Neural Networks

Introduction to Natural Language Processing (NLP)

Rafał Nowak May 6, 2025

Plan

- Word semantics (aka. context free word embedings)
- Simple NN architectures for NLP
- Recurent Neural Networks (RNN, LSTM, GRU)
- Language Models, Seq2Seq models

Bibliography

- Natural Language Processing with Deep Learning (CS224N/Ling284), by Chris Manning
- Speech and Language Processing, 3rd edition draft, by D. Jurasky, H. Martin (https://web.stanford.edu/~jurafsky/slp3/)
- Deep Learning book (Goodfellow, Bengio, Courville)

What could be treated as a natural language?

Natural Language as a Symbolic System

Natural language is one **type of symbolic system** used for communication and expression — but not the only one.

Other Symbolic Systems:

- **Computer Programs** – sequences of instructions (code) with syntactic and semantic rules
- **Musical Scores** – structured symbols indicating pitch, rhythm, dynamics
- **Images (at fixed resolution)** – interpreted as sequences of pixel blocks
- **DNA Sequences** – A, C, G, T symbols encoding biological instructions
- **Board Game Moves** – formal move sequences (e.g., in Chess, Othello, Go)

What Sets Natural Language Apart?

- **Ambiguity**: Multiple meanings per word/phrase
- **Context-dependence**: Interpretation changes with social, cultural, or conversational context
- **Evolving Semantics**: Vocabulary and usage shift over time
- **Flexible Syntax**: Grammatical rules with many exceptions and stylistic variations

What could be treated as a language?

Language = set of sentences

- **Natural Language**:
  Sentences are sequences of words (or characters, or phonemes)
  e.g., “The cat sat on the mat.”
- **Computer Programs**:
  Sequences of tokens/instructions
  e.g., for (int i = 0; i < n; i++)
- **DNA**:
  Sequences over the alphabet {A, C, G, T}
  e.g., ACGTGACCT...
- **Music Notation:**
  Sequence of musical symbols (notes, rests, dynamics)
- **Images (flattened)**:
  Fixed-size sequences over a pixel value space

What could be treated as a language?

Language = set of sentences

- Characters (ASCII, Extended ASCII, Unicode)
- Bytes (UTF-8)
- Words, Words++
- (sometimes common phrases, like New York, are treated as words)
- WordPieces (fixed number, approx. 30K, includes letters, and popular words)

Handling the Vocabulary Explosion

- Vocabulary **compression** and **generalization** are essential for scalable NLP — enabling models to deal with real-world language richness.

- new words, typos, domain-specific jargons

Common Solutions:

1. Special Tokens

Used to control model behavior or manage unknowns:

- `<BOS>` – Begin of Sentence
- `<EOS>` – End of Sentence
- `<PAD>` – Padding
- `<UNK>` – Unknown word (out of vocabulary)

2. Semantic-Class UNKs

A more informed alternative:

- `<unknown-noun>`, `<unknown-verb>`, etc.
- Helps preserve **grammatical structure** during training

3. Subword Units & Morphological Hints

Break words into **meaningful pieces**:

- Affix-based: `-ology`, `-ation`, `-izer`, etc.
- Byte-Pair Encoding (BPE), WordPiece: Learn frequent subword tokens
- Just use the **last K characters** for rare words
  → e.g., -ization, -ment, -ify can help generalization

Tokenization: Breaking Text into Units

- Tokenization is the process of splitting raw text into **tokens**, the basic units (words, subwords, or characters) that NLP models can process.

Simple Word-Based Tokenization (Baseline)

```python
def simple_tokenize(s):
    for c in punctuation:
        s = s.replace(c, ' ' + c + ' ')
    return s.lower().split()

```

Where punctuation can be:

```python
punctuation = '.,!?;:"()[]{}'
```

Example:

```python
s = "Hello, NLP world!"
tokens = simple_tokenize(s)
# Output: ['hello', ',', 'nlp', 'world', '!']
```

How to represent words/tokens

How to represent words/tokens

1. One-Hot Encoding

- Each word is represented by a binary vector.
- Only one index is 1, others are 0.
- **Pros**: Simple, intuitive.
- **Cons**: High dimensionality, no semantic meaning.

