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:

\[
\text{signifier (symbol)} \leftrightarrow \text{signified (idea or thing)}
\]

= 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('{}: {}'.format(poses[synset.pos()],
       ', '.join([l.name() for l in synset.lemmas()]))
```

noun: good
noun: good, goodness
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')]
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.

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

banking =

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

\[
\text{expect} = \begin{pmatrix}
0.286 \\
0.792 \\
-0.177 \\
-0.107 \\
0.109 \\
-0.542 \\
0.349 \\
0.271 \\
0.487
\end{pmatrix}
\]
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 $P(w_{t+j} | w_t)$
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

sometimes called cost or loss function

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 \( \Leftrightarrow \) 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

Exponentiation makes anything positive

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

Dot product compares similarity of \( o \) and \( c \).

\[ u^T v = u \cdot v = \sum_{i=1}^{n} u_i v_i \]

Larger dot product = larger probability

Normalize over entire vocabulary to give probability distribution

- This is an example of the softmax function \( \mathbb{R}^n \rightarrow \mathbb{R}^n \)

\[ \text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i \]

- 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
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: More details

Why two vectors? → 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

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 :(

![Diagram of Gradient Descent](image)
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
```
Natural Language Processing with Deep Learning

Word Window Classification and Neural Networks
Classification setup and notation

• Generally we have a training dataset consisting of samples

\[ \{x_i, y_i\}_{i=1}^{N} \]

• \(x_i\) are inputs, e.g. words (indices or vectors!), sentences, documents, etc.
  • Dimension \(d\)

• \(y_i\) are labels (one of \(C\) classes) we try to predict, for example:
  • classes: sentiment, named entities, buy/sell decision
  • other words
  • later: multi-word sequences
Classification intuition

• Training data: \( \{x_i, y_i\}_{i=1}^N \)

• Simple illustration case:
  • Fixed 2D word vectors to classify
  • Using softmax/logistic regression
  • Linear decision boundary

• Traditional ML/Stats approach: assume \( x_i \) are fixed, train (i.e., set) softmax/logistic regression weights \( W \in \mathbb{R}^{C \times d} \) to determine a decision boundary (hyperplane) as in the picture

• Method: For each \( x \), predict:

\[
p(y|x) = \frac{\exp(W_{y,x})}{\sum_{c=1}^{C} \exp(W_{c,x})}
\]
Training with softmax and cross-entropy loss

• For each training example \((x, y)\), our objective is to \text{maximize the probability of the correct class} \(y\)

• Or we can \text{minimize the negative log probability of that class}:

\[
- \log p(y|x) = - \log \left( \frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)} \right)
\]
Background: What is “cross entropy” loss/error?

- Concept of “cross entropy” is from information theory
- Let the true probability distribution be $p$
- Let our computed model probability be $q$
- The cross entropy is:

$$H(p, q) = - \sum_{c=1}^{C} p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: $p = [0,...,0,1,0,...0]$ then:
- **Because of one-hot $p$, the only term left is the negative log probability of the true class**
Classification over a full dataset

- Cross entropy loss function over full dataset \{x_i, y_i\}_{i=1}^N

\[
J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left( - \log \left( \frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}} \right) \right)
\]

- Instead of

\[
f_y = f_y(x) = W_y.x = \sum_{j=1}^{d} W_{yj}x_j
\]

We will write \( f \) in matrix notation:

\[
f = Wx
\]
Traditional ML optimization

• For general machine learning $\theta$ usually only consists of columns of $W$:

$$\theta = \begin{bmatrix} W_1 \\ \vdots \\ W_d \end{bmatrix} = W(:, \in \mathbb{R}^{C_d})$$

• So we only update the decision boundary via

$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \nabla W_1 \\ \vdots \\ \nabla W_d \end{bmatrix} \in \mathbb{R}^{C_d}$$

Visualizations with ConvNetJS by Karpathy
Classification difference with word vectors

