Stanford CS224N: Course Notes

Vikram Rawal

June 2023

1 Introduction and Word Vectors

1.1 The course

The course aims to provide an introduction to the following three topics:

1. The foundations of the effective modern methods for deep learning applied to NLP (*natural language processing*). The course starts from the basics and then delves into the key methods used in NLP: Recurrent networks, attention, transformers, etc.

2. A big-picture understanding of human languages and the difficulties in understanding and producing them.

3. An understanding of and ability to build systems (in `PyTorch`) for some of the major problems in NLP: Word meaning, dependency parsing, machine translation, question answering.

Language is complicated, and it's a more recent development then we might imagine it to be. Bridging the gap between human and computer languages might sound unromantic, but one can imagine that the zenith of NLP would be marked by humans seamlessly communicating their purest possible forms of thought, using computers and algorithms to implant their musings directly into the mind of a correspondent. There's something beautiful in that, even if it's nothing like the romance in languages we see and use today.

1.2 Human Language

Humans owe their ascendancy over other species largely to their newfound (in the grand scheme of the universe) ability to *communicate with each other*. The ability to share and to pass down knowledge between people, communities, and generations has taken us from the Stone Age to the Digital Age in under five thousand years. Today, ChatGPT is the first step towards the idea of a *universal model*—a model which is trained on large corpora of text to learn and emulate human communication, and which recognize patterns and turn human language into machine code, among other impressive feats.

1.3 Word Meaning

Meaning is defined as ...

- 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.

The commonest linguistic interpretation of meaning is that of *denotational semantics*, where

$$\text{signifier } (\textbf{symbol}) \Leftrightarrow \text{signified } \textbf{(idea or thing)} \tag{1}$$

That is, where we can *map individual* symbols (words) to *individual* ideas or things. For example, a chair maps to the set of chairs and other ideas which we ascribe to the term.

A common NLP solution to the problem of instilling *usable meanings* of words in a computer is a database like WordNet, which comprises lists of **synonym sets** and **hypernyms** ("is a" relationships). **Synonym sets** are sets of words which denote the same idea or thing, while **hypernyms** of a given word a are words b_i which denote a *superset* of the set of ideas and things denoted by a. In other words, if b_i is a hypernym of a, it would be logical to say that "a is a b_i," as in "a *panda* is a *mammal*."

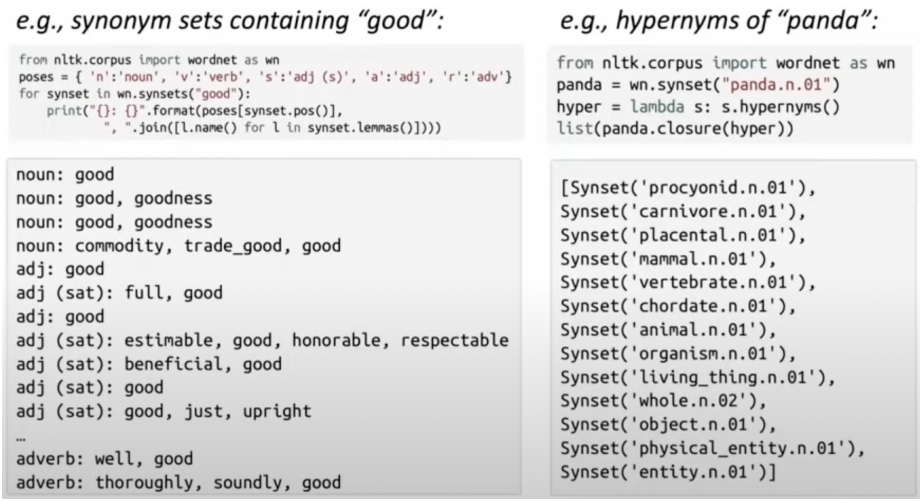


Figure 1: Synonym Sets and Hypernyms

While this is a useful tool and a step in the right direction, it lacks the nuance necessary to interpret words' varying meanings within the contexts in which they are used. For instance, "proficient" can be listed as a synonym for "good", but it would be incorrect to replace the "good" in "He's a good man" with "proficient". Furthermore, WordNet doesn't know the meanings of new words, and thus requires human intervention to remain up-to-date. And while WordNet looks at synonyms, it has no bearing on how semantically similar two words are in the event that they aren't synonyms.

