Statistical Methods in Natural Language Processing

7. Word Representations

Pavel Pecina, Jindřich Helcl

2 December, 2025

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Course Segments

1. Introduction, probability, essential information theory
2. Statistical language modelling (n-gram)
3. Statistical properties of words
4. Word representations
5. Hidden Markov models, Tagging

Recap from Last Week

The New Model

Rewrite the n-gram LM using classes:

- Original definition: $[k = 1 \ldots n]$

  $p_k(w_i|h_i) = c(h_i,w_i) / c(h_i)$ [history: $(k-1)$ words]

- Introduce classes:

$$p_k(w_i|h_i) = p(w_i|c_i)\ p_k(c_i|h_i)$$

  - history: classes, too: [for trigram: $h_i = c_{i-2},c_{i-1}$, bigram: $h_i = c_{i-1}$]
- Smoothing as usual
  - over $p_k(w_i|h_i)$, where each is defined as above (except uniform which is $1/|V|$)

Creating the Word-to-Class Map (Brown's Classes)

- Consider bigram model for now.
- Bigram estimate:

$$p_2(c_i|h_i) = p_2(c_i|c_{i-1}) = c(c_{i-1},c_i) / c(c_{i-1}) = c(r(w_{i-1}),r(w_i)) / c(r(w_{i-1}))$$

- Form of the model:
  - just raw bigram for now:

    $P(T) = \Pi_{i=1..|T|} p(w_i|r(w_i))\ p_2(r(w_i)|r(w_{i-1}))$ $\qquad (p_2(c_1|c_0) =_{df} p(c_1))$

- Maximize over r (given $r \to$ fixed p, p_2):
  - define objective

    $L(r) = 1/|T|\ \Sigma_{i=1..|T|} \log(p(w_i|r(w_i))\ p_2(r(w_i))|r(w_{i-1})))$

    $r_{best} = \text{argmax}_r\ L(r)$ $\quad$ ($L(r)$ = norm. logprob of training data ... as usual)

Simplifying the Objective Function

- Start from $L(r) = 1/|T| \, \Sigma_{i=1..|T|} \log(p(w_i|r(w_i)) \, p_2(r(w_i)|r(w_{i-1})))$:

$$1/|T| \, \Sigma_{i=1..|T|} \log(p(w_i|r(w_i)) \, \underline{p(r(w_i))} \, p_2(r(w_i)|r(w_{i-1})) \, / \, \underline{p(r(w_i))}) =$$
$$1/|T| \, \Sigma_{i=1..|T|} \log(\underline{p(w_i,r(w_i))} \, p_2(r(w_i)|r(w_{i-1})) \, / \, p(r(w_i))) =$$
$$1/|T| \, \Sigma_{i=1..|T|} \log(\underline{p(w_i)}) + 1/|T| \, \Sigma_{i=1..|T|} \log(p_2(r(w_i)|r(w_{i-1})) \, / \, p(r(w_i))) =$$
$$-H(W) + 1/|T| \, \Sigma_{i=1..|T|} \log(p_2(r(w_i)|r(w_{i-1})) \, \underline{p(r(w_{i-1})) \, / \, (p(r(w_{i-1}))} \, p(r(w_i)))) =$$
$$-H(W) + 1/|T| \, \Sigma_{i=1..|T|} \log(\underline{p(r(w_i),r(w_{i-1})}) \, / \, (p(r(w_{i-1})) \, p(r(w_i)))) =$$
$$-H(W) + \Sigma_{d,e \in C} \, p(d,e) \log(\, p(d,e) \, / \, (p(d) \, p(e)) \,) =$$
$$-H(W) + I(D,E)$$

(event E picks class adjacent (to the right) to the one picked by D)

- Since W does not depend on r, we ended up with maximizing $I(D,E)$

The Greedy Algorithm

- Define merging operation on the mapping $r: V \rightarrow C$:
  - merge: $R \times C \times C \rightarrow R' \times C\text{-}1: (r,k,l) \rightarrow r',C'$ such that
  - $C^{-1} = \{C - \{k,l\} \cup \{m\}\}$ (throw out k and l, add new $m \notin C$)
  - $r'(w) = m$ for $w \in r_{INV}(\{k,l\})$,
    $r(w)$ otherwise.

1. Start with each word in its own class $(C = V)$, $r = id$.
2. Merge two classes k,l into one, m, such that
   $(k,l) = \text{argmax}_{k,l}\ I_{merge(r,k,l)}(D,E)$.
3. Set new $(r,C) = merge(r,k,l)$.
4. Repeat 2 and 3 until $|C|$ reaches predetermined size.