2. Count-Based Methods

- **Bag of Words (BoW)**: Word frequencies in a document.
- **TF-IDF**: Adjusts word frequency by inverse document frequency.
- **Pros**: Easy to compute.
- **Cons**: Ignores context and word order.

3. Word Embeddings

- **Word2Vec**, **GloVe**, **FastText**.
- Learn low-dimensional, dense vector representations.
- Capture semantic relationships: e.g., *king - man + woman ≈ queen*.
- **Pros**: Efficient, meaningful distances.
- **Cons**: Context-independent (static).

. Tokenization Strategies

- **Word-level**: Each word is a token.
- **Subword-level**: e.g., Byte Pair Encoding (BPE), SentencePiece.
- **Character-level**: Useful for morphologically rich or noisy text.

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).
- We 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***

How do we have usable meaning in a computer?

Previously commonest NLP solution: Use, e.g., WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** ("is a" relationships)

e.g., synonym sets containing "good":

```
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":

```
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

- A useful resource but missing nuance:
  - e.g., “proficient” is listed as a synonym for “good”
    This is only correct in some contexts
  - Also, WordNet list offensive synonyms in some synonym sets without any coverage of the connotations or appropriateness of words
- 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 be used to accurately compute word similarity (see following slides)

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

Such symbols for 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+)

Word vectors

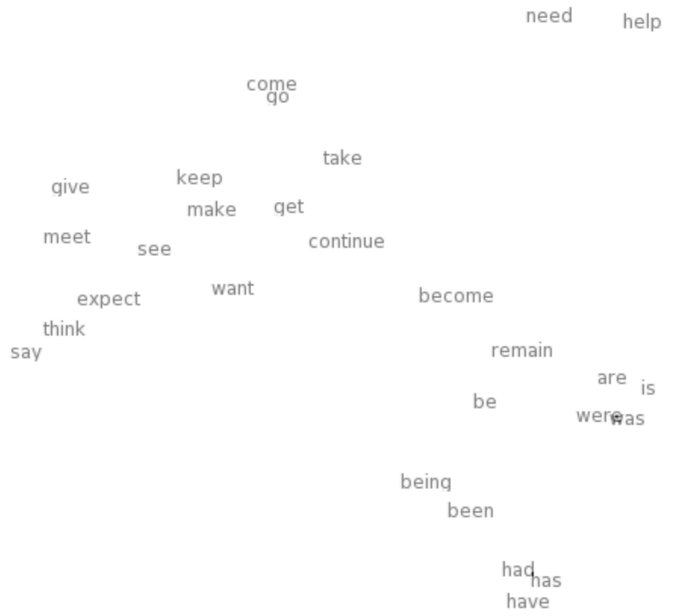
We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts, measuring similarity as the vector dot (scalar) product

$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix} \qquad \textit{monetary} = \begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix}$$

Note: word vectors are also called (word) embeddings or (neural) word representations
They are a distributed representation

Word meaning as a neural word vector – visualization

$$\textit{expect} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$



see https://projector.tensorflow.org/

Word2Vec

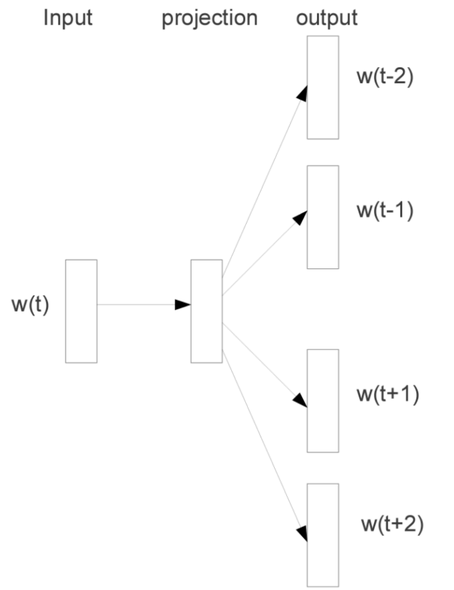
- Word2Vec is a family of models developed by Tomas Mikolov and colleagues at Google in 2013. It converts words into dense vectors of real numbers (called embeddings), capturing semantic meaning based on the context in which words appear.
  - Efficient Estimation of Word Representations in Vector Space Tomas Mikolov, Kai Chen, Greg Corrado, Jerey Dean
  - Distributed representations of words and phrases and their compositionality, Mikolov, Tomas; Sutskever, Ilya; Chen, Kai; Corrado, Greg S.; Dean, Je (2013).
- Instead of treating words as isolated symbols (like one-hot vectors), Word2Vec represents them in a continuous vector space, where similar words are close together.
- The first robust dense vector representation of words!
- The firrst example of successful pretraning in NLP!

