# Generative Models for NLP

## Introduction and Word Embeddings

Joseph Le Roux

30/12/25

# Outline

# Topic

- Introduction
- Word Vectors
- Wor2dVec
- Complements
- From MLE to Contrastive Learning
- Relation MLE / Contrastive
- The End

# NLP

**Various tasks that *process data encoded in natural language* (NL)**

speech recognition, text classification, NL understanding, and NL generation. . .

**At the crossroad of :**

- Linguistics
- Computer Science (formal languages, automata, graphs. . . )
- Logic
- Machine Learning (probability/optimization/statistics)
- Artificial Intelligence
- Cognitive Sciences

# Language as a Symbolic System

## Language

a system that allows a speaker (writer) to communicate (among other functions)

## Elements of this system are discrete (symbols)

- speech/gestures/writings are transctiptions of this system
- if continuous, no writing system (discretization) possible without major loss of meaning

## Problems for Machine Learning

- Requires many different symbols, some very frequent some very rare
- Discrete units make gradient descent impractical

# Why it's difficult

## Ambiguity at all levels

- *saw, duck, her, round, like*, (lexical amb. of words)
- *I saw her duck* (lexical amb. in context)
- *John eats a pizza with a fork* vs. *John eats a pizza with an egg* (syntactic amb., attachment)
- *A computer that understands you like your mother* (amb. syntaxique, rattachement)
- *I seek a unicorn* (semantic amb. existential quantification)
- *avocat pourri* → *dirty lawyer, rotten avocado* (expressions and translation)

## Yes but... we understand each other!

- Context may help, but how?
- world knowledge, social convention, statistics may also help
- other modalities (vision, touch...)

## Great variability

sometimes very ambiguous... in general simple (simple enough for humans to learn!)

# What we will discuss this year

## Session 1: Word Embeddings

*<2026-01-09 Fri>*

- What are the units? how do we represent them?

## Session 2: Language Models

*<2026-01-16 Fri>*

- How do we represent words in context
- How can we represent sequences and generate texts

## Session 3: RNN/Transformers and Attention

*<2026-01-23 Fri>*

- Attention mechanism

## Session 4: Fine-tuning LLMs with RLHF and DPO

*<2026-01-30 Fri>*

- How to incorporate application preferences in LLMs

## Session 5: Latent Variable Models for text: Diffusion Language Models

# Topic

## Definitions

- word (token) = atomic element : *dog*, *car*, *the*, *Trump* ...
  - 🤔 what about letters?
- Vocabulary $V$ finite set of all accepted words
- add a *special token* `UNK`
  - representing all the things in texts that are not words (*ie qwerw3y*)
  - representing all the words that may be missing in $V$
- Lexicon $L_V$ finite set of words or `UNK` : $L_V = V \cup \{\texttt{UNK}\}$
- Language $\mathcal{L}_V$ is the set of strings on the lexicon $L_V^* = \bigcup_{i=0}^{\infty} L_V^i = \{\varepsilon\} \cup L \cup LL \cup \ldots$
- To each element of the lexicon $L$ correponds a unique integer, its *index*, between 0 (for `UNK`) and $|\mathrm{V}|$.

# Word sense – Word meaning

Fundamental issue in NLP: assign a meaning to words

## Different point of views:

- Linguistics: link *signifier* (sound/writing) and the *signified* (the thing/the meaning)
  - not really mechanizable
- Ontology (data/knowledge base): organize a hierarchy of concepts and terms on a semantic basis
  - do not really say anything about usage (how to use words)
- CS/NLP: pragmatic approach, meaning depends on context defined by application
  - EX: if a vocal assistant must perform the same when user says *play the Beatles* or *put the Beatles on* or *read the Beatles*, in this context *play,read,put on* must have the same meaning/representation
  - Moreover their meaning is to *launch the player*

# Multi-dimensional representation (word vectors)

## Assume contexts are countable (and sparse)

Then, if meaning depends on context, and if context can change, we can represent words as vectors with one dimension by context. 🤯

## Synonymity between two words

- Compute *distance* / *inner product* between their vectors.
- *Partial* synonyms, depend on contexts 🤔
- Geometrical Meaning!