Complexity Issues

Still too complex:

- $|V|$ iterations of the steps 2 and 3.
- $|V|^2$ steps to maximize $\mathrm{argmax}_{k,l}$ (selecting k,l freely from $|C|$, which is in the order of $|V|^2$)
- $|V|^2$ steps to compute $I(D,E)$ (sum within sum, all classes, also: includes $\log$)

  ⇒ total: $|V|^5$

- i.e., for $|V| = 100$, about 10^{10} steps (several hours!)
- but $|V| \sim 50{,}000$ or more

Formula breakdown

- Mutual Information at kth iteration (= k classes):
  - $I_k = \Sigma_{l,r \in C}\, p_k(l,r) \log(p_k(l,r) / (p_{kl}(l)\, p_{kr}(r)))$
- For each pair of classes at iteration k, we define:
  - $q_k(l,r) = p_k(l,r) \log(p_k(l,r) / (p_{kl}(l)\, p_{kr}(r)))$
- So:
  - $I_k = \Sigma_{l,r \in C}\, q_k(l,r)$
- $q_k(l,r)$ using bigram counts $c_k(l,r)$:

$$q_k(l,r) = c_k(l,r)/N \log(N\, c_k(l,r)/(c_{kl}(l)\, c_{kr}(r)))$$

l \ r	c_1	c_2	c_3	c_4
c_1	10	2	0	1
c_2	0	0	5	2
c_3	0	2	0	3
c_4	2	3	0	0

unigram/marginal counts

Trick #1: Recomputing MI the Smart Way

- For test-merging c_2 and c_4 we recompute only rows/columns 2 & 4:
  1. Subtract column/row (2 & 4) from the MI sum:

     (be careful at the intersections)

  2. add sums of the merged counts (row & column for (c_2' is the merged class):

     (watch the intersection again)

l \ r	c_1	c_2	c_3	c_4
c_1	10	2	0	1
c_2	0	0	5	2
c_3	0	2	0	3
c_4	2	3	0	0

l \ r	c_1	c_2'	c_3
c_1	10	3	0
c_2'	2	5	5
c_3	0	5	0

Trick #2: Precompute the Counts-to-be-Subtracted

- Summing loop goes through i,j
  - … but the single row/column sums do not depend on the (resulting sums after the) merge ⇒ can be precomputed
  - only $2k$ logs to compute at each algorithm iteration, instead of k^2
- Then for each "merge-to-be" compute only add-on sums, plus "intersection adjustment"

Formulas for Tricks #1 and #2

- Recap:

$$q_k(l,r) = p_k(l,r) \log(p_k(l,r) / (p_{kl}(l)\, p_{kr}(r)))$$

the same, but using counts:

$$q_k(l,r) = c_k(l,r)/N \log(N\, c_k(l,r)/(c_{kl}(l)\, c_{kr}(r)))$$

- Define further (row+column a sum):

$$s_k(a) = \Sigma_{l=1..k} q_k(l,a) + \Sigma_{r=1..k} q_k(a,r) - q_k(a,a)$$

- Then, the subtraction part of Trick #1 amounts to

$$sub_k(a,b) = s_k(a) + s_k(b) - q_k(a,b) - q_k(b,a)$$

Intersection adjustment

	c_2	c_3	c_4
c_2	0	5	2
c_3	2	0	3
c_4	3	0	0

precomputed

remaining intersection adjustment

Formulas - cont.

- After-merge add-on:

  $$add_k(a,b) = \sum_{l=1..k,l\neq a,b} q_k(l,a+b) + \sum_{r=1..k,r\neq a,b} q_k(a+b,r) + q_k(a+b,a+b)$$

- $a+b$ is the new (merged) class
  - Hint: use the definition of q_k as a "macro", and then:

    $p_k(a+b,r) = p_k(a,r) + p_k(b,r)$ (same for other sums, equivalent)
- The above sums cannot be precomputed
- Mutual Information after merge of class a,b:
  - $I_{k-1}(a,b) = I_k - sub_k(a,b) + add_k(a,b)$
  - I_k is the "old" MI, kept from previous iteration of the algorithm

Trick #3: Ignore Zero Counts

- Many bigrams are 0
  - e.g. in the Canadian Hansards corpus, < .1 % of bigrams are non-zero)
- Consider non-zero bigrams only:
  - e.g. create linked lists of non-zero counts in columns and rows
  - similar effect: use hashes (store non-zero-count bigrams)
- Update links after merge (after step 3)

Trick #4: Use Updated Loss of MI

- We are now down to $|V|^4$: $|V|$ merges, each merge takes $|V|^2$ "test-merges", each test-merge involves order-of-$|V|$ operations ($add_k(i,j)$ term, slide 34)
- Observation:
    - many numbers (s_k, q_k) needed to compute the mutual information loss due to a merge of $i+j$ ***do not change*:** namely, those which are not in the vicinity of neither i nor j.
- Idea:
    - keep the MI loss matrix for all pairs of classes, and (after a merge) update only those cells which have been influenced by the merge.

