Why is \texttt{word2vec} so fast?

Efficiency tricks for neural nets

Taylor Berg-Kirkpatrick

Carnegie Mellon University
Language Technologies Institute

Site
https://phontron.com/class/nn4nlp2017/
Glamorous Life of an AI Scientist

**Perception**

![Image of a glamorous person]

**Reality**

```
neubig@itachi:~$ python nn-lm.py
[dy?net] random seed: 3454201866
[dy?net] allocating memory: 512MB
-- finished 500 sentences
-- finished 1000 sentences
-- finished 1500 sentences
-- finished 2000 sentences
-- finished 2500 sentences
-- finished 3000 sentences
-- finished 3500 sentences
-- finished 4000 sentences
```

Waiting....

Photo Credit: Antoine Miech @ Twitter
Why are Neural Networks Slow and What Can we Do?

- Big operations, especially for softmaxes over large vocabularies
  - \(\rightarrow\) **Approximate operations or use GPUs**

- GPUs love big operations, but hate doing lots of them
  - \(\rightarrow\) **Reduce the number of operations** through optimized implementations or batching

- Our networks are big, our data sets are big
  - \(\rightarrow\) **Use parallelism** to process many data at once
Sampling-based Softmax Approximations
A Visual Example of the Softmax

\[
p = \text{softmax}(W h + b)
\]
Sampling-based Approximations

- Calculate the denominator over a subset $W$ and $b$
- Sample negative examples according to distribution $q$
Softmax

• Convert scores into probabilities by taking the exponent and normalizing (softmax)

\[
P(x_i \mid h_i) = \frac{e^{s(x_i \mid h_i)}}{\sum_{\tilde{x}_i} e^{s(\tilde{x}_i \mid h_i)}}
\]

This is expensive, would like to approximate

\[
Z(h_i) = \sum_{\tilde{x}_i} e^{s(\tilde{x}_i \mid h_i)}
\]
Importance Sampling
(Bengio and Senecal 2003)

• Sampling is a way to approximate a distribution we cannot calculate exactly

• **Basic idea:** sample from arbitrary distribution $Q$ (uniform/unigram), then re-weight with $e^s/Q$ to approximate denominator

$$Z(h_i) \approx \frac{1}{N} \sum_{\tilde{x}_i \sim Q(\cdot | h_i)} \frac{e^{s(\tilde{x}_i | h_i)}}{Q(\tilde{x}_i | h_i)}$$

• This is a biased estimator (esp. when $N$ is small)
Noise Contrastive Estimation
(Mnih & Teh 2012)

• **Basic idea:** Try to guess whether it is a true sample or one of \( N \) random noise samples. Prob. of true:

\[
P(d = 1 \mid x_i, h_i) = \frac{P(x_i \mid h_i)}{P(x_i \mid h_i) + N \times Q(x_i \mid h_i)}
\]

• Optimize the probability of guessing correctly:

\[
\mathbb{E}_P[\log P(d = 1 \mid x_i, h_i)] + N \times \mathbb{E}_Q[\log P(d = 0 \mid x_i, h_i)]
\]

• During training, approx. with unnormalized prob.

\[
\tilde{P}(x_i \mid h_i) = \frac{P(x_i \mid h_i)}{e^{cn_i}} \quad \text{(set } c_{h_i} = 0)\]

\[
\tilde{P}(x_i \mid h_i) = P(x_i \mid h_i)/e^{cn_i} \quad \text{(set } c_{h_i} = 0)
\]
Simple Negative Sampling
(Mikolov 2012)

• Used in word2vec

• Basically, sample one positive $k$ negative examples, calculate the log probabilities

$$P(d = 1 \mid x_i, h_i) = \frac{P(x_i \mid h_i)}{P(x_i \mid h_i) + 1}$$

• Similar to NCE, but biased when $k \neq |V|$ or $Q$ is not uniform
Mini-batching Negative Sampling

• Creating and arranging memory on the is expensive, especially on the GPU

• **Simple solution:** select the same negative samples for each minibatch

• (See Zoph et al. 2015 for details)
Let’s Try it Out!

wordemb-negative-sampling.py
Structure-based Softmax Approximations
Structure-based Approximations

• We can also change the structure of the softmax to be more efficiently calculable

  • Class-based softmax
  • Hierarchical softmax
  • Binary codes
Class-based Softmax
(Goodman 2001)

- Assign each word to a class
- Predict class first, then word given class

\[
P(c|h) = \text{softmax}(W_c h + b_c)
\]

\[
P(x|c,h) = \text{softmax}(W_x h + b_x)
\]

**Quiz:** What is the computational complexity?
Hierarchical Softmax
(Morin and Bengio 2005)

• Create a tree-structure where we make one decision at every node

Quiz: What is the computational complexity?