1.3.1 Representing Words as Discrete Symbols

One possible solution to represent words as symbols is assigning each word a discrete vector (a *localist* representation) using *one-hot encoding*. Symbols for words can be represented by one-hot vectors, where the length of the vector is the number of words in the dictionary:

$$\text{hotel} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$
$$\text{motel} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The problem with this scheme, of course, is that vectors which aren't equal to each other are *orthogonal* to each other—so there is no concept of semantic similarity! Relying on WordNet for semantic similarity would be clunky and inconvenient. **Solution:** Let's try to encode the notion similarity *into the vectors themselves.*

1.3.2 Representing Words by their Context

The alternative to representing each word (symbol) with a corresponding vector is to appeal to the idea of **distributional semantics**, which dictates that *a word's meaning is given by the words that frequently appear close-by.* This is 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).
- We can use the many contexts of w to build up a *representation* of w.

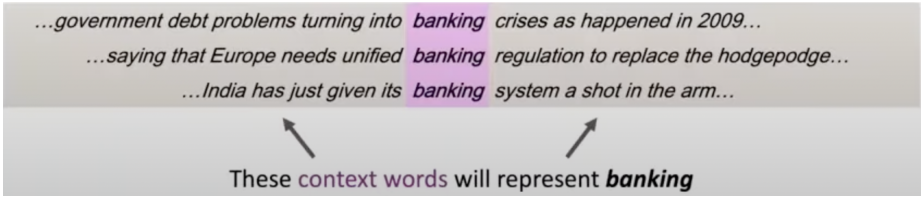


Figure 2: Contexts of "Banking"

1.4 Word Vectors

The alternative to representing words as discrete symbols is to use **word vectors**, or **word embeddings**. Essentially, these are real valued vectors whose positions in a higher-dimensional vector space are such that "words that are closer together in the vector space are expected to be similar in meaning." For example:

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

These word embeddings are a *distributed representation* of the meaning of words, as the meaning is "distributed" over the 300 dimensions of the higher-dimensional vector space. For example:

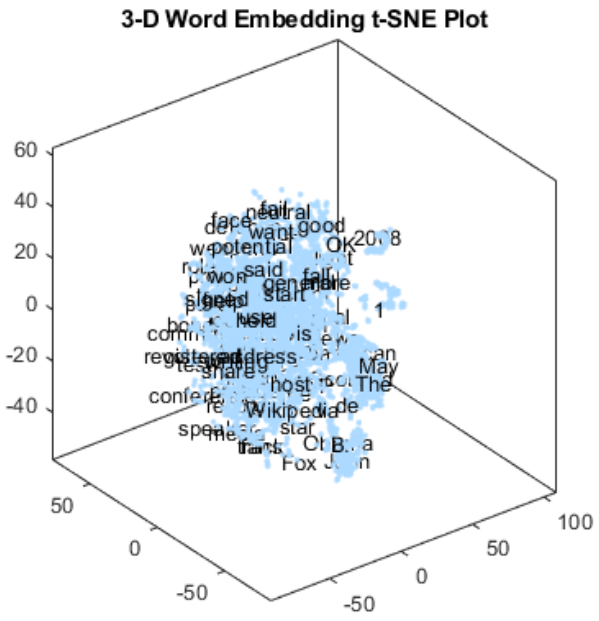


Figure 3: 3D Projection of Higher-Dimensional Vector Space

1.5 Word2vec

Word2vec is a framework for learning word vectors. The idea is as follows:

- We have a large corpus ("body") 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.

For instance, below is the procedure for computing $P(w_{t+j} \mid w_t)$, where $j \in (-2, 2)$ represents a window of size 2:

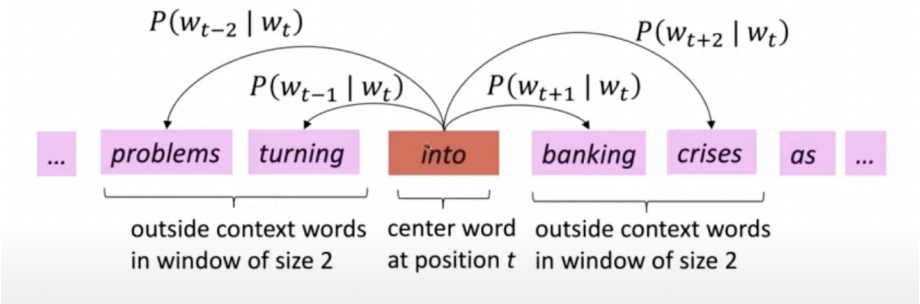


Figure 4: Computing $P(w_{t+j} \mid w_t)$

Here, "into" is our center word, w_t, at position t. We want to modify values for P("problems" | "into"), P("turning" | "into"), P("banking" | "into"), and P("crises" | "into"). Given that we've seen "into", what is the probability that we see each of the context words in its vicinity? After we've modified these probabilities accordingly, "banking" becomes our new w_t, and we iterate over the corpus of text to find all $P(w_{t+j} \mid w_t)$ for all t.