Formulas for Trick #4 (s_{k-1}, L_{k-1})

- Keep a matrix of "losses" $L_k(d,e)$ [symmetry: $L_k(d,e) = L_k(e,d)$]
- Init: $L_k(d,e) = sub_k(d,e) - add_k(d,e)$ [then $I_{k-1}(d,e) = I_k - L_k(d,e)$]
- Suppose a,b are now the two classes merged into a
- Update (k-1: index used for the next iteration; $i,j \neq a,b$):

$$s_{k-1}(i) = s_k(i) - q_k(i,a) - q_k(a,i) - q_k(i,b) - q_k(b,i) + q_{k-1}(a,i) + q_{k-1}(i,a)$$

$$L_{k-1}(i,j) = L_k(i,j) - s_k(i) + s_{k-1}(i) - s_k(j) + s_{k-1}(j) + \\ + q_k(i+j,a) + q_k(a,i+j) + q_k(i+j,b) + q_k(b,i+j) - \\ - q_{k-1}(i+j,a) - q_{k-1}(a,i+j)$$

Word Representations

Word Representations in n-gram LMs

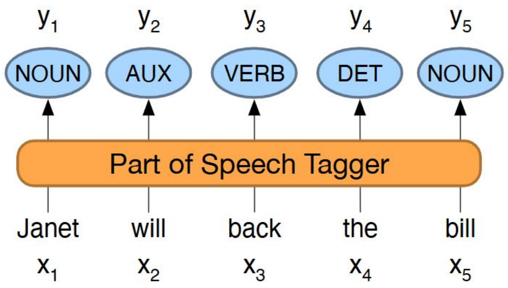
Word representation in n-gram language models

- by themselves (strings) as dictionary keys (or integer indices to a vocabulary list)
- by class numbers
- context-independent, needs to be modeled with longer n-grams
- also independent on each other (does not capture semantic similarity between words/classes)

Word Representations in NLP Tasks

Many NLP tasks depend on good word representations.

- Example: part-of-speech tagging
  - for given word x, produce a part of speech tag y
  - using task-specific training data T_{pos} and test data S_{pos}

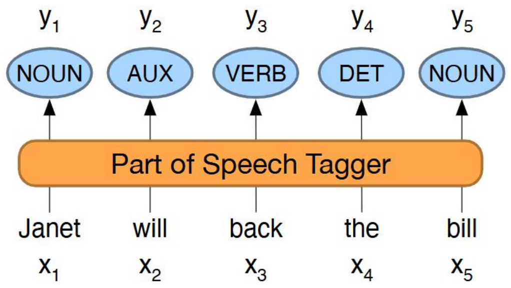


Original image credit: Figure 17.3 in Jurafsky and Martin, Speech and Language Processing (3rd ed. draft), 2025

Word Representations in NLP Tasks

Many NLP tasks depend on good word representations.

- Example: part-of-speech tagging
  - for given word x, produce a part of speech tag y
  - using task-specific training data T_{pos} and test data S_{pos}
- OOV problem
  - S_{pos} contains words (or word/tag combinations) not present in T_{pos}
  - If we use T_{pos} with discrete word representations we cannot decide about OOVs

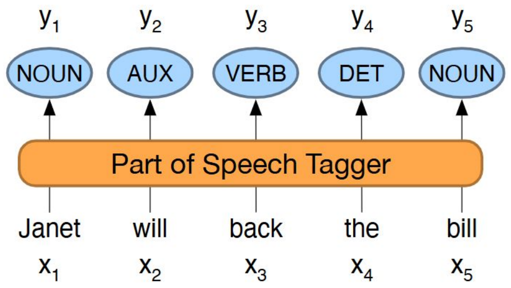


Original image credit: Figure 17.3 in Jurafsky and Martin, Speech and Language Processing (3rd ed. draft), 2025

Word Representations in NLP Tasks

Many NLP tasks depend on good word representations.

- Example: part-of-speech tagging
  - for given word x, produce a part of speech tag y
  - using task-specific training data T_{pos} and test data S_{pos}



- OOV problem
  - S_{pos} contains words (or word/tag combinations) not present in T_{pos}
  - If we use T_{pos} with discrete word representations we cannot decide about OOVs
- Idea: Re-use word representations from a language model
  - trained on much larger data (larger than T_{pos}) with much larger vocabulary
  - how can this be done with our n-gram LMs?

