None:

# prevents a bug where the parent class defaults to the type of the first input tensor.

dtype = tf.float32

super().\_\_init\_\_(trainable=trainable, name=name, dtype=dtype, \*\*kwargs)

# Weights initialization is set to 0s. This is safe for full sparse layers because

# you are supposed to learn your embedding from the label.

is\_constant\_init = isinstance(weight\_initializer, tf.keras.initializers.Constant)

if use\_placeholder and (not is\_constant\_init) and (weight\_initializer is not None):

raise ValueError("Weight initializer should be a `Constant` or `None`.")

if weight\_initializer is None:

self.weight\_initializer = tf.zeros\_initializer()

else:

self.weight\_initializer = weight\_initializer

self.use\_placeholder = use\_placeholder

self.checkpoint\_dir = checkpoint\_dir

self.convert\_to\_lowercase = convert\_to\_lowercase

self.vocab\_initializer = vocab\_initializer

self.vocab\_size = vocab\_size

self.output\_size = output\_size

self.num\_partitions = num\_partitions

self.partition\_axis = partition\_axis

self.weight\_regularizer = weight\_regularizer

self.trainable = trainable

self.oov\_word\_id = oov\_word\_id

self.num\_oov\_buckets = num\_oov\_buckets

if self.oov\_word\_id is not None and self.num\_oov\_buckets is not None:

raise ValueError("At most one of oov\_word\_id or num\_oov\_buckets should be specified")

elif self.oov\_word\_id is None and self.num\_oov\_buckets is None:

self.oov\_word\_id = OOV\_WORD\_ID # use the default OOV word id

if partition\_axis != 0:

raise NotImplementedError("embedding\_lookup only supports partition\_axis = 0")

def build(self, input\_shapes):

"""

creates the ``vocab`` and ``weight`` Variables

of shape ``[vocab\_size]`` and ``[vocab\_size, output\_size]`` respectively.

"""

partitioner = None

additional\_buckets\_for\_oov = self.num\_oov\_buckets if self.num\_oov\_buckets is not None else 0

shape = [self.vocab\_size + additional\_buckets\_for\_oov, self.output\_size]

if self.use\_placeholder:

embedding\_weight\_initializer = twml.contrib.initializers.PlaceholderInitializer(

shape, self.dtype

)

tf.add\_to\_collection(

twml.contrib.initializers.TWML\_INIT\_FEED\_KEY,

{embedding\_weight\_initializer.value: self.weight\_initializer.value},

)

else:

embedding\_weight\_initializer = self.weight\_initializer

if self.num\_partitions:

partition\_axis = int(self.partition\_axis)

partitioner = tf.fixed\_size\_partitioner(self.num\_partitions, axis=partition\_axis)

else:

# Regular variables do not like it when you pass both constant tensors and shape

if not callable(self.weight\_initializer):

shape = None

self.vocab = self.add\_variable(

'vocab',

initializer=self.vocab\_initializer,

shape=[self.vocab\_size],

dtype=tf.string,

trainable=False,

)

self.weight = self.add\_variable(

'weight',

initializer=None if self.checkpoint\_dir is not None else embedding\_weight\_initializer,

regularizer=self.weight\_regularizer,

shape=shape,

dtype=self.dtype,

trainable=self.trainable,

partitioner=partitioner,

)

if self.checkpoint\_dir is not None:

twml.trainers.trainer.init\_from\_checkpoint(self.checkpoint\_dir, {'weight': self.weight.name})

self.built = True

def call(

self, inputs, debug=False, oov\_summaries=False, \*\*kwargs

): # pylint: disable=unused-argument

"""Converts word strings to word ids using the vocabulary lookup table.

Then converts the word ids to their commensurate embedding vector.

Arguments:

inputs:

A tensor of word strings. Typically, of size batch\_size x seq\_len.

debug:

When True, prints the input strings and their commensurate input\_ids.

Defaults to False.

oov\_summaries:

When True, log the out-of-vocabulary (OOV) rate to TensorBoard

Defaults to False.

Returns:

The mapping of input word strings to output embedding vectors.

Given an input of shape ``batch\_size x seq\_len``, the output has shape

``batch\_size x seq\_len x embedding\_size``.

"""

if self.convert\_to\_lowercase:

inputs = tf.strings.lower(inputs)

if self.num\_oov\_buckets is None:

lookup\_table = index\_table\_from\_tensor(self.vocab, default\_value=self.oov\_word\_id)

else:

lookup\_table = index\_table\_from\_tensor(self.vocab, num\_oov\_buckets=self.num\_oov\_buckets)

input\_ids = lookup\_table.lookup(inputs)

if oov\_summaries:

oov\_count = tf.reduce\_sum(

tf.cast(tf.math.equal(input\_ids, self.oov\_word\_id), tf.dtypes.float32)

)

valid\_count = tf.reduce\_sum(

tf.cast(tf.math.not\_equal(input\_ids, PAD\_WORD\_ID), tf.dtypes.float32)

)

oov\_rate = oov\_count / valid\_count

tf.summary.scalar('OOV\_rate', oov\_rate)

if debug:

def print\_debug():

return tf.print("input\_strings:", inputs, "\ninput\_ids: ", input\_ids, summarize=140)

with tf.control\_dependencies([twml.util.do\_every\_n\_steps(print\_debug, 1000)]):

input\_ids = tf.identity(input\_ids)

output\_embeddings = tf.nn.embedding\_lookup(

params=self.weight, ids=input\_ids, partition\_strategy='div'

)

output\_shape = inputs.shape.concatenate(tf.TensorShape([self.output\_size]))

output\_embeddings.set\_shape(output\_shape)

return output\_embeddings