1.5.1 Objective Function

The model we want to use is the minimization of an *objective function*, which is equivalent to maximizing a data likelihood function:

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta) \tag{2}$$

Where θ represents the variables that are to be optimized—the objective function is simply a joint PDF with parameter θ. Recall that our goal is to determine the value of the vector (or tensor) θ such that the observed contexts over all center words in the corpus of text have the maximum possible joint probability. Essentially, for every word in our corpus of text, we are multiplying together the probabilities that the words appearing in its window do, in fact, appear in its window.

These probabilities depend on the tensor θ, so we can alternatively write the likelihood function for a given t as follows:

$$L(w_{t-j}...w_{t+j}; \theta) = f(w_{t-j}...w_{t+j}; \theta) \tag{3}$$

$$= g(w_{t-j}; \theta) g(w_{t-j+1}; \theta) ... g(w_{t+j}; \theta) \tag{4}$$

,

Where $g(w_i; \theta)$ is the probability density function of the word at position i. Note that we're assuming that the appearances of the words are independent of each other, so that the total probability of seeing all the words in the window is the product of $g(w_i; \theta)$ for all $i \in (t-j, t+j \mid j \neq 0)$.

In this way, we can maximize the likelihoods of the contexts we see around each center word. Since it's easier to work with sums than it is to work with products, we define the *objective function* $J(\theta)$ as the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta) \tag{5}$$

Minimizing the objective function allows us to *maximize* the predictive accuracy of our model. Of course, this begs the question: **How exactly are we to calculate $P(w_{t+j} \mid w_t; \theta)$?**

1.5.2 Prediction Function

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:

$$P(o \mid c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \tag{6}$$

What is equation (6) doing? Well, $\exp\left(u_o^T v_c\right)$ is simply the exponential of the *dot product* of vectors u_o and v_c, since

$$\begin{bmatrix} u_{o1} & u_{o2} & u_{o3} & \dots & u_{on} \end{bmatrix} \cdot \begin{bmatrix} v_{c1} \\ v_{c2} \\ v_{c3} \\ \vdots \\ v_{cn} \end{bmatrix} = \sum_{i=1}^{n} u_{oi} v_{ci} = u_o \cdot v_c \tag{7}$$

We are taking the dot product of u_o, the context word vector, and v_c, the center word vector, to determine the similarity between o and c. The dot product is maximum when the vectors are parallel (ie: they are equal) and is minimum when the vectors are perpendicular (ie: the words bear no resemblance with each other). We then normalize by dividing by the sum of the exponentials of the dot products of the center word with *all* context words present in the corpus of text.

A few key points:

- We are exponentiating because we require that all probabilities are *positive.*
- Dividing by $\sum_{w \in V} \exp\left(u_w^T v_c\right)$ simply normalizes the numerator and provides us with a probability density function, as summing the numerator over all possible context words u_o simply yields the denominator, and so the function integrates to 1.

Note that this is an example of the *softmax function* $\mathbb{R}^n \to (0,1)^n$:

$$\text{softmax}(x_i) = \frac{\exp\left(x_i\right)}{\sum_{j=1}^{n} \exp\left(x_j\right)} \tag{8}$$

which maps arbitrary values x_i to return a probability distribution p_i:

- "max" because it amplifies the probability of the largest x_i
- "soft" because it still assigns some probability to smaller x_i.
- This function is frequently used in deep learning.

1.5.3 Training the Model: Optimizing Parameters to Minimize Loss

To train the model, we want to gradually adjust the parameters to minimize loss. Recall that θ represents *all* the model parameters in one long vector. Therefore, with $d-$dimensional vectors and V words with 2 vectors each in our corpus of text, we have

$$\theta = \begin{bmatrix} v_{aardvark} \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV} \tag{9}$$

As the $2V$ word vectors and their d dimensions each are all orthogonal, we are working in $\mathbb{R}^{2dV}$. We optimize these parameters by computing *all* vector gradients and "walking down" the gradient (gradient descent) to minimize the objective function. Recall the objective function $J(\theta)$ and the prediction function $P(o \mid c)$:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

$$P(o \mid c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)}$$

Now, we want to work out the *gradient* (the direction of steepest descent) for the objective function. We do this by finding its partial derivative with respect to every parameter in the model—that is, with respect to every center word vector v_w and every context word vector u_w. For the center words:

$$\begin{aligned}\frac{dJ(\theta)}{dv_c} &= \frac{d}{dv_c}\left(-\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m\\ j\neq 0}}\log\frac{\exp\left(u_{t+j}^T v_t\right)}{\sum_{w\in V}\exp\left(u_w^T v_t\right)}\right)\\ &= -\frac{1}{T}\sum_{\substack{-m\leq j\leq m\\ j\neq 0}}\frac{d}{dv_c}\left(\log\frac{\exp\left(u_{c+j}^T v_c\right)}{\sum_{w\in V}\exp\left(u_w^T v_c\right)}\right)\end{aligned}$$

