

This shows the text split into lists

This is the code I was trying to use for building the space vector model

# Copyright 2015 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.

# ==============================================================================

"""Multi-threaded word2vec mini-batched skip-gram model.

Trains the model described in:

(Mikolov, et. al.) Efficient Estimation of Word Representations in Vector Space

ICLR 2013.

http://arxiv.org/abs/1301.3781

This model does traditional minibatching.

The key ops used are:

\* placeholder for feeding in tensors for each example.

\* embedding\_lookup for fetching rows from the embedding matrix.

\* sigmoid\_cross\_entropy\_with\_logits to calculate the loss.

\* GradientDescentOptimizer for optimizing the loss.

\* skipgram custom op that does input processing.

"""

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import os

import sys

import threading

import time

from six.moves import xrange # pylint: disable=redefined-builtin

import numpy as np

import tensorflow as tf

word2vec = tf.load\_op\_library(os.path.join(os.path.dirname(os.path.realpath(\_\_file\_\_)), 'word2vec\_ops.so'))

flags = tf.app.flags

flags.DEFINE\_string("save\_path", None, "Directory to write the model and "

"training summaries.")

flags.DEFINE\_string("train\_data", None, "Training text file. "

"E.g., unzipped file http://mattmahoney.net/dc/text8.zip.")

flags.DEFINE\_string(

"eval\_data", None, "File consisting of analogies of four tokens."

"embedding 2 - embedding 1 + embedding 3 should be close "

"to embedding 4."

"See README.md for how to get 'questions-words.txt'.")

flags.DEFINE\_integer("embedding\_size", 200, "The embedding dimension size.")

flags.DEFINE\_integer(

"epochs\_to\_train", 15,

"Number of epochs to train. Each epoch processes the training data once "

"completely.")

flags.DEFINE\_float("learning\_rate", 0.2, "Initial learning rate.")

flags.DEFINE\_integer("num\_neg\_samples", 100,

"Negative samples per training example.")

flags.DEFINE\_integer("batch\_size", 16,

"Number of training examples processed per step "

"(size of a minibatch).")

flags.DEFINE\_integer("concurrent\_steps", 12,

"The number of concurrent training steps.")

flags.DEFINE\_integer("window\_size", 5,

"The number of words to predict to the left and right "

"of the target word.")

flags.DEFINE\_integer("min\_count", 5,

"The minimum number of word occurrences for it to be "

"included in the vocabulary.")

flags.DEFINE\_float("subsample", 1e-3,

"Subsample threshold for word occurrence. Words that appear "

"with higher frequency will be randomly down-sampled. Set "

"to 0 to disable.")

flags.DEFINE\_boolean(

"interactive", False,

"If true, enters an IPython interactive session to play with the trained "

"model. E.g., try model.analogy(b'france', b'paris', b'russia') and "

"model.nearby([b'proton', b'elephant', b'maxwell'])")

flags.DEFINE\_integer("statistics\_interval", 5,

"Print statistics every n seconds.")

flags.DEFINE\_integer("summary\_interval", 5,

"Save training summary to file every n seconds (rounded "

"up to statistics interval).")

flags.DEFINE\_integer("checkpoint\_interval", 600,

"Checkpoint the model (i.e. save the parameters) every n "

"seconds (rounded up to statistics interval).")

FLAGS = flags.FLAGS

class Options(object):

"""Options used by our word2vec model."""

def \_\_init\_\_(self):