3. Word2vec: Overview

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

Idea:

- We have a large corpus ("body") of text: a long list of words
- 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



Skip-gram model
(Mikolov et al. 2013)

How Word2Vec Works

Word2Vec comes in **two main architectures**:

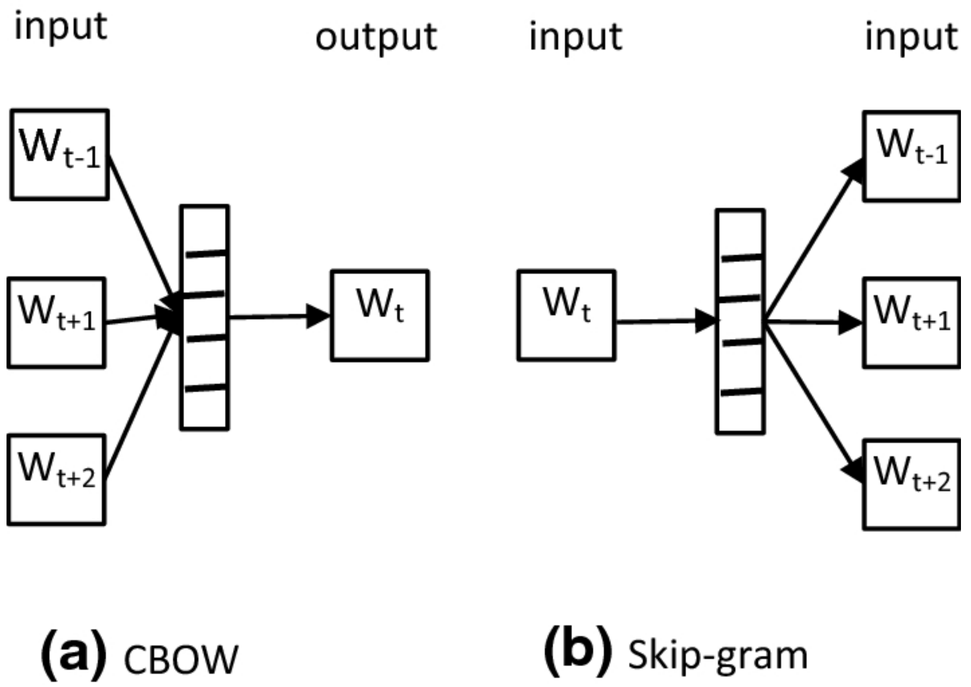
1. CBOW (Continuous Bag of Words)

- **Objective**: Predict a target word from its surrounding context words.
- **Example**: Given the context "the cat ___ on the mat", predict the missing word "sat".
- **Good for**: Small datasets and infrequent words.