Evaluating the partial derivative now amounts to evaluating, for each center word c in the corpus and for each j in the window:

$$\begin{aligned}&\frac{d}{dv_c}\left(\log\frac{\exp\left(u_{c+j}^T v_c\right)}{\sum_{w\in V}\exp\left(u_w^T v_c\right)}\right)\\ &= \frac{d}{dv_c}\left(\log\exp\left(u_{c+j}^T v_c\right) - \log\sum_{w\in V}\exp\left(u_w^T v_c\right)\right)\\ &= \frac{d}{dv_c}\left(u_{c+j}^T v_c\right) - \frac{d}{dv_c}\left(\log\sum_{w\in V}\exp\left(u_w^T v_c\right)\right)\\ &= u_{c+j} - \frac{1}{\sum_{w\in V}\exp\left(u_w^T v_c\right)}\cdot\sum_{x\in V}u_x\exp\left(u_x^T v_c\right)\end{aligned}$$

Note that the final line reindexes the derivative of the sum in the log term of the line before, so as to clarify that we are iterating over V a separate time. We finally have that

$$\frac{d}{dv_c}\left(\log P(u_{c+j}\mid c)\right) = u_{c+j} - \frac{\sum_{x=1}^{V}u_x\exp\left(u_x^T v_c\right)}{\sum_{w=1}^{V}\exp\left(u_w^T v_c\right)} \tag{10}$$

And moving the sum over x outside the fraction:

$$\frac{d}{dv_c}\left(\log P(u_{c+j}\mid c)\right) = u_{c+j} - \sum_{x=1}^{V}\frac{\exp\left(u_x^T v_c\right)}{\sum_{w=1}^{V}\exp\left(u_w^T v_c\right)}u_x \tag{11}$$

We see that the fraction is simply the softmax function $P(x\mid c)$:

$$\frac{d}{dv_c}\left(\log P(u_{c+j}\mid c)\right) = u_{c+j} - \sum_{x=1}^{V}P(x\mid c)u_x \tag{12}$$

Intuitively, this result makes sense. u_{c+j} is the context vector for a given context word of a center word c, and $\sum_{x=1}^{V} P(x \mid c)u_x$ is the *sum over all words* in the corpus of *the product of* each word's probability distribution (given the center word c) and its context vector. In this sense, this term acts as the *expectation* of the vector u_x, and equation (12) can be thought of as $\frac{d}{dv_c}\Big(\log P(u_{c+j} \mid c)\Big) = \text{observed} - \text{expected}$.

1.6 Introduction to Gensim

Since I'm going to be using Python to fiddle around with Word2vec, I thought it prudent to include some of the basic concepts and documentation in Gensim, the go-to Python library for working with semantic vectors.

1.6.1 Core Concepts

```
import pprint
```

The core concepts of Gensim are:

1. Document: some text
2. Corpus: a collection of documents
3. Vector: a mathematically convenient representation of a document
4. Model: an algorithm for transforming vectors from one representation to another

Let's go into a bit more detail.

Document: A document is a string in Python, and can be anything from a Tweet to a book. For example:

```
document = "Human machine interface for lab applications"
```

Corpus: A corpus is a collection of document objects. Corpora serve two roles in Gensim:

1. Input for training a model—the models use the training corpus to look for common themes and topics, and to initialize their internal model parameters.
2. Documents to organize: After training, a topic model can be used to extract topics from documents not seen in the training corpus.

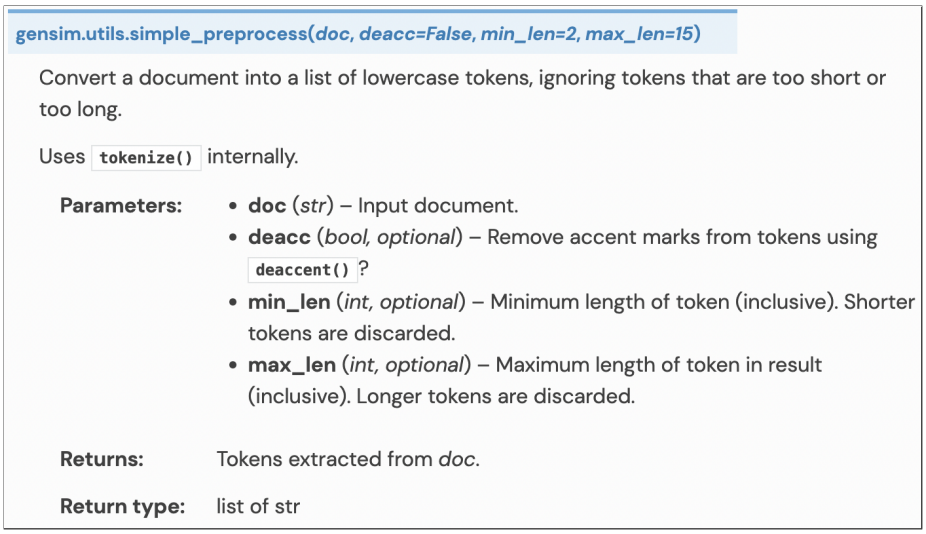
For example:

```
textcorpus = ["Humans rock", "Machines suck", "Dessert rocks"]
```

