Parallelizing word2vec in Python

RADIM ŘEHŮŘEK (HTTPS://RARE-TECHNOLOGIES.COM/AUTHOR/RADIM/) /
2013-10-04 /
GENSIM (HTTPS://RARE-TECHNOLOGIES.COM/CATEGORY/GENSIM/),
PROGRAMMING (HTTPS://RARE-TECHNOLOGIES.COM/CATEGORY/PROGRAMMING/) /
21 COMMENTS (HTTPS://RARE-TECHNOLOGIES.COM/PARALLELIZING-WORD2VEC-IN-PYTHON/#CO

The final instalment on optimizing word2vec in Python: how to make use of multicore machines.

You may want to read Part One (http://radimrehurek.com/2013/09/deep-learning-with-word2vec-and-gensim/) and Part Two (http://radimrehurek.com/2013/09/word2vec-in-python-part-two-optimizing/) first.

Multi-what?

The original C toolkit allows setting a "*-threads N*" parameter, which effectively **splits the training corpus into N parts**, each to be processed by a separate thread in parallel. The result is a nice speed-up: 1.9x for N=2 threads, 3.2x for N=4.

We'd like to be able to do the same with the gensim port.

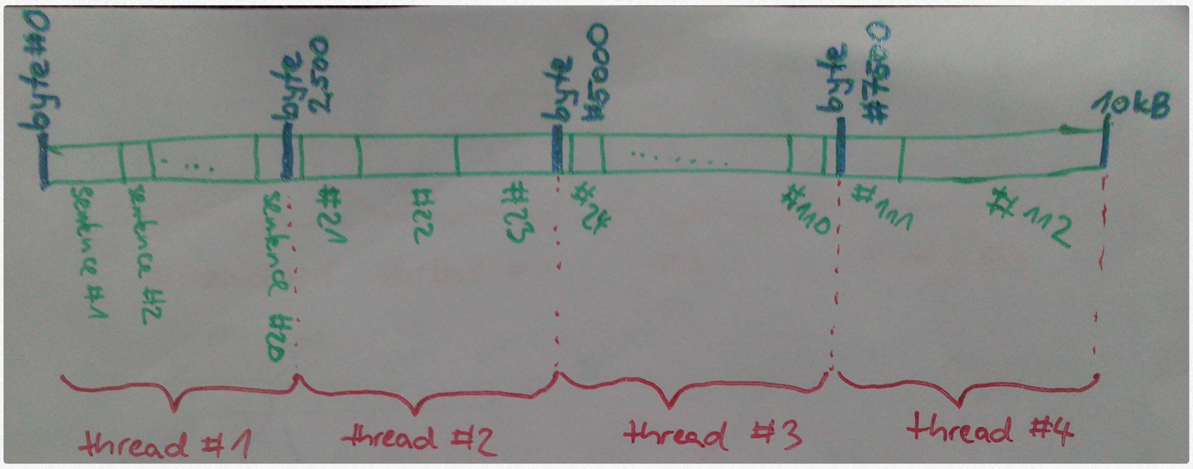
In standard Python (https://en.wikipedia.org/wiki/CPython) world, the answer to **"multi-processing or multi-threading?"** is usually "multiprocessing". The reason is the so-called GIL aka global interpreter lock (https://wiki.python.org/moin/GlobalInterpreterLock), which effectively enforces single-threaded execution for CPU-bound tasks, no matter how many threads you launch.

On the other hand, creating new processes with multiprocessing has its own issues. It works differently on Windows which is a maintenance headache, plus the way Python does it (fork without exec) can mess up some libraries. One such example is Apple's BLAS on Mac (=the Accelerate framework with its "grand central dispatch"), as investigated here (https://mail.scipy.org/pipermail/numpy-discussion/2012-August/063589.html) by none other than the machine learning guru Olivier Grisel (https://twitter.com/ogrisel). No matter where you go, ogrisel has been there first!

With word2vec, we're already working with low-level C calls, so the GIL is no issue. I therefore decided to parallelize word2vec using threads, just like the original C toolkit.

One two three (not only you and me)