2. Skip-gram

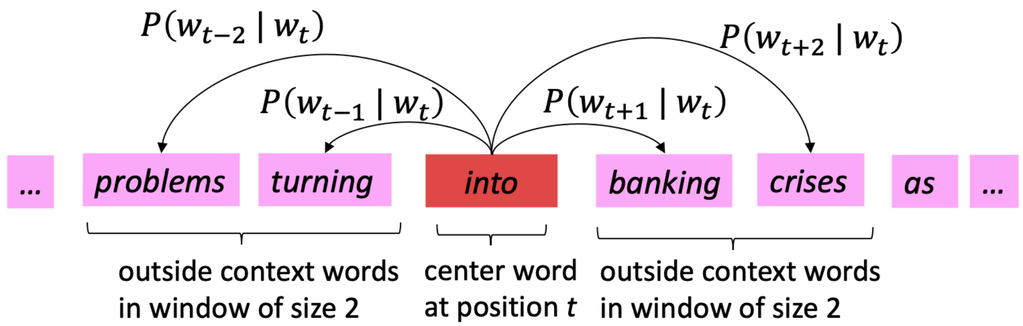
- **Objective**: Predict the surrounding context words from a target word.
- **Example**: Given the word "sat", predict the context words "the", "cat", "on", "the", "mat".
- **Good for**: Large datasets and better for infrequent words.

Both use a **neural network** with a single hidden layer, but the training is shallow and fast.



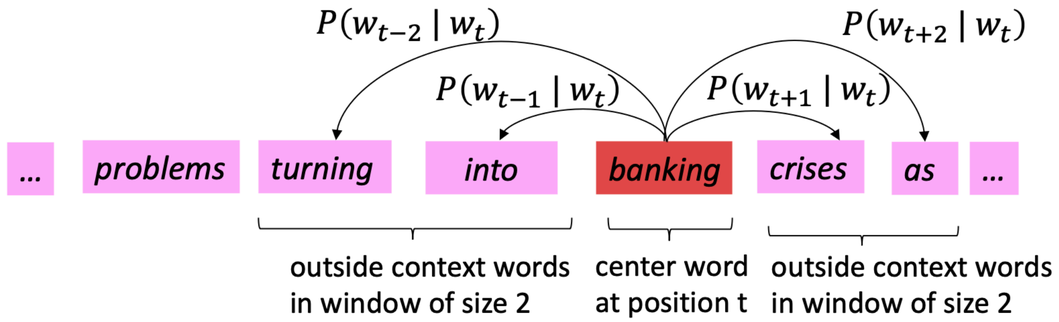
Word2Vec Overview

Example windows and process for computing $P(w_{t+j} \mid w_t)$



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_t. Data likelihood:

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

θ is all variables to be optimized

sometimes called a *cost* or *loss* function

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

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

Minimizing objective function $\Leftrightarrow$ Maximizing predictive accuracy

Word2vec: objective function

- We want to minimize the objective function:

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

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

} These word vectors are subparts of the big vector of all parameters θ

- 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

② Exponentiation makes anything positive

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

① 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

③ Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \to (0,1)^n$ ← Open region

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

- 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

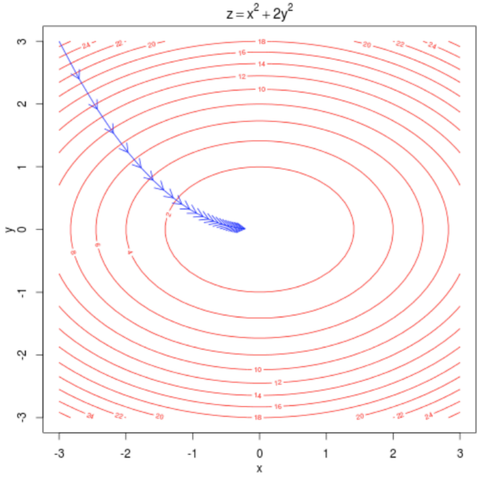
But sort of a weird name because it returns a distribution!

To train the model: Optimize value of parameters to minimize loss

To train a model, we gradually adjust parameters to minimize a loss

- Recall: θ represents **all** the model parameters, in one long vector
- In our case, with *d*-dimensional vectors and *V*-many words, we have →
- Remember: every word has two vectors

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



- We optimize these parameters by walking down the gradient (see right figure)
- We compute all vector gradients!

Interactive Session!