Is a corpus with three documents.

You can also use corpora without loading them into memory by *streaming* them in, one document at a time. See Corpus Streaming—One Document at a Time for more information.

We can now preprocess our data. There are many ways to go about doing this, but we can start by removing common words (such as 'the') and words that occur only once in the corpus. More complex preprocessing is done using the `simple_preprocess` utility function:



gensim.utils.simple_preprocess(*doc, deacc=False, min_len=2, max_len=15*)

Convert a document into a list of lowercase tokens, ignoring tokens that are too short or too long.

Uses `tokenize()` internally.

Parameters:

- **doc** (*str*) – Input document.
- **deacc** (*bool, optional*) – Remove accent marks from tokens using `deaccent()`?
- **min_len** (*int, optional*) – Minimum length of token (inclusive). Shorter tokens are discarded.
- **max_len** (*int, optional*) – Maximum length of token in result (inclusive). Longer tokens are discarded.

Returns: Tokens extracted from *doc*.

Return type: list of str

Figure 5: Preprocessing with `gensim.utils.simple_preprocess`

```
# Create a set of frequent words
stoplist = set('for a of the and to in'.split(' '))
# Lowercase each document, split it by white space and filter out stopwords
texts = [[word for word in document.lower().split() if word not in stoplist]
         for document in text_corpus]

# Count word frequencies
from collections import defaultdict
frequency = defaultdict(int)
for text in texts:
    for token in text:
        frequency[token] += 1

# Only keep words that appear more than once
processed_corpus = [[token for token in text if frequency[token] > 1] for text in texts]
pprint.pprint(processed_corpus)
```

Figure 6: Basic Preprocessing

Finally, we want to associate each word in the corpus with a unique integer ID. This is done as follows:

```
from gensim import corpora

dictionary = corpora.Dictionary(processed_corpus
print(dictionary)
```

Vectors: We want to represent *documents* in a way such that we can manipulate them mathematically. There are several approaches to doing this:

1. Represent each document as a vector of *features*, which define things like the number of words, frequency of a given word, or number of fonts in the document. For example, $(0, 2, 5)$ can define the answers to questions about three different features. Note that Gensim automatically omits all vector elements with a value of 0.0.

2. Use a bag-of-words model. Each document is represented by a vector consisting of the frequency count of each word in the dictionary. This model can be created using the `doc2bow` method.

Models: These can be thought of as a *transformation* from one document representation to another, and therefore from one vector space to another.

For example, the following code initializes the tf-idf model (which weighs the frequency counts of each word in a given document against the frequency counts of those words in the entire corpus), trains it on the corpus, and transforms the string "system minors":

```
from gensim import models

# train the model
tfidf = models.TfidfModel(bow_corpus)

# transform the "system minors" string
words = "system minors".lower().split()
print(tfidf[dictionary.doc2bow(words)])
```

Figure 7: Initializing and Training the tf-idf model to Transform a String

We can do cool things with word vectors! For example:

```
def analogy(x1, x2, y1):
  result = model.most_similar(positive=[y1, x2], negative = [x1])
  return result[0][0]
```

Figure 8: Analogy function that answers the question "x_1 is to x_2 as y_1 is to what?"

```
<class 'gensim.models.keyedvectors.KeyedVectors'>
> analogy('man', 'woman', 'king')
'queen'
>
```

Figure 9: analogy('man', 'woman', 'king') = 'queen'

2 Neural Classifiers

In this lecture, we'll wrap up Word2vec (incorporating gradient descent) before looking at a different way to capture the essence of word meaning.

2.1 Word2vec Parameters and Computations

Note that the only parameters in the Word2vec model are the outside (u) and center (v) word vectors. We therefore have a *matrix* U consisting of outside column vectors u, and a *matrix* V consisting of center column vectors v. Then, for every one of the n columns in matrix V, we take the dot product $U \cdot v_n^T$ to get a column vector where the m^{th} entry represents the similarity between the m^{th} outside word and the n^{th} center word. We finally apply the softmax function to this vector to get a vector of probabilities that the outside words $u_1 ... u_m$ exist given a center word v_n.