# Model options.

# Embedding dimension.

self.emb\_dim = FLAGS.embedding\_size

# Training options.

# The training text file.

self.train\_data = FLAGS.train\_data

# Number of negative samples per example.

self.num\_samples = FLAGS.num\_neg\_samples

# The initial learning rate.

self.learning\_rate = FLAGS.learning\_rate

# Number of epochs to train. After these many epochs, the learning

# rate decays linearly to zero and the training stops.

self.epochs\_to\_train = FLAGS.epochs\_to\_train

# Concurrent training steps.

self.concurrent\_steps = FLAGS.concurrent\_steps

# Number of examples for one training step.

self.batch\_size = FLAGS.batch\_size

# The number of words to predict to the left and right of the target word.

self.window\_size = FLAGS.window\_size

# The minimum number of word occurrences for it to be included in the

# vocabulary.

self.min\_count = FLAGS.min\_count

# Subsampling threshold for word occurrence.

self.subsample = FLAGS.subsample

# How often to print statistics.

self.statistics\_interval = FLAGS.statistics\_interval

# How often to write to the summary file (rounds up to the nearest

# statistics\_interval).

self.summary\_interval = FLAGS.summary\_interval

# How often to write checkpoints (rounds up to the nearest statistics

# interval).

self.checkpoint\_interval = FLAGS.checkpoint\_interval

# Where to write out summaries.

self.save\_path = FLAGS.save\_path

if not os.path.exists(self.save\_path):

os.makedirs(self.save\_path)

# Eval options.

# The text file for eval.

self.eval\_data = FLAGS.eval\_data

class Word2Vec(object):

"""Word2Vec model (Skipgram)."""

def \_\_init\_\_(self, options, session):

self.\_options = options

self.\_session = session

self.\_word2id = {}

self.\_id2word = []

self.build\_graph()

self.build\_eval\_graph()

self.save\_vocab()

def read\_analogies(self):

"""Reads through the analogy question file.

Returns:

questions: a [n, 4] numpy array containing the analogy question's

word ids.

questions\_skipped: questions skipped due to unknown words.

"""

questions = []

questions\_skipped = 0

with open(self.\_options.eval\_data, "rb") as analogy\_f:

for line in analogy\_f:

if line.startswith(b":"): # Skip comments.

continue

words = line.strip().lower().split(b" ")

ids = [self.\_word2id.get(w.strip()) for w in words]

if None in ids or len(ids) != 4:

questions\_skipped += 1

else:

questions.append(np.array(ids))

print("Eval analogy file: ", self.\_options.eval\_data)

print("Questions: ", len(questions))

print("Skipped: ", questions\_skipped)

self.\_analogy\_questions = np.array(questions, dtype=np.int32)

def forward(self, examples, labels):

"""Build the graph for the forward pass."""

opts = self.\_options

# Declare all variables we need.

# Embedding: [vocab\_size, emb\_dim]

init\_width = 0.5 / opts.emb\_dim

emb = tf.Variable(

tf.random\_uniform(

[opts.vocab\_size, opts.emb\_dim], -init\_width, init\_width),

name="emb")

self.\_emb = emb

# Softmax weight: [vocab\_size, emb\_dim]. Transposed.

sm\_w\_t = tf.Variable(

tf.zeros([opts.vocab\_size, opts.emb\_dim]),

name="sm\_w\_t")

# Softmax bias: [vocab\_size].

sm\_b = tf.Variable(tf.zeros([opts.vocab\_size]), name="sm\_b")

# Global step: scalar, i.e., shape [].

self.global\_step = tf.Variable(0, name="global\_step")

# Nodes to compute the nce loss w/ candidate sampling.

labels\_matrix = tf.reshape(

tf.cast(labels,

dtype=tf.int64),

[opts.batch\_size, 1])

# Negative sampling.

sampled\_ids, \_, \_ = (tf.nn.fixed\_unigram\_candidate\_sampler(

true\_classes=labels\_matrix,

num\_true=1,

num\_sampled=opts.num\_samples,

unique=True,

range\_max=opts.vocab\_size,

distortion=0.75,

unigrams=opts.vocab\_counts.tolist()))

# Embeddings for examples: [batch\_size, emb\_dim]

example\_emb = tf.nn.embedding\_lookup(emb, examples)

# Weights for labels: [batch\_size, emb\_dim]

true\_w = tf.nn.embedding\_lookup(sm\_w\_t, labels)

# Biases for labels: [batch\_size, 1]

true\_b = tf.nn.embedding\_lookup(sm\_b, labels)