- $L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P\left(w_{t+j} \mid w_t; \theta\right)$
- For a center word *c* and a context word *o*: $P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$

4. Objective Function

Maximize $J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} p(w'_{t+j} | w_t; \theta)$

Or minimize ave. neg. log likelihood $J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log p(w'_{t+j} | w_t)$

[negate to minimize; log is monotone]

text length

window size

where $p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$

word IDs

We now take derivatives to work out minimum

Each word type (vocab entry) has two word representations: as center word and context word

$$\frac{\partial}{\partial v_c} \log \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w v_c)}$$

$$= \underbrace{\frac{\partial}{\partial v_c} \log \exp(u_o^T v_c)}_{①} - \underbrace{\frac{\partial}{\partial v_c} \log \sum_{w=1}^{V} \exp(u_w^T v_c)}_{②}$$

① $\frac{\partial}{\partial v_c} \underbrace{\log \exp}_{\text{inverses}}(u_o^T v_c) = \frac{\partial}{\partial v_c} u_o^T v_c = u_o$

↑ Vector! Not high school single variable calculus

You can do things one variable at a time, and this may be helpful when things get gnarly.

$$\forall j \quad \frac{\partial}{\partial (v_c)_j} u_o^T v_c = \frac{\partial}{\partial (v_c)_j} \sum_{i=1}^{d} (u_o)_i (v_c)_i$$

$$= (u_o)_j$$

Each term is zero except when $i = j$

② $\frac{\partial}{\partial v_c} \underbrace{\log}_{f} \underbrace{\sum_{w=1}^{V} \exp(u_w^T v_c)}_{z = g(v_c)}$

$= \underbrace{\frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)}}_{\frac{\partial}{\partial z} f(z)} \cdot \underbrace{\frac{\partial}{\partial v_c} \sum_{x=1}^{V} \exp(u_x^T v_c)}_{\text{Important to change index}}$

$\frac{\partial}{\partial v_c} f(\overbrace{g(v_c)}^{z}) = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial v_c}$ Use chain rule

$= \frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)} \cdot \left(\sum_{x=1}^{V} \frac{\partial}{\partial v_c} \underbrace{\exp}_{f} \underbrace{(u_x^T v_c)}_{z = g(v_c)} \right)$ Move deriv inside sum

$\left(\sum_{x=1}^{V} \exp(u_x^T v_c) \frac{\partial}{\partial v_c} u_x^T v_c \right)$ Chain rule

$\left(\sum_{x=1}^{V} \exp(u_x^T v_c) u_x \right)$

$$\frac{\partial}{\partial v_c} \log\left(p(o|c)\right) = u_o - \frac{1}{\sum_{w=1}^{V} \exp(u_w^T v_c)} \cdot \left(\sum_{x=1}^{V} \exp(u_x^T v_c)\, u_x\right)$$

$$= u_o - \sum_{x=1}^{V} \frac{\exp(u_x^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}\, u_x$$