This model is known as the bag-of-words model, and makes the *same* predictions at each position. That is, the probability that an outside word exists in a sentence, given a center word, is the *same* regardless of the position which that outside word might take in the sentence and its distance from the center word. Obviously, this is a rather crude model.

2.2 Optimization: Gradient Descent

To learn good word vectors, we want to minimize a cost function $J(\theta)$, and we can do this by employing *gradient descent*, which essentially involves calculating

the gradient of $J(\theta)$ at the current value of θ, taking a small step in the direction of negative gradient, and repeating this process. The update equation, in matrix notation is as follows:

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J(\theta) \tag{13}$$

And for each single parameter θ_j, this can be written as

$$\theta_j^{\text{new}} = \theta_j^{\text{old}} - \alpha \frac{\partial}{\partial \theta_j^{\text{old}}} J(\theta) \tag{14}$$

The algorithm is therefore as follows:

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

The problem here is that because $J(\theta)$ is a function of *all* windows in the corpus, $\nabla_\theta J(\theta)$ is very computationally expensive. The standard proposed solution is to use **stochastic gradient descent**, where we use a sample of center words and their associated windows (instead of the entire corpus of text) to work out the gradient and to iterate. This would look something like the following:

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

In this case, our $\nabla_\theta J(\theta)$ vector would be very sparse, since we aren't even seeing the majority of the words. We might therefore endeavour to selectively update the rows of U and V that we need, or to keep a *hash* for word vectors (more on this later). We don't want to have to send gigantic updates around, for the sake of system optimization.

Quick note: In actual deep learning packages, word vectors are represented as *row vectors* rather than column vectors, so that accessing a vector is tantamount to accessing a contiguous block of memory, which is much more efficient.

2.3 Word2vec Algorithm Family: More Details

2.3.1 SG vs. CBOW Models

We traditionally use two vectors per word (center and context) so that optimization is easier—we can just average both at the end to get our final word vectors. That being said, we can also implement the Word2vec model with only one vector per word, and this too has its pros.

We just presented the **Skip-grams (SG)** model, which predicts context ("outside") words (position independent) given a center word. The other option

is to use the **Continuous Bag of Words (CBOW)** model. While this model ends up being more accurate, it is also significantly messier to work with, since when a center word is surrounded by the *same* context word we end up with an $\boldsymbol{x} \cdot \boldsymbol{x}$ term, which is quite difficult to navigate around during gradient descent.

2.3.2 Naïve Softmax vs. Negative Sampling (HW2)

We've been using the (naïve) softmax function as our prediction function. While this method is simple and makes intuitive sense, it is computationally expensive. With **negative sampling** (Milokov et al. 2013), the main idea is to train *binary* logistic regressions (using the sigmoid function) for a *true* pair—a center word paired with a word in its context window—versus several *noise* pairs, where the center word is just paired with a random word.

The objective function which is maximized with negative sampling is $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$, where:

$$J_t(\theta) = \log \sigma(u_o^T v_c) + \sum_{i=1}^{k} \mathbb{E}_{j \sim P(w)} \Big[\log \sigma(-u_j^T v_c) \Big] \tag{15}$$

What does this do? Note that σ represents the sigmoid function $\sigma(x) = \frac{1}{1-e^{-x}}$, which essentially maps every real number in the domain of $(0, 1)$ to what is essentially the most likely value of a coin flip (given that x is the probability of flipping heads, for instance). Essentially, if $u_o^T v_c$ is large, then $\sigma(u_o^T v_c)$ will be virtually 1. Then, for a sample of *random* words, we want to *minimize* the probability that they show up around the center word. To this end, for each random word, we take the sigmoid of the *negative* of the dot product (so that smaller dot products have greater probabilities), and add these terms to the first term. In terms which are more similar to the naïve softmax cost function, the above is equivalent to minimizing the following cost function:

$$J_{neg-sample}(\boldsymbol{u}_o, \boldsymbol{v}_c, U) = -\log \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - \sum_{k \in \{K \text{ sampled indices}\}} \log \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c) \tag{16}$$

In brief:

- We take k negative samples (using word probabilities)
- We maximize the probability that the *real* context ("outside") word appears, while minimizing the probability that *random* words appear around the center word.

A trick used in the original Word2vec paper was to sample these random words according to $P(w) = \frac{U(w)^{\frac{3}{4}}}{Z}$, where $U(w)$ is the unigram distribution (which is effectively just the proportion of the total words that is a given word). The power allows less frequent words to be sampled more often than they would be otherwise.