Original image credit: Figure 17.3 in Jurafsky and Martin, Speech and Language Processing (3rd ed. draft), 2025

Representations as the Product of LMs

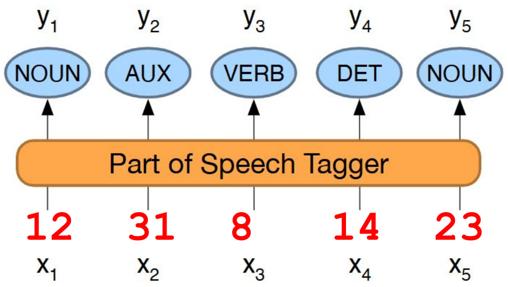
Using word representations from n-gram LMs to solve POS tagging

- originally words are represented with dictionary keys
    - no direct comparability, no similarity modeled

Representations as the Product of LMs

Using word representations from n-gram LMs to solve POS tagging

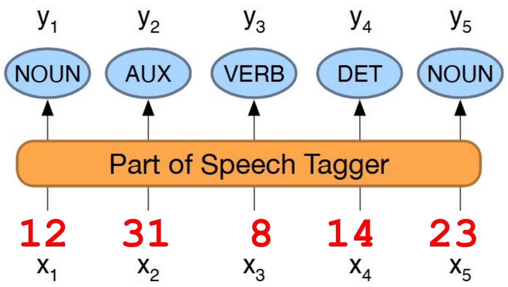
- originally words are represented with dictionary keys
  - no direct comparability, no similarity modeled
- representing a word by its class
  - we can represent words that were not seen in T_{pos}!
  - to train a POS classifier, we can pre-process the task-specific data T_{pos} and S_{pos} by replacing words by their class as given by a class n-gram model
  - works, but it is too coarse
    - no similarity between classes
    - words either belong or do not belong to one class



Beyond Word Classes

Capturing word similarities

- word classes divide the vocabulary into equivalence classes
  - words within a class considered similar
- real groups of similar words are more general
  - non-disjoint, not transitive, ambiguous ...
  - e.g. *book+pay* vs. *book+text*

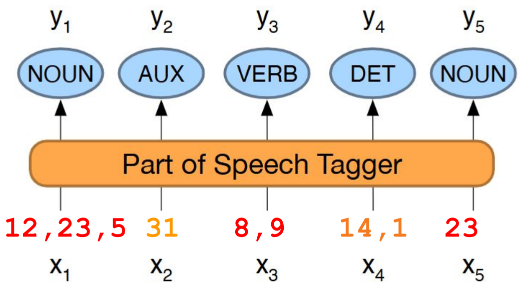


Beyond Word Classes

Capturing word similarities

- word classes divide the vocabulary into equivalence classes
  - words within a class considered similar
- real groups of similar words are more general
  - non-disjoint, not transitive, ambiguous ...
  - e.g. {*book,pay*} vs. {*book,text*}

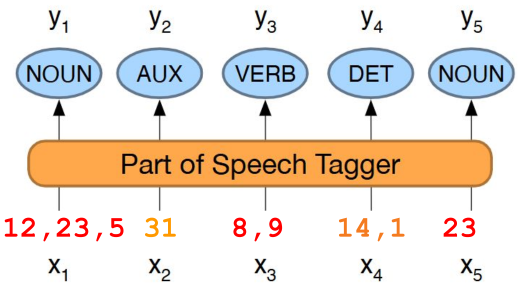
Each word should be able to belong to more classes!



Word Embeddings

Capturing word similarities

- Previously: each word can belong to more than one class
  - stored as a boolean vector
  - a word either belongs to a class or not
  - enables looking at similarity of classes (in terms of words they contain)
  - but we still cannot distinguish words within the same (set of) class(es)

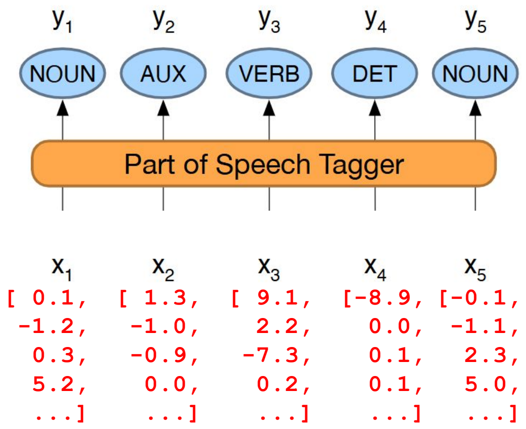


Word Embeddings

Capturing word similarities

- Previously: each word can belong to more than one class
  - stored as a boolean vector
  - a word either belongs to a class or not
  - enables looking at similarity of classes (in terms of words they contain)
  - but we still cannot distinguish words within the same (set of) class(es)