Distribute term across sum

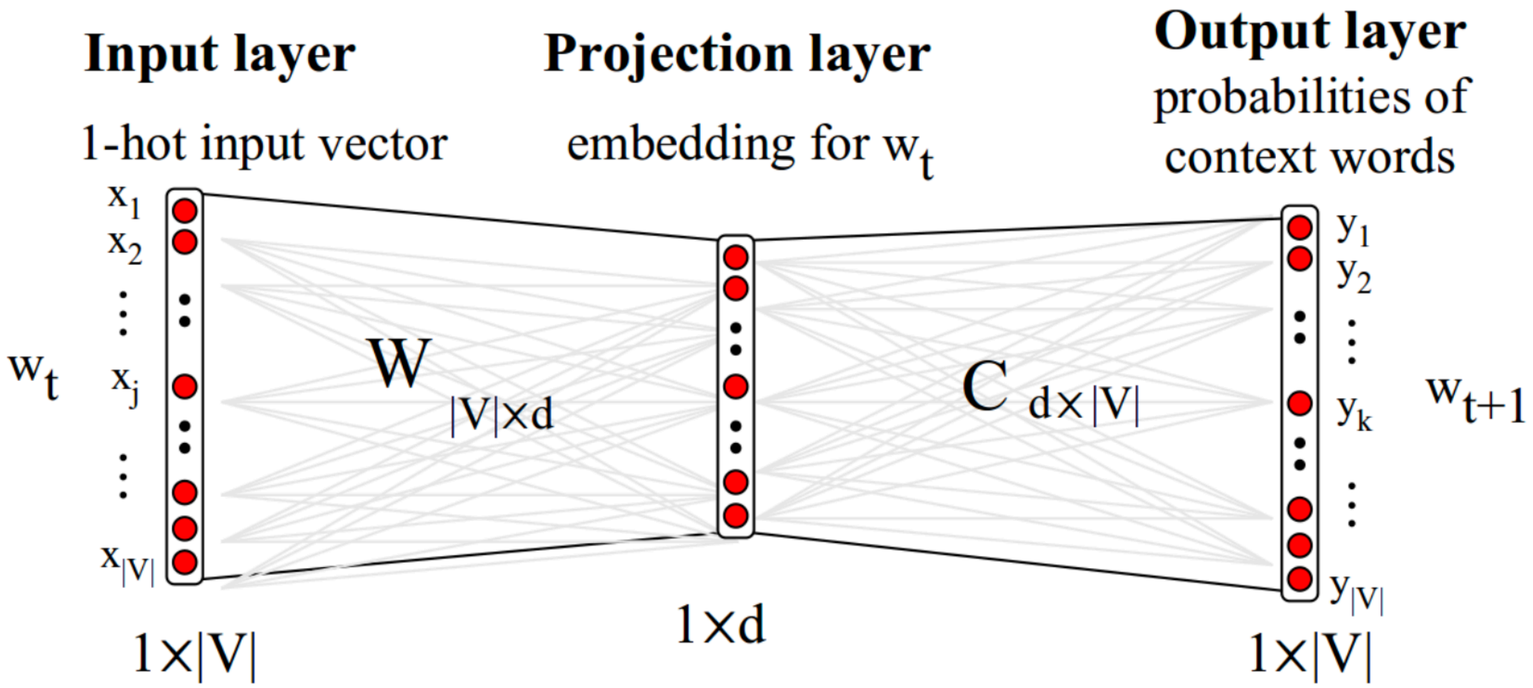
$$= u_o - \sum_{x=1}^{V} p(x|c)\, u_x$$

this an expectation: average over all context vectors weighted by their probability

$$= \text{observed} - \text{expected}$$

This is just the derivatives for the center vector parameters
Also need derivatives for output vector parameters
(they're similar)
Then we have derivative w.r.t. all parameters and can minimize

Skip-Gram model



Problem with large softmax

- Large sum in the gradient
- If V (vocabulary size) is large (e.g., 100,000), this is very slow.
- You must compute a dot product and an exponential for every word at every training step.

- **Solution:**
  - hierarchical softmax (decompose the decision to the list of decisions)
  - **negative sampling**

Sampling

- We can easily obtain related objects
  - Two consecutive words, sentences, or sentences in one article, ...
- We can **sample unrelated** objects

The skip-gram model with negative sampling (HW2)

- Notation more similar to class and HW2:

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

- We take *k* negative samples (using word probabilities)
- Maximize probability that real outside word appears;
  minimize probability that random words appear around center word

- Sample with $P(w)=U(w)^{3/4}/Z$, the unigram distribution U(w) raised to the 3/4 power (We provide this function in the starter code).
- The power makes less frequent words be sampled more often