Commonly in NLP deep learning:

- We learn **both** $W$ and word vectors $x$
- We learn **both** conventional parameters and representations
- The word vectors re-represent one-hot vectors—move them around in an intermediate layer vector space—for easy classification with a (linear) softmax classifier via layer $x = Le$

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla W.1 \\ \vdots \\ \nabla W.d \\ \nabla x_{aardvark} \\ \vdots \\ \nabla x_{zebra} \end{bmatrix} \in \mathbb{R}^{C_d + V_d}$$

Very large number of parameters!
An artificial neuron

- Neural networks come with their own terminological baggage
- But if you understand how softmax models work, then you can easily understand the operation of a neuron!
A neuron can be a binary logistic regression unit

\[ h_{w,b}(x) = f(w^T x + b) \]

\[ f(z) = \frac{1}{1 + e^{-z}} \]

\( w, b \) are the parameters of this neuron i.e., this logistic regression model

\( b \): We can have an “always on” feature, which gives a class prior, or separate it out, as a bias term.

\( f = \) nonlinear activation fct. (e.g. sigmoid), \( w = \) weights, \( b = \) bias, \( h = \) hidden, \( x = \) inputs
A neural network = running several logistic regressions at the same time

... which we can feed into another logistic regression function

It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.
Matrix notation for a layer

We have

\[ a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1) \]
\[ a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2) \]

etc.

In matrix notation

\[ z = Wx + b \]
\[ a = f(z) \]

Activation $f$ is applied element-wise:

\[ f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)] \]
Non-linearities (aka “\( f \)”): Why they’re needed

- Example: function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can’t do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: \( W_1 W_2 x = Wx \)
  - With more layers, they can approximate more complex functions!
The European Commission [ORG] said on Thursday it disagreed with German [MISC] advice.

Only France [LOC] and Britain [LOC] backed Fischler [PER] 's proposal.

“What we have to be extremely careful of is how other countries are going to take Germany 's lead”, Welsh National Farmers ' Union [ORG] ( NFU [ORG] ) chairman John Lloyd Jones [PER] said on BBC [ORG] radio.

Possible purposes:

- Tracking mentions of particular entities in documents
- For question answering, answers are usually named entities
- A lot of wanted information is really associations between named entities
- The same techniques can be extended to other slot-filling classifications
- Often followed by Named Entity Linking/Canonicalization into Knowledge Base
Named Entity Recognition on word sequences

We predict entities by classifying words in context and then extracting entities as word subsequences.

Foreign ORG
Ministry ORG
spokesman O
Shen PER
Guofang PER
told O
Reuters ORG
that O
:
:

B-ORG
I-ORG
O
B-PER
I-PER
O
B-ORG
O

拇指 BIO encoding
Why might NER be hard?

- Hard to work out boundaries of entity
  
  Is the first entity “First National Bank” or “National Bank”

- Hard to know if something is an entity
  
  Is there a school called “Future School” or is it a future school?

- Hard to know class of unknown/novel entity:
  
  What class is “Zig Ziglar”? (A person.)

- Entity class is ambiguous and depends on context
  
  “Charles Schwab” is PER not ORG here! 👉
Binary word window classification

• In general, classifying single words is rarely done

• Interesting problems like ambiguity arise in context!

• Example: auto-antonyms:
  • "To sanction" can mean "to permit" or "to punish"
  • "To seed" can mean "to place seeds" or "to remove seeds"

• Example: resolving linking of ambiguous named entities:
  • Paris → Paris, France vs. Paris Hilton vs. Paris, Texas
  • Hathaway → Berkshire Hathaway vs. Anne Hathaway
Window classification

• **Idea**: classify a word in its context window of neighboring words.