- Use real numbers instead of booleans?
  - stored as vector of floats
  - each word belong to all classes *to different extents*
  - fine-grained categorization of vocabulary
  - these vectors are called **word embeddings**



Embedding Matrix

With pre-defined vocabulary V, let $E \in \mathbb{R}^{|V| \times d}$ be an **embedding matrix**

- Each row of E corresponds to a word from the vocabulary
- d is the **embedding size** (or dimension)
  - a hyper-parameter and needs to be decided before obtaining the embeddings

hello	-1.2	0.3	5.0	...
the	0.1	3.6	2.2	...
of	3.3	-1.0	3.1	...
dog	-1.0	7.5	7.1	...

- OOV words?
  - Usually collapsed into a special `<oov>` token that will get its embedding

Obtaining Embeddings

Getting word embeddings

- implicitly in end-to-end scenario
  - in neural-network-based models with sufficient data volumes (such as tasks like machine translation or language modeling), embeddings are trained along with the rest of the network
- using off-the-shelf pre-trained embeddings
  - embeddings for many languages readily available (FastText, Hugging Face, ...)
  - download the embedding matrix and train your classifier on the task-specific data
- **learning** from data

Learning Word Embeddings - Word2vec

Ideas of the Word2vec model (Mikolov et al., 2013)

- Use language modeling data for obtaining word embeddings, then re-use word embeddings for a specific task (such as POS tagging)
- Keep the word embedding model simple
  - smaller model means we can use more data for the same cost
- Embedding of a word should be based on the word's context
  - "you shall know a word by the company it keeps"
  - i.e. the Distributional Hypothesis (see Lecture 5, slide 14)

Word2vec Model Architecture

Two-layer linear neural network $w = (0,0,0,1,0,0)$

1. word embedding (here w is a one-hot representation of the input word)