# Weights for sampled ids: [num\_sampled, emb\_dim]

sampled\_w = tf.nn.embedding\_lookup(sm\_w\_t, sampled\_ids)

# Biases for sampled ids: [num\_sampled, 1]

sampled\_b = tf.nn.embedding\_lookup(sm\_b, sampled\_ids)

# True logits: [batch\_size, 1]

true\_logits = tf.reduce\_sum(tf.multiply(example\_emb, true\_w), 1) + true\_b

# Sampled logits: [batch\_size, num\_sampled]

# We replicate sampled noise labels for all examples in the batch

# using the matmul.

sampled\_b\_vec = tf.reshape(sampled\_b, [opts.num\_samples])

sampled\_logits = tf.matmul(example\_emb,

sampled\_w,

transpose\_b=True) + sampled\_b\_vec

return true\_logits, sampled\_logits

def nce\_loss(self, true\_logits, sampled\_logits):

"""Build the graph for the NCE loss."""

# cross-entropy(logits, labels)

opts = self.\_options

true\_xent = tf.nn.sigmoid\_cross\_entropy\_with\_logits(

labels=tf.ones\_like(true\_logits), logits=true\_logits)

sampled\_xent = tf.nn.sigmoid\_cross\_entropy\_with\_logits(

labels=tf.zeros\_like(sampled\_logits), logits=sampled\_logits)

# NCE-loss is the sum of the true and noise (sampled words)

# contributions, averaged over the batch.

nce\_loss\_tensor = (tf.reduce\_sum(true\_xent) +

tf.reduce\_sum(sampled\_xent)) / opts.batch\_size

return nce\_loss\_tensor

def optimize(self, loss):

"""Build the graph to optimize the loss function."""

# Optimizer nodes.

# Linear learning rate decay.

opts = self.\_options

words\_to\_train = float(opts.words\_per\_epoch \* opts.epochs\_to\_train)

lr = opts.learning\_rate \* tf.maximum(

0.0001, 1.0 - tf.cast(self.\_words, tf.float32) / words\_to\_train)

self.\_lr = lr

optimizer = tf.train.GradientDescentOptimizer(lr)

train = optimizer.minimize(loss,

global\_step=self.global\_step,

gate\_gradients=optimizer.GATE\_NONE)

self.\_train = train

def build\_eval\_graph(self):

"""Build the eval graph."""

# Eval graph

# Each analogy task is to predict the 4th word (d) given three

# words: a, b, c. E.g., a=italy, b=rome, c=france, we should

# predict d=paris.

# The eval feeds three vectors of word ids for a, b, c, each of

# which is of size N, where N is the number of analogies we want to

# evaluate in one batch.

analogy\_a = tf.placeholder(dtype=tf.int32) # [N]

analogy\_b = tf.placeholder(dtype=tf.int32) # [N]

analogy\_c = tf.placeholder(dtype=tf.int32) # [N]

# Normalized word embeddings of shape [vocab\_size, emb\_dim].

nemb = tf.nn.l2\_normalize(self.\_emb, 1)

# Each row of a\_emb, b\_emb, c\_emb is a word's embedding vector.

# They all have the shape [N, emb\_dim]

a\_emb = tf.gather(nemb, analogy\_a) # a's embs

b\_emb = tf.gather(nemb, analogy\_b) # b's embs

c\_emb = tf.gather(nemb, analogy\_c) # c's embs

# We expect that d's embedding vectors on the unit hyper-sphere is

# near: c\_emb + (b\_emb - a\_emb), which has the shape [N, emb\_dim].

target = c\_emb + (b\_emb - a\_emb)

# Compute cosine distance between each pair of target and vocab.

# dist has shape [N, vocab\_size].

dist = tf.matmul(target, nemb, transpose\_b=True)

# For each question (row in dist), find the top 4 words.

\_, pred\_idx = tf.nn.top\_k(dist, 4)

