Natural Language Processing with Deep Learning

Word Vectors
How do we represent the meaning of a word?

Definition: **meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Commonest linguistic way of thinking of meaning:

```
signifier (symbol) ⇔ signified (idea or thing)
```

\[= \text{denotational semantics}\]

cf. connotational semantics — implied not literal
How do we have usable meaning in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** (“is a” relationships).

**e.g. synonym sets containing “good”:**

```python
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}\n".format(poses[synset.pos()],
    ", ".join([l.name() for l in synset.lemmas()])))
```

- noun: good
- noun: good, goodness
- noun: commodity, trade_good, good
- adj: good
- adj (sat): full, good
- adj: good
- adj (sat): estimable, good, honorable, respectable
- adj (sat): beneficial, good
- adj (sat): good
- adj (sat): good, just, upright
- ...
- adverb: well, good
- adverb: thoroughly, soundly, good

**e.g. hypernyms of “panda”:**

```python
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

- [Synset('procyonid.n.01'), Synset('carnivore.n.01'), Synset('placental.n.01'), Synset('mammal.n.01'), Synset('vertebrate.n.01'), Synset('chordate.n.01'), Synset('animal.n.01'), Synset('organism.n.01'), Synset('living_thing.n.01'), Synset('whole.n.02'), Synset('object.n.01'), Synset('physical_entity.n.01'), Synset('entity.n.01')]
Problems with resources like WordNet

• Great as a resource but missing nuance
  • e.g. “proficient” is listed as a synonym for “good”. This is only correct in some contexts.

• Missing new meanings of words
  • e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
  • Impossible to keep up-to-date!

• Subjective

• Requires human labor to create and adapt

• Can’t compute accurate word similarity →
Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols: hotel, conference, motel — a localist representation

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000)
Problem with words as discrete symbols

Example: in web search, if user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”.

But:

\[
\text{motel} = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] \\
\text{hotel} = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
\]

These two vectors are orthogonal.
There is no natural notion of similarity for one-hot vectors!

Solution:

- Could try to rely on WordNet’s list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- Instead: learn to encode similarity in the vectors themselves
Representing words by their context

- **Distributional semantics:** A word’s meaning is given by the words that frequently appear close-by
  - “You shall know a word by the company it keeps” (J. R. Firth 1957: 11)
  - One of the most successful ideas of modern statistical NLP!

- When a word $w$ appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).

- Use the many contexts of $w$ to build up a representation of $w$

---

...government debt problems turning into **banking** crises as happened in 2009...
...saying that Europe needs unified **banking** regulation to replace the hodgepodge...
...India has just given its **banking** system a shot in the arm...

These context words will represent **banking**
Word vectors

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts.

\[
\text{banking} = \begin{pmatrix}
0.286 \\
0.792 \\
-0.177 \\
-0.107 \\
0.109 \\
-0.542 \\
0.349 \\
0.271
\end{pmatrix}
\]

Note: word vectors are sometimes called word embeddings or word representations. They are a distributed representation.
Word meaning as a neural word vector – visualization

\[
\text{expect} = \begin{bmatrix}
0.286 \\
0.792 \\
-0.177 \\
-0.107 \\
0.109 \\
-0.542 \\
0.349 \\
0.271 \\
0.487
\end{bmatrix}
\]
Word2vec: Overview

Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

Idea:
• We have a large corpus of text
• Every word in a fixed vocabulary is represented by a vector
• Go through each position $t$ in the text, which has a center word $c$ and context (“outside”) words $o$
• Use the similarity of the word vectors for $c$ and $o$ to calculate the probability of $o$ given $c$ (or vice versa)
• Keep adjusting the word vectors to maximize this probability
Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$
Word2vec: objective function

For each position \( t = 1, \ldots, T \), predict context words within a window of fixed size \( m \), given center word \( w_j \).

\[
\text{Likelihood} = L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w_{t+j} | w_t; \theta)
\]

\( \theta \) is all variables to be optimized

The objective function \( J(\theta) \) is the (average) negative log likelihood:

\[
J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j} | w_t; \theta)
\]

Minimizing objective function \( \iff \) Maximizing predictive accuracy
Word2vec: objective function

- We want to minimize the objective function:

\[
J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j} | w_t; \theta)