$$h = w \times E$$

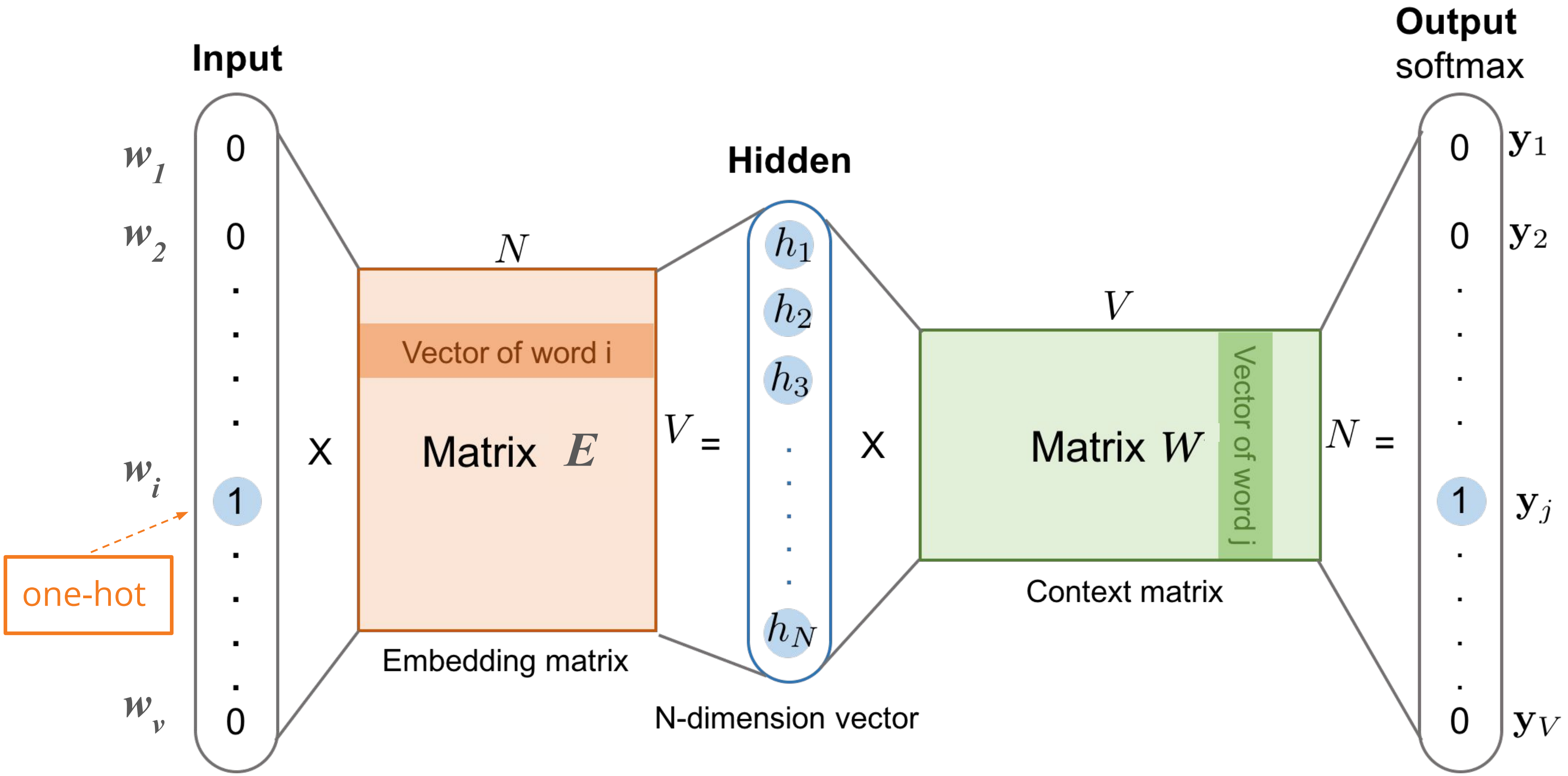
2. output projection with softmax

$$y = \mathrm{softmax}(h \times W + b)$$

$E \in \mathbb{R}^{|V| \times d}$, $W \in \mathbb{R}^{d \times |V|}$, and $b \in \mathbb{R}^{|V|}$ are *trainable* parameters

- The output vector y can be represented as probability distribution over vocabulary: the probability of each word appearing in the context of the input word w.
- *Note the shape of W (similar to E, only transposed): columns of W are called "output embeddings" or "context embeddings"*

Word2vec Model Architecture



Original image credit: https://lilianweng.github.io/posts/2017-10-15-word-embedding/

Word2vec Training

Training objective: minimizing cross-entropy loss

- cross-entropy: distance between *two* distributions
  - in our case, the predicted distribution (output of softmax), vs the "true" distribution, i.e. the one-hot vector representing the target word
- goal: find parameters E, W, b that minimize the cross-entropy
  - using gradient descent,
  - and a large training dataset to sample from

Word2vec Training

Training is done by sampling input and target word(s) from data.

- **CBOW** (continuous bag-of-words): For a given target word *w*, sum embeddings of the context words and predict *w* (one training example).
- **Skip-gram**: For a given input word *w*, predict words in its context (one training example per context word)

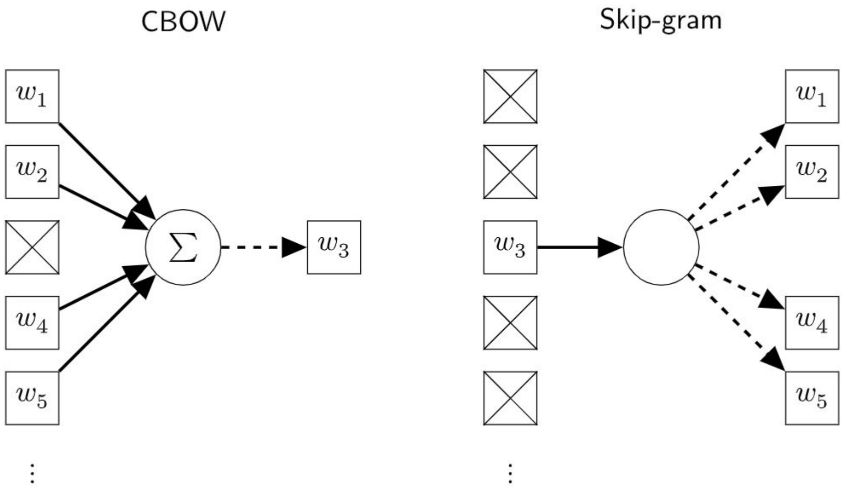


Image credit: Libovický et al., Slides for NPFL129 lecture 6, 2025

Word2vec Data Sampling (skip-gram case)

1.	[All human beings] are born free and equal in dignity …	→	(All, humans) (All, beings)
2.	[All human beings are] born free and equal in dignity …	→	(human, All) (human, beings) (human, are)
3.	[All human beings are born] free and equal in dignity …	→	(beings, All) (beings, human) (beings, are) (beings, born)
4.	All [human beings are born free] and equal in dignity …	→	(are, human) (are, beings) (are, born) (are, free)

Slide courtesy of Libovický et al., Slides for NPFL129 lecture 6, 2025

Word2vec Implementation

The *softmax* operation normalizes probabilities across the vocabulary.