2.4 Co-occurance Matrices

A natural question to ask: Are there any intuitive alternatives to defining word vectors in the way we did? And of course there are. One plausible idea is to create a *co-occurrence matrix*, where we do not have to specify which word is the "center" word and which word is the "context" word. Instead, left and right contexts are equivalent, and the matrix is therefore symmetric. For example, let's use the corpus of:

- "I like deep learning."
- "I like NLP."
- "I enjoy flying."

The co-occurance matrix looks like this:

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Figure 10: Co-occurance Matrix

Where we've defined the window size to be 1 (so that two words are defined to be in each other's contexts every time they're right next to each other (and in the same sentence). We can therefore define words by the row vector they form based on their frequency in the contexts of all words in the corpus. Intuitively, it makes sense that words with similar meanings would have similar-looking row vectors.

When building a co-occurrence matrix X, one can either use windows or look at the whole document (ie: the sentence, paragraph, or webpage) holistically:

- Window: Similar to Word2vec, using a window around each word will capture some syntactic and semantic information.

- Word-Document: Co-occurrence matrix will give general topics (for example, all sports terms will have similar entries), leading to "Latent Semantic Analysis".

Unfortunately, there are some problems with this sort of implementation:

1. The vectors increase in size with vocabulary, and are very high-dimensional. Furthermore, they're quite sparse (one would expect that the vast majority of words do not appear in conjunction with the vast majority of other words), and so our space/storage is not being used efficiently.
2. Subsequent classification models will exacerbate the sparsity issues, and so the models will be less robust.

Solution: We want to use low-dimensional vectors.

- Idea: store "most" of the important information in a fixed, small number of dimensions—a dense vector
- This is usually $25 - 1000$ dimensions, similar to Word2vec
- How can we reduce the dimensionality?

2.4.1 Singular Value Decomposition

One of the most common ways to reduce the dimensionality of these sparse vectors (while retaining all the key/non-zero information) is singular value decomposition. Essentially, we factorize X into $U\Sigma V^T$, where U and V are *orthonormal*.

Orthonormal matrices are square matrices Q such that $QQ^{-1} = I$. This means that

$$\begin{bmatrix} X_{11} & \dots & X_{1n} \\ \vdots & \ddots & \vdots \\ X_{m1} & \dots & X_{mn} \end{bmatrix} = \begin{bmatrix} U_{11} & \dots & U_{1m} \\ \vdots & \ddots & \vdots \\ U_{m1} & \dots & U_{mm} \end{bmatrix} \begin{bmatrix} \Sigma_{11} & \dots & \Sigma_{1n} \\ \vdots & \ddots & \vdots \\ \Sigma_{m1} & \dots & \Sigma_{mn} \end{bmatrix} \begin{bmatrix} V_{11} & \dots & U_{1n} \\ \vdots & \ddots & \vdots \\ U_{n1} & \dots & U_{nn} \end{bmatrix} \tag{17}$$

Here, we retain only k singular values, in order to generalize. $\hat{X}$ is the best rank k approximation to X, in terms of least squares. This is a classic linear algebra result, but is expensive to compute for large matrices.

However, running SVD on raw counts doesn't work well due to frequently used *function words* (ie: "the", "he", "has", etc.), which put too much of the focus on syntax. Some possible fixes are (Rohde et al. 2005 in COALS)

- log the frequencies
- $\min(X, t)$ with $t \approx 100$

- Ignore the function words altogether
- Ramped windows that weight closer words higher than further words
- Using Pearson Correlations instead of counts, then setting negative values to 0 ... etc.

2.5 GloVe Algorithm

The GloVe algorithm was borne out of a desire to combine the linear algebra methods using co-occurrence matrices and the Word2vec methods (SG, CBOW, NNLM, HLBL, RNN). It was found that while the linear algebra methods had faster training and a more efficient use of statistics, they were primarily used to capture word similarity, and there was a disproportionate importance given to words which showed up more frequently in the corpus.

On the other hand, while the Word2vec-based algorithms generated improved performance on other tasks and captured complex patterns beyond word similarity, the training time scaled with the corpus size and its usage of statistics was inefficient.

Crucial Insight: Ratios of co-occurrence probabilities can encode meaning components. [Pennington, Socher, and Manning, EMNLP 2014]. For example:

	x = solid	x = gas	x = water	x = random
$P(x\|\text{ice})$	large	small	large	small
$P(x\|\text{steam})$	small	large	large	small
$\frac{P(x\|\text{ice})}{P(x\|\text{steam})}$	large	small	~1	~1

Figure 11: Crucial Insight

Here, $P(x \mid \text{ice})$ is *large* for $x = \text{solid}$ and *small* for $x = \text{gas}$, which makes sense given that ice is a solid and not a gas. Similarly, $P(x \mid \text{steam})$ is *small* for $x = \text{solid}$ and *large* for $x = \text{gas}$, as steam is a gas and not a solid. Note also that

both $P(x \mid \text{ice})$ and $P(x \mid \text{steam})$ are large when $x = \text{water}$, as water pertains to both ice and steam, and that both probabilities are small when $x = \text{random}$, as "random" has very little to do with ice or steam.