# Nodes for computing neighbors for a given word according to

# their cosine distance.

nearby\_word = tf.placeholder(dtype=tf.int32) # word id

nearby\_emb = tf.gather(nemb, nearby\_word)

nearby\_dist = tf.matmul(nearby\_emb, nemb, transpose\_b=True)

nearby\_val, nearby\_idx = tf.nn.top\_k(nearby\_dist,

min(1000, self.\_options.vocab\_size))

# Nodes in the construct graph which are used by training and

# evaluation to run/feed/fetch.

self.\_analogy\_a = analogy\_a

self.\_analogy\_b = analogy\_b

self.\_analogy\_c = analogy\_c

self.\_analogy\_pred\_idx = pred\_idx

self.\_nearby\_word = nearby\_word

self.\_nearby\_val = nearby\_val

self.\_nearby\_idx = nearby\_idx

def build\_graph(self):

"""Build the graph for the full model."""

opts = self.\_options

# The training data. A text file.

(words, counts, words\_per\_epoch, self.\_epoch, self.\_words, examples,

labels) = word2vec.skipgram\_word2vec(filename=opts.train\_data,

batch\_size=opts.batch\_size,

window\_size=opts.window\_size,

min\_count=opts.min\_count,

subsample=opts.subsample)

(opts.vocab\_words, opts.vocab\_counts,

opts.words\_per\_epoch) = self.\_session.run([words, counts, words\_per\_epoch])

opts.vocab\_size = len(opts.vocab\_words)

print("Data file: ", opts.train\_data)

print("Vocab size: ", opts.vocab\_size - 1, " + UNK")

print("Words per epoch: ", opts.words\_per\_epoch)

self.\_examples = examples

self.\_labels = labels

self.\_id2word = opts.vocab\_words

for i, w in enumerate(self.\_id2word):

self.\_word2id[w] = i

true\_logits, sampled\_logits = self.forward(examples, labels)

loss = self.nce\_loss(true\_logits, sampled\_logits)

tf.summary.scalar("NCE loss", loss)

self.\_loss = loss

self.optimize(loss)

# Properly initialize all variables.

tf.global\_variables\_initializer().run()

self.saver = tf.train.Saver()

def save\_vocab(self):

"""Save the vocabulary to a file so the model can be reloaded."""

opts = self.\_options

with open(os.path.join(opts.save\_path, "vocab.txt"), "w") as f:

for i in xrange(opts.vocab\_size):

vocab\_word = tf.compat.as\_text(opts.vocab\_words[i]).encode("utf-8")

f.write("%s %d\n" % (vocab\_word,

opts.vocab\_counts[i]))

def \_train\_thread\_body(self):

initial\_epoch, = self.\_session.run([self.\_epoch])

while True:

\_, epoch = self.\_session.run([self.\_train, self.\_epoch])

if epoch != initial\_epoch:

break

def train(self):

"""Train the model."""

opts = self.\_options

initial\_epoch, initial\_words = self.\_session.run([self.\_epoch, self.\_words])

summary\_op = tf.summary.merge\_all()

summary\_writer = tf.summary.FileWriter(opts.save\_path, self.\_session.graph)

workers = []

for \_ in xrange(opts.concurrent\_steps):

t = threading.Thread(target=self.\_train\_thread\_body)

t.start()

workers.append(t)

last\_words, last\_time, last\_summary\_time = initial\_words, time.time(), 0

last\_checkpoint\_time = 0

while True:

time.sleep(opts.statistics\_interval) # Reports our progress once a while.