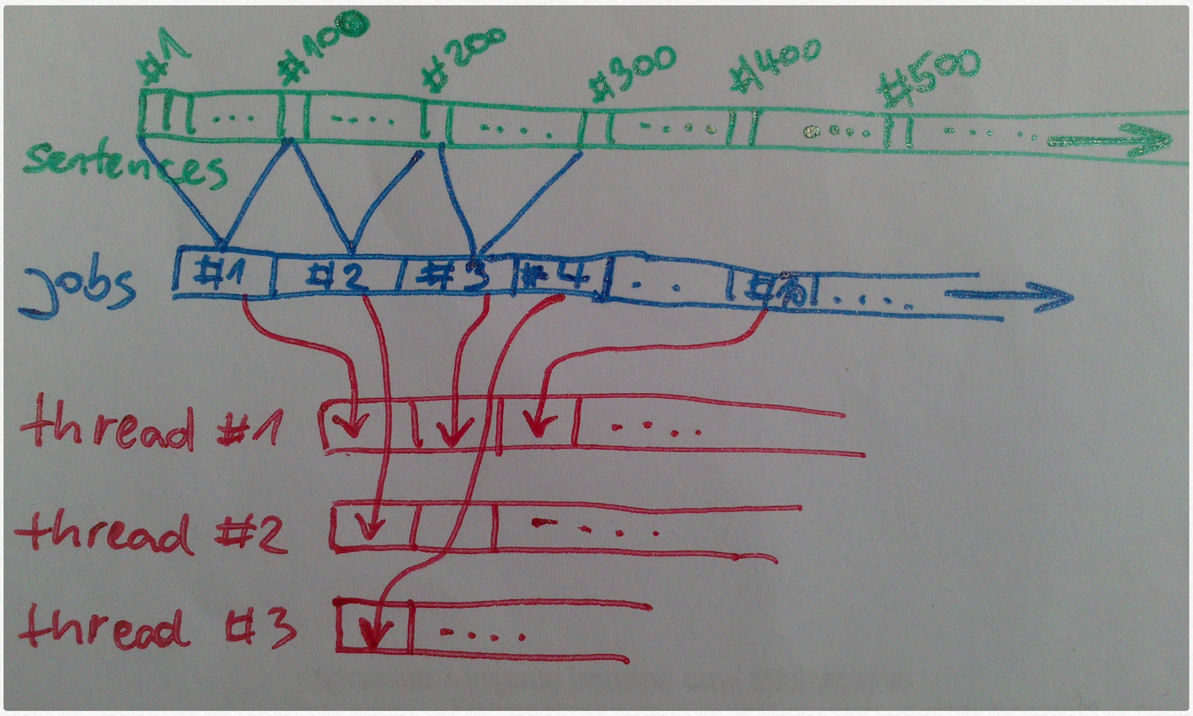
The **original C tool** assumes the training corpus resides in a file on disk, in a fixed format. Each thread will open this file and seek into its assigned byte position, splitting the file size evenly among the N threads. Then it starts parsing words from there, building sentences & training on the sentences sequentially:



Dividing an input file into threads in the original C word2vec. Note that there are nasty edge cases, like initially seeking into the middle of a word, or a sentence.

A thread will process a fixed number of *words* (not bytes) before terminating, so if the word lengths are distributed unevenly, some file parts may be trained on twice, and some never. I personally think the I/O handling is not the prettiest part of the C word2vec.

The **gensim word2vec port** accepts a generic sequence of sentences, which can come from a filesystem, network, or even be created on-the-fly as a stream, so there's no seeking or skipping to the middle. Instead, we'll take a fixed number of sentences (100 by default) and put them in a "job" queue, from which worker threads will repeatedly lift jobs for training:



Chunking an input stream of sentences into jobs that are sent to threads, in gensim word2vec.

Python contains excellent built-in tools for both multiprocessing and threading, so adjusting the code to use several threads was fairly trivial. The final threaded code (logging omitted) looks like this:

```python
def worker_train(job):
    """
    Train model on a list of sentences = a job.
    Each worker runs in a separate thread, executing this function repeatedl

    """
    # update the learning rate before every job
    alpha = max(0.0001, initial_alpha * (1 - 1.0 * word_count / total_words)
    # return back how many words we trained on
    # out-of-vocabulary (unknown) words are excluded and do not count
    return sum(train_sentence(model, sentence, alpha) for sentence in job)

jobs = utils.grouper(sentences, 100)  # generator; 1 job = 100 sentences
pool = multiprocessing.pool.ThreadPool(num_threads)  # start worker threads
word_count = 0
for job_words in pool.imap(worker_train, jobs):  # process jobs in parallel
    word_count += job_words
pool.close()
pool.join()
```

Interestingly, the producer/consumer pattern made the code clearer and more explicit, in my opinion. That's already a plus, regardless of any speed-up.

The core computation is done inside *train_sentence*, which is the tiny function I optimized with Cython and BLAS last time (http://radimrehurek.com/2013/09/word2vec-in-python-part-two-optimizing/). As I mentioned above, we must release the GIL for multithreading to be practical, and this is also done inside *train_sentence*, using Cython's *"with nogil:"* syntax (http://docs.cython.org/src/userguide/external_C_code.html#releasing-the-gil).

For the curious: in the original C word2vec, all threads access the same matrix of neural weights, and there's no locking, so they may be overwriting the same weights willy-nilly at the same time. This is also true for the gensim port. Apparently this hack even has a fancy name in academia according to Tomáš (https://groups.google.com/d/msg/word2vec-toolkit/NLvYXU99cAM/rryQhcaxKSQJ): "asynchronous stochastic gradient descent". But just to be sure, I measured the model accuracy for different numbers of threads below, too.