- For vector $z = (z_1, z_2, ..., z_n)$ the softmax component is $\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$
- For large $n=|V|$ this slows down the training (computing the norm. factor)
- Improvement: Use sigmoid instead of softmax
  - does not model the probability for all context words given an input
  - works as binary classification (0/1) for a given pair of input and context words
- The cross-entropy loss given (input, context) word pair becomes:

$$-\log \sigma(e^\top \cdot c)$$

where *e* and *c* are the input embedding and context output embedding respectively and

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Word2vec Implementation, cont.

Using sigmoid instead of softmax gives us:

$$-\log \sigma(e^\top \cdot c)$$

- This is only for word pairs that do occur together (positive samples)
- For better training we need to also train on negative ones
  - a process called *negative sampling*
- Taking into account pairs of words that *do not* occur together, we can complete the loss function:

$$L = -\log \sigma(e^\top \cdot c_p) - \sum_{c_n} \log(1 - \sigma(e^\top \cdot c_n))$$

Word2vec Implementation, cont.

Using sigmoid instead of softmax gives us:

$$-\log \sigma(e^\top \cdot c)$$

- This is only for word pairs that do occur together (positive samples)
- For better training we need to also train on negative ones
  - a process called *negative sampling*
- Taking into account pairs of words that *do not* occur together, we can complete the loss function:

$$L = \boxed{-\log \sigma(e^\top \cdot c_p)} - \sum_{c_n} \log(1 - \sigma(e^\top \cdot c_n))$$

loss from the positive example

Word2vec Implementation, cont.

Using sigmoid instead of softmax gives us:

$$-\log \sigma(e^\top \cdot c)$$

- This is only for word pairs that do occur together (positive samples)
- For better training we need to also train on negative ones
  - a process called *negative sampling*
- Taking into account pairs of words that *do not* occur together, we can complete the loss function:

loss from the negative example

$$L = -\log \sigma(e^\top \cdot c_p) - \sum_{c_n} \log(1 - \sigma(e^\top \cdot c_n))$$

loss from the positive example

Word2vec Implementation, cont.

Using sigmoid instead of softmax gives us:

$$-\log \sigma(e^\top \cdot c)$$

- This is only for word pairs that do occur together (positive samples)
- For better training we need to also train on negative ones
  - a proces
- Taking into a *do not* occur together, we can complete th

we sample more negative examples than positive, usually around 2-5

loss from the negative example

$$L = -\log \sigma(e^\top \cdot c_p) - \sum_{c_n} \log(1 - \sigma(e^\top \cdot c_n))$$

loss from the positive example

Word2vec Implementation, cont.

Using sigmoid instead of softmax gives us:

$$-\log \sigma(e^\top \cdot c)$$

- This is only for word pairs that do occur together (positive samples)
- For better training we need to also train on negative ones
  - a process
- Taking into a... *do not* occur together, we can complete th...

we sample more negative examples than positive, usually around 2-5

loss from the negative example

$$L = -\log \sigma(e^\top \cdot c_p) - \sum_{c_n} \log(1 - \sigma(e^\top \cdot c_n))$$