(epoch, step, loss, words, lr) = self.\_session.run(

[self.\_epoch, self.global\_step, self.\_loss, self.\_words, self.\_lr])

now = time.time()

last\_words, last\_time, rate = words, now, (words - last\_words) / (

now - last\_time)

print("Epoch %4d Step %8d: lr = %5.3f loss = %6.2f words/sec = %8.0f\r" %

(epoch, step, lr, loss, rate), end="")

sys.stdout.flush()

if now - last\_summary\_time > opts.summary\_interval:

summary\_str = self.\_session.run(summary\_op)

summary\_writer.add\_summary(summary\_str, step)

last\_summary\_time = now

if now - last\_checkpoint\_time > opts.checkpoint\_interval:

self.saver.save(self.\_session,

os.path.join(opts.save\_path, "model.ckpt"),

global\_step=step.astype(int))

last\_checkpoint\_time = now

if epoch != initial\_epoch:

break

for t in workers:

t.join()

return epoch

def \_predict(self, analogy):

"""Predict the top 4 answers for analogy questions."""

idx, = self.\_session.run([self.\_analogy\_pred\_idx], {

self.\_analogy\_a: analogy[:, 0],

self.\_analogy\_b: analogy[:, 1],

self.\_analogy\_c: analogy[:, 2]

})

return idx

def eval(self):

"""Evaluate analogy questions and reports accuracy."""

# How many questions we get right at precision@1.

correct = 0

try:

total = self.\_analogy\_questions.shape[0]

except AttributeError as e:

raise AttributeError("Need to read analogy questions.")

start = 0

while start < total:

limit = start + 2500

sub = self.\_analogy\_questions[start:limit, :]

idx = self.\_predict(sub)

start = limit

for question in xrange(sub.shape[0]):

for j in xrange(4):

if idx[question, j] == sub[question, 3]:

# Bingo! We predicted correctly. E.g., [italy, rome, france, paris].

correct += 1

break

elif idx[question, j] in sub[question, :3]:

# We need to skip words already in the question.

continue

else:

# The correct label is not the precision@1

break

print()

print("Eval %4d/%d accuracy = %4.1f%%" % (correct, total,

correct \* 100.0 / total))

def analogy(self, w0, w1, w2):

"""Predict word w3 as in w0:w1 vs w2:w3."""

wid = np.array([[self.\_word2id.get(w, 0) for w in [w0, w1, w2]]])

idx = self.\_predict(wid)

for c in [self.\_id2word[i] for i in idx[0, :]]:

if c not in [w0, w1, w2]:

print(c)

return

print("unknown")

def nearby(self, words, num=20):

"""Prints out nearby words given a list of words."""

ids = np.array([self.\_word2id.get(x, 0) for x in words])

vals, idx = self.\_session.run(

[self.\_nearby\_val, self.\_nearby\_idx], {self.\_nearby\_word: ids})

for i in xrange(len(words)):

print("\n%s\n=====================================" % (words[i]))

for (neighbor, distance) in zip(idx[i, :num], vals[i, :num]):

print("%-20s %6.4f" % (self.\_id2word[neighbor], distance))

def \_start\_shell(local\_ns=None):

# An interactive shell is useful for debugging/development.

import IPython

user\_ns = {}

if local\_ns:

user\_ns.update(local\_ns)

user\_ns.update(globals())

IPython.start\_ipython(argv=[], user\_ns=user\_ns)

def main(\_):

"""Train a word2vec model."""

if not FLAGS.train\_data or not FLAGS.eval\_data or not FLAGS.save\_path:

print("--train\_data --eval\_data and --save\_path must be specified.")

sys.exit(1)

opts = Options()

with tf.Graph().as\_default(), tf.Session() as session:

with tf.device("/cpu:0"):

model = Word2Vec(opts, session)

model.read\_analogies() # Read analogy questions

for \_ in xrange(opts.epochs\_to\_train):

model.train() # Process one epoch

model.eval() # Eval analogies.

# Perform a final save.

model.saver.save(session,

os.path.join(opts.save\_path, "model.ckpt"),

global\_step=model.global\_step)

if FLAGS.interactive:

# E.g.,

# [0]: model.analogy(b'france', b'paris', b'russia')

# [1]: model.nearby([b'proton', b'elephant', b'maxwell'])

\_start\_shell(locals())

if \_\_name\_\_ == "\_\_main\_\_":

tf.app.run()