\]

- **Question:** How to calculate \( P(w_{t+j} | w_t; \theta) \) ?

- **Answer:** We will use two vectors per word \( w \):
  - \( v_w \) when \( w \) is a center word
  - \( u_w \) when \( w \) is a context word

- Then for a center word \( c \) and a context word \( o \):

\[
P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}
\]
Word2vec: prediction function

\[ P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)} \]

- Exponentiation makes anything positive
- Dot product compares similarity of \( o \) and \( c \).
  \[ u^T v = u.v = \sum_{i=1}^{n} u_i v_i \]
- Larger dot product = larger probability
- Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function** \( \mathbb{R}^n \to \mathbb{R}^n \)
  \[ \text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i \]

- The softmax function maps arbitrary values \( x_i \) to a probability distribution \( p_i \)
  - “max” because amplifies probability of largest \( x_i \)
  - “soft” because still assigns some probability to smaller \( x_i \)
  - Frequently used in Deep Learning
Training a model by optimizing parameters

To train a model, we adjust parameters to minimize a loss
E.g., below, for a simple convex function over two parameters
Contour lines show levels of objective function

\[ z = x^2 + 2y^2 \]
To train the model: Compute all vector gradients!

- Recall: $\theta$ represents all model parameters, in one long vector.
- In our case with $d$-dimensional vectors and $V$-many words:
  \[
  \theta = \begin{bmatrix}
  v_{aardvark} \\
  v_a \\
  \vdots \\
  v_{zebra} \\
  u_{aardvark} \\
  u_a \\
  \vdots \\
  u_{zebra}
  \end{bmatrix} \in \mathbb{R}^{2dV}
  \]

- Remember: every word has two vectors.
- We optimize these parameters by walking down the gradient.
Word2vec derivations of gradient

- The basic Lego piece
- Useful basics: $\frac{\partial x^T a}{\partial x} = \frac{\partial a^T x}{\partial x} = a$
- If in doubt: write out with indices
- Chain rule! If $y = f(u)$ and $u = g(x)$, i.e. $y = f(g(x))$, then:
  $$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$
Let’s derive gradient for center word together
For one example window and one example outside word:

\[
J(\theta) = \frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log p(w_{t+j} | w_t)
\]

\[
\log p(o|c) = \log \frac{\exp(u_o^T \nu_c)}{\sum_{w=1}^{V} \exp(u_w^T \nu_c)}
\]

You then also need the gradient for context words (it’s similar; left for homework). That’s all of the parameters \( \theta \) here.
Calculating all gradients!

- We went through gradient for each center vector $v$ in a window.
- We also need gradients for outside vectors $u$.

- Generally in each window we will compute updates for all parameters that are being used in that window. For example:

\[
\begin{align*}
P(u_{\text{turning}} | v_{\text{banking}}) & \quad P(u_{\text{into}} | v_{\text{banking}}) & \quad P(u_{\text{crises}} | v_{\text{banking}}) \\
& \quad P(u_{\text{as}} | v_{\text{banking}})
\end{align*}
\]
Word2vec: More details

Why two vectors? \(\rightarrow\) Easier optimization. Average both at the end.

Two model variants:

1. Skip-grams (SG)
   Predict context ("outside") words (position independent) given center word

2. Continuous Bag of Words (CBOW)
   Predict center word from (bag of) context words

This lecture so far: **Skip-gram model**

Additional efficiency in training:

1. Negative sampling
   pair of sigmoid for positive sample vs \(-\)negative sample

   So far: Focus on **naïve softmax** (simpler training method)
Optimization: Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
- **Gradient Descent** is an algorithm to minimize $J(\theta)$
- **Idea**: for current value of $\theta$, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient. Repeat.

Note: Our objectives may not be convex like this :(  

![Graph showing optimization process with gradient descent](image-url)
Gradient Descent

- Update equation (in matrix notation):

\[
\theta_{new} = \theta_{old} - \alpha \nabla_{\theta} J(\theta)
\]

\(\alpha = \text{step size or learning rate}\)

- Update equation (for single parameter):

\[
\theta_{j, new} = \theta_{j, old} - \alpha \frac{\partial}{\partial \theta_{j, old}} J(\theta)
\]

- Algorithm:

```python
while True:
    theta_grad = evaluate_gradient(J, corpus, theta)
    theta = theta - alpha * theta_grad
```
Stochastic Gradient Descent

- **Problem:** $J(\theta)$ is a function of all windows in the corpus (potentially billions!)
  - So $\nabla_\theta J(\theta)$ is very expensive to compute
- You would wait a very long time before making a single update!

- **Very** bad idea for pretty much all neural nets!
- **Solution:** Stochastic gradient descent (SGD)
  - Repeatedly sample windows, and update after each one
- Algorithm:

```python
while True:
    window = sample_window(corpus)
    theta_grad = evaluate_gradient(J, window, theta)
    theta = theta - alpha * theta_grad
```

*mini-batch: $2^N$ step update*