loss from the positive example

note that the input word *e* stays the same

Word2vec Vector Properties I

Word2vec vectors seem to enable meaningful arithmetic operations

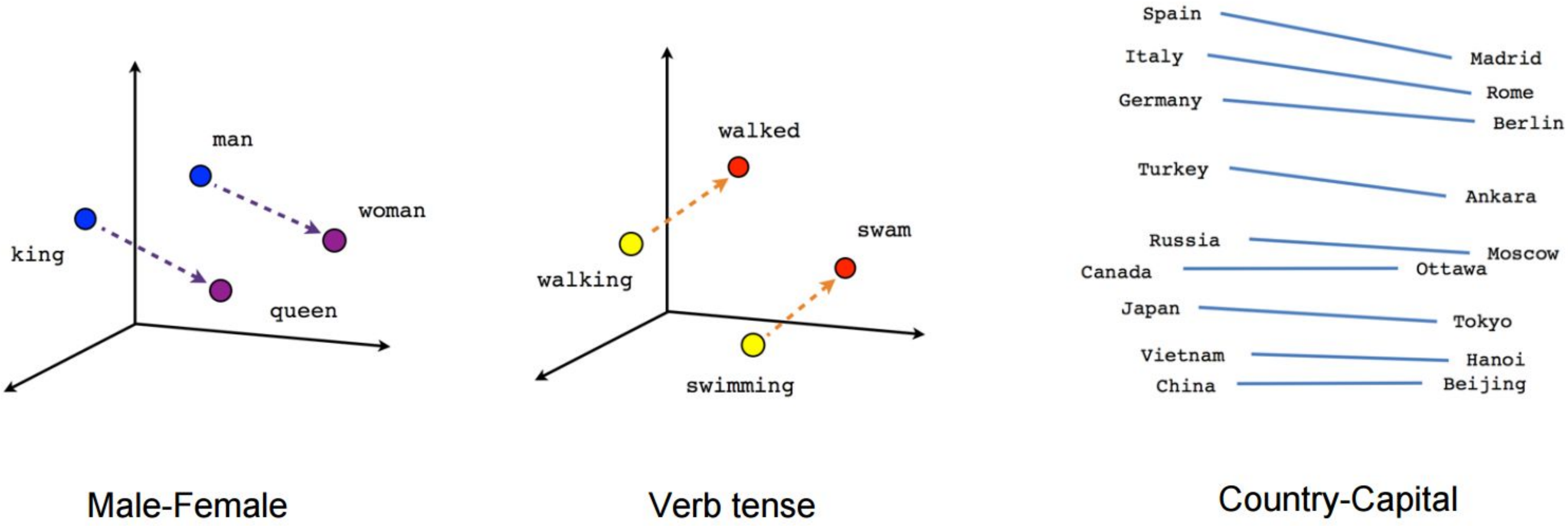


Image from https://medium.com/data-science/mapping-word-embeddings-with-word2vec-99a799dc9695

Word2vec Vector Properties II

Word2vec vectors encode distributional semantics: similarity in meaning → proximity in vector space.

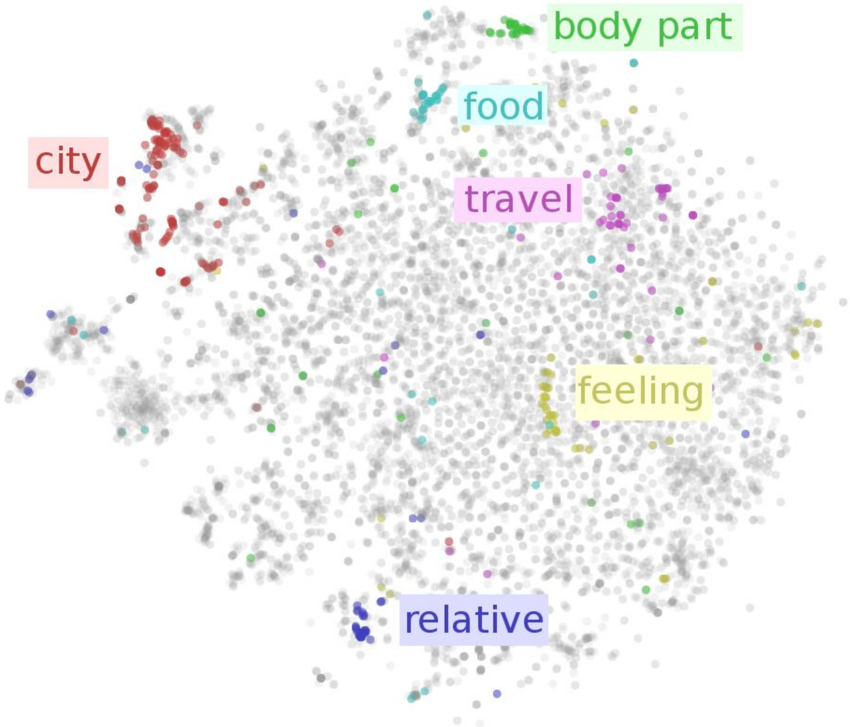


Image from https://www.ruder.io/word-embeddings-1/

Suitable Tasks

Examples of tasks where using pre-trained word embeddings helps:

- Text classification (spam detection, news category)
- Sequence labeling (part-of-speech tagging, named-entity recognition)
- Sentiment analysis
- Document clustering
- Information retrieval / search ranking

Notable Implementations

- word2vec itself probably only used in tutorials such as https://www.tensorflow.org/text/tutorials/word2vec
- GloVe (https://nlp.stanford.edu/projects/glove/)
    - count global coocurrences, i.e. not just within a small window
- fastText (https://fasttext.cc/)
    - takes into account subwords and characters - good for OOVs
    - provide both trained models for many languages and code to train your own

- contextual embeddings, such as BERT
    - produce different word embeddings in different contexts, using a (large) language model

Moodle Quiz

Moodle Quiz

https://dl1.cuni.cz/course/view.php?id=18547