0 1 1 1 0 → word 14
Binary Code Prediction
(Dietterich and Bakiri 1995, Oda et al. 2017)

- Choose all bits in a single prediction

\[ \sigma(W_c h + b_c) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \]

- Simpler to implement and fast on GPU
Let’s Try it Out!

wordemb-binary-code.py
Two Improvement to Binary Code Prediction

**Hybrid Model**

- **(a) Softmax prediction (conventional)**
  - $h \xrightarrow{\text{Softmax}} \arg\max\ O(HV)

- **(b) Binary code prediction**
  - $h \xrightarrow{\text{Sigmoid}} \text{Returns corresponding word} \quad O(H \log V)

- **(c) Hybrid prediction (softmax & binary code)**
  - $h' \xrightarrow{\text{Softmax, Sigmoid}} q \quad O(HN)\{O(H \log V)\}$

**Error Correcting Codes**

- **(a) Training**
  - $b' \xrightarrow{\text{Redundancy}} b \xrightarrow{\text{Encode}} w \xrightarrow{\text{Decode (Absorbs bit errors)}} q$

- **(b) Generating**
  - $\tilde{q} \xrightarrow{\text{Decode}} w$
Parallelism in Computation Graphs
Three Types of Parallelism

- Within-operation parallelism
- Operation-wise parallelism
- Example-wise parallelism

Model parallelism

Data parallelism
Within-operation Parallelism

- GPUs excel at this!
- Libraries like MKL implement this on CPU, but gains less striking.
- Thread management overhead is counter-productive when operations small.
Operation-wise Parallelism

- Split each operation into a different thread, or different GPU device

**Difficulty**: How do we minimize dependencies and memory movement?
Example-wise Parallelism

- Process each training example in a different thread or machine

<table>
<thead>
<tr>
<th>this is an example</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>this is another example</td>
<td>Thread 2</td>
</tr>
<tr>
<td>this is the best example</td>
<td>Thread 3</td>
</tr>
<tr>
<td>no, i’m the best example</td>
<td>Thread 4</td>
</tr>
</tbody>
</table>

- **Difficulty**: How do we accumulate gradients and keep parameters fresh across machines?
GPU Training Tricks
GPUs vs. CPUs

CPU, like a motorcycle
Quick to start, top speed, not shabby

GPU, like an airplane
Takes forever to get off the ground, but super-fast once flying

A Simple Example

- How long does a matrix-matrix multiply take?
Practically

• Use **CPU for profiling**, it’s plenty fast (esp. DyNet) and you can run many more experiments

• For **many applications, CPU is just as fast** or faster than GPU: NLP analysis tasks with small or complicated data/networks

• You see **big gains on GPU when** you have:
  
  • Very big networks (or softmaxes with no approximation)
  
  • Do mini-batching

  • Optimize things properly
Speed Trick 1: Don’t Repeat Operations

• Something that you can do once at the beginning of the sentence, don’t do it for every word!

**Bad**

```python
for x in words_in_sentence:
    vals.append( W * c + x )
```

**Good**

```python
W_c = W * c
for x in words_in_sentence:
    vals.append( W_c + x )
```
Speed Trick 2: Reduce # of Operations

- e.g. can you combine multiple matrix-vector multiplies into a single matrix-matrix multiply? Do so!

**Bad**

```python
for x in words_in_sentence:
    vals.append(W * x)
val = dy.concatenate(vals)
```

**Good**

```python
X = dy.concatenate_cols(words_in_sentence)
val = W * X
```

- (DyNet’s auto-batching does this for you (sometimes))
Speed Trick 3: Reduce CPU-GPU Data Movement

• Try to **avoid memory moves** between CPU and GPU.

• When you do move memory, try to do it as early as possible (GPU operations are asynchronous)

**Bad**
```
for x in words_in_sentence:
    # input data for x
    # do processing
```

**Good**
```
# input data for whole sentence
for x in words_in_sentence:
    # do processing
```
What About Memory?

- Most GPUs only have up to 12GB, so memory is a major issue.
- Minimize unnecessary operations, especially ones over big pieces of data.
- If absolutely necessary, use multiple GPUs (but try to minimize memory movement).
Let’s Try It!

slow-impl.py
Questions?