The insight is to realize that we can *isolate* words which pertain specifically to some *difference* between two words which are otherwise quite similar by dividing their co-occurrence probabilities. In this case, the ratio of $\frac{P(x|\text{ice})}{P(x|\text{steam})}$ will tend to 1 whenever x pertains equally to both ice and steam. However, when x pertains to ice *more* than it does to steam, the ratio will be significantly greater than 1, and when x pertains to ice *less* than it does to steam, the ratio will be significantly less than 1.

Q: How can we capture ratios of co-occurrence probabilities as linear meaning components in a word vector space? We want to achieve something to the effect of "woman" + "king" - "man" = "queen". In the terms of our steam and ice analogy, we want:

$$\text{steam} + \text{solid} - \text{gas} = \text{ice} \tag{18}$$

How can we define our operations of addition and subtraction? Well, when we *add*, we want words which pertain (somewhat equally) to the words we're adding. We can thus define that

$$a + b = \left\{ x \;\middle|\; P(x \mid a) + P(x \mid b) = \max\left(P(x \mid a) + P(x \mid b) \right) \right\} \tag{19}$$

That is, simply assign $a + b$ to the word whose word vector x is such that $P(x \mid a) + P(x \mid b)$ is maximized. But this was just an aside. Let's see how Prof. Manning approaches the problem.

The proposed solution is to use a **log-bilinear model**, where:

$$w_i \cdot w_j = \log P(i \mid j) \tag{20}$$

At first, it seems like we're just using the obvious $w_i \cdot w_j = P(i \mid j)$ to determine the similarity between the row vectors in the co-occurrence matrix, and using a log term for no good reason. However, note that

$$\begin{aligned} w_x \cdot (w_a - w_b) &= w_x \cdot w_a - w_x \cdot w_b \\ &= \log P(w_x \mid w_a) - \log P(w_x \mid w_b) \\ &= \log \frac{P(w_x \mid w_a)}{P(w_x \mid w_b)} \end{aligned}$$

Which is the log of the *difference* between w_a and w_b in terms of whatever specific quality is denoted by w_x. For instance, in our previous example, $\text{solid} \cdot (\text{ice} - \text{steam})$ will be quite high.

We combine the best of both worlds by using the GloVe Model, which makes use of the log-bilinear model, $w_i \cdot w_j = \log P(i \mid j)$ and the following cost function:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{21}$$

This model unifies the pros of both the co-occurrence matrix models and the neural "Word2vec-esque" — it is calculated using a co-occurrence matrix but shares similarities with a neural model. The $f(X_{ij})$ simply caps the effect of words which occur very often. Then we simply have our regular log-bilinear model term, two bias terms, and the negative of the log of the co-occurrence. Essentially, then, we want the log of the entry in the co-occurrence matrix to be as similar as possible to the log of the dot product of the row vectors in the co-occurrence matrix.

This model yielded:

- Fast training
- Training scalable to huge corpora
- Good performance, even with small corpus and small vectors

2.6 How to Evaluate Word Vectors?

The answer to this question is related to general evaluation in NLP: Intrinsic vs. Extrinsic

Intrinsic:

- Evaluation on a specific/intermediate subtask
- Fast to compute
- Helps to understand that system
- Not clear if really helpful, unless correlation to real task is established

Extrinsic:

- Evaluation on a real task
- Can take a long time to compute accuracy
- Unclear if the subsystem or its interaction with other subsystems is the problem
- If replacing exactly one subsystem with another improves accuracy, we're winning!

2.6.1 Intrinsic Word Vector Evaluation

Essentially, we give our model a big collection of word analogy problems. Mathematically, the question of "a is to b as c is to what?" is answered by

$$d = \arg\max \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||} \tag{22}$$

Note: make sure to discard c from the search, because d is often calculated to be very close to c! The GloVe vectors have a strong linear component, and so you'd expect word analogies to work well with that particular model.

3 Codenames Bot

I want to make a Codenames bot. Essentially, given an input list of words we want to make associations with, and an input list of words we want to avoid associations with, we want to output the optimal word. We can try to use different models to go about this, but let's start with our original Word2vec model — we can endeavour to implement it ourselves instead of with Gensim.

The simplest possible algorithm, after forming the vector space, would just be to find the two words that are closest together and choose the word that is right between them (geometrically, in the vector space). So let's implement Word2vec first, try a basic strategy, and see where that leaves us.