• For example, **Named Entity Classification** of a word in context:
  • Person, Location, Organization, None

• A simple way to classify a word in context might be to average the word vectors in a window and to classify the average vector
  • Problem: that would lose position information
Window classification: Softmax

- Train softmax classifier to classify a center word by taking concatenation of word vectors surrounding it in a window

- **Example**: Classify “Paris” in the context of this sentence with window length 2:

  \[ X_{\text{window}} = \begin{bmatrix} x_{\text{museums}} & x_{\text{in}} & x_{\text{Paris}} & x_{\text{are}} & x_{\text{amazing}} \end{bmatrix}^T \]

- Resulting vector \( x_{\text{window}} = x \in \mathbb{R}^{5d} \), a column vector!
Simplest window classifier: Softmax

- With $x = x_{\text{window}}$ we can use the same softmax classifier as before

$$\hat{y}_y = p(y|x) = \frac{\exp(W_{y,x})}{\sum_{c=1}^{C} \exp(W_{c,x})}$$

- With cross entropy error as before:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} - \log \left( \frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}} \right)$$

- How do you update the word vectors?
- Short answer: Just take derivatives like last week and optimize
Binary classification with unnormalized scores

Method used by Collobert & Weston (2008, 2011)

- Just recently won ICML 2018 Test of time award

- For our previous example:
  \[ X_{\text{window}} = [ x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}} ] \]

- Assume we want to classify whether the center word is a Location

- Similar to word2vec, we will go over all positions in a corpus. But this time, it will be supervised and only some positions should get a high score.

- E.g., the positions that have an actual NER Location in their center are “true” positions and get a high score
Binary classification for NER Location

• Example: Not all museums in Paris are amazing.
• Here: one true window, the one with Paris in its center and all other windows are “corrupt” in terms of not having a named entity location in their center.

  museums in Paris are amazing

• “Corrupt“ windows are easy to find and there are many: Any window whose center word isn’t specifically labeled as NER location in our corpus

  Not all museums in Paris
Neural Network Feed-forward Computation

Use neural activation $a$ simply to give an unnormalized score

$$\text{score}(x) = U^T a \in \mathbb{R}$$

We compute a window’s score with a 3-layer neural net:

1. $s = \text{score}("museums in Paris are amazing")$

$$s = U^T f(Wx + b)$$

$$x \in \mathbb{R}^{20 \times 1}, W \in \mathbb{R}^{8 \times 20}, U \in \mathbb{R}^{8 \times 1}$$

\[
\begin{align*}
    z &= Wx + b \\
    a &= f(z) \\
    s &= U^T a
\end{align*}
\]

$x_{\text{window}} = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}}]$
Main intuition for extra layer

The middle layer learns **non-linear interactions** between the input word vectors.

\[
X_{\text{window}} = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}}]
\]

**Example:** only if “museums” is first vector should it matter that “in” is in the second position.
The max-margin loss

- Idea for training objective: Make true window’s score larger and corrupt window’s score lower (until they’re good enough)

- \( s = \text{score(museums in Paris are amazing)} \)

- \( s_c = \text{score(Not all museums in Paris)} \)

- Minimize

\[
J = \max(0, 1 - s + s_c)
\]

- This is not differentiable but it is continuous → we can use SGD.
Max-margin loss

- Objective for a single window:

\[ J = \max(0, 1 - s + s_c) \]

- Each window with an NER location at its center should have a score +1 higher than any window without a location at its center

- For full objective function: Sample several corrupt windows per true one. Sum over all training windows.

- Similar to negative sampling in word2vec
Simple net for score

\[ s = u^T h \]

\[ h = f(Wx + b) \]

\[ x \quad \text{(input)} \]

\[ x = [ x_{\text{museums}}, x_{\text{in}}, x_{\text{paris}}, x_{\text{are}}, x_{\text{amazing}} ] \]
Remember: Stochastic Gradient Descent

- Update equation:

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

\[ \alpha = \text{step size or learning rate} \]

- How do we compute \( \nabla_{\theta} J(\theta) \)?
  - By hand
  - Algorithmically: the backpropagation algorithm