Negative sampling

Loss function

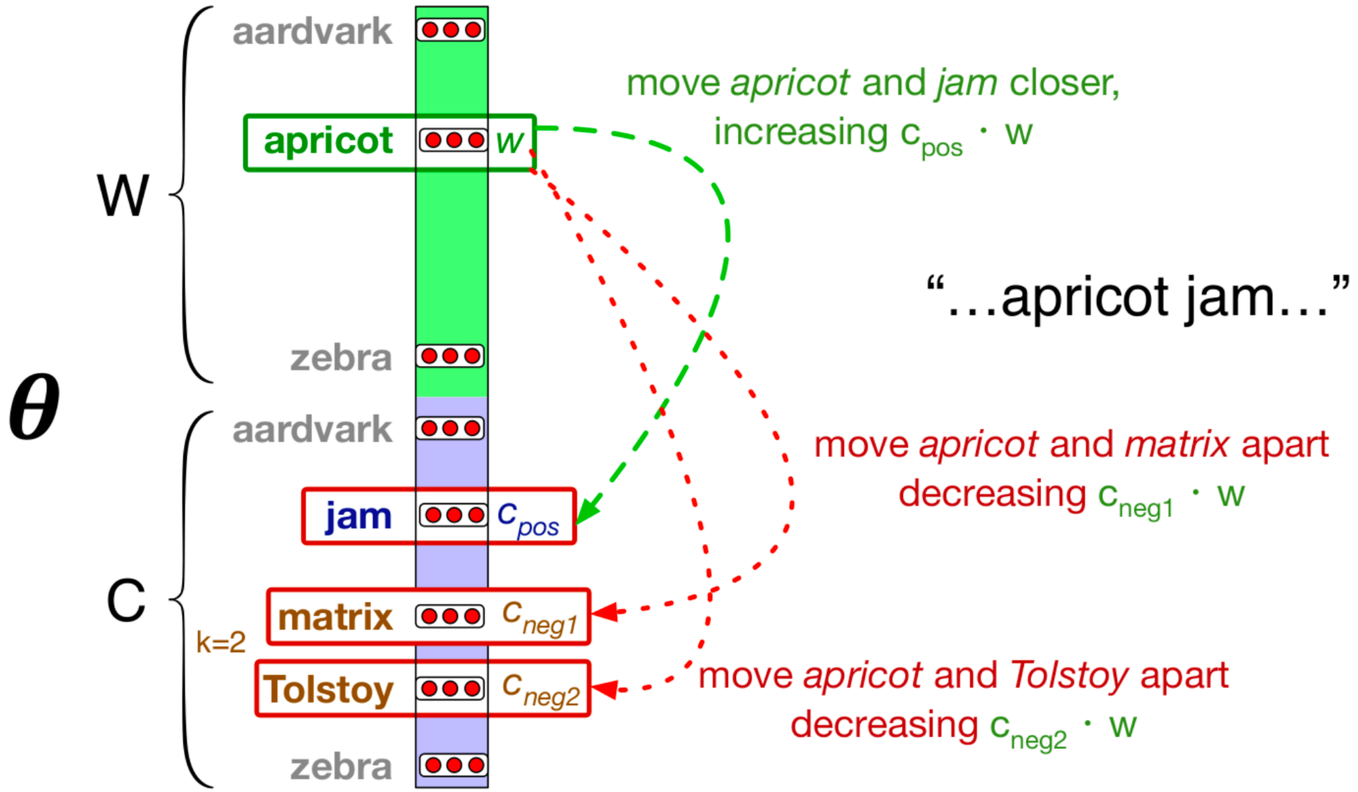
Given:

- A **center word** `w_c`
- A **true context word** `w_o`
- *k* **negative samples**: words not in the context (denoted as `w_1`, ..., `w_k`)
- Word vectors:
  - `v_c` = embedding of the center word (from input matrix)
  - `u_o` = embedding of the context word (from output matrix)
  - `u_i` = embeddings of negative samples

The Negative Sampling Loss:

$$L = -\log \sigma(u_o^T v_c) - \sum_{i=1}^{k} \log \sigma(-u_i^T v_c)$$

Intuition of one step of gradient descent



Word2Vec training details

- Linear learning rate decay
- Window size ≈ 10
  - smaller window – more syntactic relations
  - bigger window – more semantic
- 3-6 epochs
- Starting learning rate $= 0.003$

Sample negative words with $P'(w) \sim \text{cnt}(w)^{0.75}$

How Word2Vec works

- Training Mechanisms
  - Negative Sampling
  - Hierarchical Softmax
- Dimensinality & Output
  - each word is represented by a vector (100-300 dim)
  - After training, you get a word embedding matrix.
  - Words with similar meaning (or that appear in similar contexts) have similar vectors
    - king - man + woman ≈ queen

W2V applications

- Text classification
- Sentiment analysis
- Recommendation systems
- Machine translation
- Pre-training for deep learning models

see slides nn_nlp1.pdf