Timings

I'll be measuring performance on the first 100,000,000 bytes of the English Wikipedia text, aka the text8 corpus (http://mattmahoney.net/dc/text8.zip). At 17,005,207 words, it's about 17x larger than the Brown corpus I used in Part II. The reported speed is in "words per second" again, taken from the best of three runs. Accuracy is computed on the questions-words.txt (https://code.google.com/p/word2vec/source/browse/trunk/questions-words.txt) file that comes with the original C word2vec:

	# worker threads (speed/peak RAM/accuracy)			
Implementation	**1**	**2**	**3**	**4**
C word2vec	22.6k / 252MB / 27.4%	42.94k / 252MB / 26.4%	62.04k / 252MB / 26.8%	72.44k / 252MB / 27.2%
gensim word2vec	109.5k / 591MB / 27.5%	191.6k / 596MB / 27.1%	263k / 592MB / 27.3%	**311.7k** / 601MB / **28.2%**

In other words: **2 threads equal a 1.75x speed improvement, 4 threads 2.85x**, compared to one thread. This is worse than C's respective 1.9x and 3.2x, presumably because of all the thread-safe job queueing that happens in Python.

I'm not sure why the original C toolkit has slightly inferior accuracy results than that Python port though. The input, the algorithm and the evaluation process are exactly the same. It may have to do with the different way the training corpus is split among threads and therefore there are different collisions. Or with its random number generation… or the I/O parsing. Working with text in C is no fun and fairly error prone. I don't have time (nor will) to reconstruct and replicate the C code that deals with I/O, but if you have an idea, let me know in the comments.

I was naturally interested in how the accuracy improves with larger corpora, so I also ran both implementations on **the first billion bytes (http://mattmahoney.net/dc/textdata.html) of wiki** (~124M words). With four threads, the C code took 28 minutes overall (78k words/s in the training part) and scored 47.3% accuracy; the Python code 9½ minutes (318k words/s) and **50.3% accuracy**.

Memory

During this optimization series, I didn't talk about memory yet. How much memory does the Python port need, especially compared to the original C code?

Python has a reputation of being memory intense… which is completely true. It contains its own memory manager, so there's no control over how much memory something really consumes. For example, the above model built on text8 needs ~280MB, roughly the same as the C code. But Python over-allocates and creates temporary copies during training, never to return this memory back to the OS until the process exits, even when the memory's not needed anymore. So the actual RAM footprint, as OS X sees it, is more than double that, ~600MB.

The good news is, this **memory footprint is constant**, regardless of how large the training corpus is. This is true both for the C toolkit and for the Python port. Only a small, fixed number of sentences is loaded and held in RAM at any one time.

Summary

With hyper-threading ("fake" 8 cores on my quadcore i7 MacbookPro), Cython-optimized hotspot loop, BLAS and all the other optimizations, **the best result I got was 44 seconds, or 378k words/s**, on the text8 corpus. Compared to the initial implementation in NumPy, which took several hours at 1.2k words/s, and even to the 91k of the original C tool on the same setup, this is a substantial improvement and a reason to rejoice!

As this is the third and final blog on word2vec, I'll conclude with a few practical points:

- **Where**: The code lives inside gensim (http://radimrehurek.com/gensim/), release >= 0.8.8. Installation instructions (http://radimrehurek.com/gensim/install.html). Don't forget to install Cython (http://docs.cython.org/src/quickstart/install.html) too (*pip install cython*), to make use of all the optimizations described here.

- **How**: Documentation & API reference: gensim website (http://radimrehurek.com/gensim/models/word2vec.html). From Python shell, use standard docstrings: *help(gensim.models.word2vec)* etc. For the great and ever-improving IPython (https://ipython.org/): *gensim.models.word2vec?*. I also added a simple **command line example**: when you run *python -m gensim.models.word2vec text8 questions-words.txt*, a model will be trained and its accuracy evaluated, with the log telling you what's happening. Use as a template.

- **When**: These blog posts always reflect the gensim code at the time of writing. To go back in time to check out an older version/replicate older results, use the git versioning (https://github.com/piskvorky/gensim/).

I have to say these were pleasantly spent 3 weekends. It's fun to go back to the roots and just optimize stuff, without any care for business and management 🙂 Of course, I have to thank Tomáš Mikolov and his colleagues, who did all the hard work so that people like me can play around. I hope they publish a new, improved algorithm soon — with all the scaffolding done, a Python port should be much simpler next time.

(**If you liked this series, you may also like this word2vec tutorial (http://radimrehurek.com/2014/02/word2vec-tutorial) and benchmark of libraries for nearest-neighbour (kNN) search (http://radimrehurek.com/2013/11/performance-shootout-of-nearest-neighbours-intro/).**)

GENSIM (HTTPS://RARE-TECHNOLOGIES.COM/TAG/GENSIM/)   OPTIMIZATION (HTTPS://RARE-TECHNOLOGIES.COM/TAG/OPTIMIZATION/)

WORD2VEC (HTTPS://RARE-TECHNOLOGIES.COM/TAG/WORD2VEC/)

21 COMMENTS

SUMIT

2013-10-15 AT 8:10 PM (HTTPS://RARE-TECHNOLOGIES.COM/PARALLELIZING-WORD2VEC-IN-PYTHON/#COMMENT-2184)