## Geometrical Meaning

- Represent words in dense vectors $\mathbb{R}^d$
- Distance/similarity interpreted as semantic relatedness (*eg* synonyms)

Eg. assuming that *play* and *launch* are closer semantically than *play* and *clean*, we have:

$$\|V(\text{play}) - V(\text{launch})\|_2^2 \leq \|V(\text{play}) - V(\text{clean})\|_2^2$$

# (A long time ago...)

## Very different representations in NLP/CL!

Lexicon elements used to be considered as unique by default. Equivalently, they were represented by *one-hot* vectors in $\{0,1\}^{|L_V|}$. For instance:

| | |
|---|---|
| play | $[0000000000010000000]$ |
| read | $[0010000000000000000]$ |
| stop | $[0000000000000000100]$ |

All tokens were orthogonal, (inner product zero, distance $\sqrt{2}$), whether they had the same meaning or not. Geometry meant nothing

# Contextual (or Distributional) Representation

## Contexts as neighbouring words

*You shall know a word by the company it keeps. (J. R. Firth, 1957)*

An old idea in linguistics, used in NLP since 90s/00s
The meaning of a word can be inferred from its *usage*, therefore from the sequence of token in which it appears.
A word is represented by a set of contexts

## Example

For instance for *théorie* in a French text corpus:

| | | |
|---|---|---|
| ... Ezion qui le mettaient en | théorie | à l' abri de soucis ... |
| ... essais assez pointus sur une | théorie | mathématique quelques articles réunis trois ... |
| ... poète météo dépendant pas de | théorie | . |

théorie = { ... Ezion qui le mettaient en, à l' abri de soucis ..., ... essais assez pointus sur une, mathématique quelques articles réunis trois ..., ... }

## Observation

- The more contexts 2 words have in common, the more they share meaning.
- This is not scalable ⇒ contexts must be compressed

# Word Vectors x Distributional Representation = Word Space Models

*Vector similarity is the only information present in Word Space: semantically related words are close, unrelated words are distant. (H. Schütze, 1993)*

In the 90s several methods appear, based on word co-occurrences, with dimension reduction for efficiency and densification.

## Big Picture

1. Associate each lexicon elt $e$ to a vector count $v_e$ of length $|L_V|$, initialized to zero
2. From a set of documents
   - count $n_{e,e'}$ for each lexelt $e$, the number each $e'$ occurs in the neighbourhood of $e$
   - filter: discard tokens such as *a, the...*, limit to a window of $k$ tokens around $e$,
   - set $v_e[id(e')] = n_{e,e'}$
3. Concatenate all vectors $v_e, \forall e$ into a co-occurrence matrix $M$
   - perform some normalization (*eg* TF/IDF or a variant)
4. $M$ is big and sparse
   - use (Truncated) Singular Value Decomposition to get matrix $M'$
   - such as $M'_e$ is a dense vector of dimension $d << |L_V|$

# Word Vectors x Distributional Representation = Word Space Models (2)

## Usage

To decide whether two words are similar semantically, DR usually uses *normalized* inner product also called *cosine similarity*:

$$sim(x, y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}} = \frac{x \cdot y}{(\sqrt{x^2})(\sqrt{y^2})} = \frac{\langle x, y \rangle}{|x| \times |y|}$$

(🤔 what are the min/max values of *sim*)

## Issues with distributional approach

- normalisation, scaling → compression (matrix dimension reduction)
- not incremental → need to retrain from scratch for new data

# Topic

# Word2Vec (2)

## Algorithm to parametrize word vectors

- implements Firth's principle, related to SVD methods (1)
- simple idea (but implementation may be *tricky*)

## Not a *Neural Network* but

- the resulting vectors will be used as input by NNs later
- learning performed via gradient descent
- probabilistic modelling (and approximation)

## A little old. . .

- More powerful recent methods
- . . . but `Word2Vec` is simple and implements Firh's idea quite directly

# WordVec Skip-gram Model

## Principle

1. Assume we have a corpus of texts (big, >1M words) for training;
2. We set vocabulary $V$, out of vocabulary (OOV) words are replaced by token `UNK`;
3. At each position $t$ of a text, we write token $c$ at position $t$ and its neighbouring tokens $O_c$ (words in a window of size $m$ around $t$). We assume a factorized probability to generate neighbouring tokens: $p(O_c|c;\theta) = \prod_{o \in O_c} p(o|c;\theta)$
4. From the computation of similarity between center word vector $v_c$ and context word vectors $v_o$, we will define probability $p_\theta(o|c)$;
5. Training objective: maximize parameters for likelihood: $\prod_c p(O_c|c;\theta)$.

## Loss

Maximizing the log-likelihood amounts to the following loss:

$$\max_\theta \log \prod_t \prod_{o \in O_t} p(o|c_t;\theta) = \max_\theta \sum_t \sum_{o \in O_t} \log p(o|c_t;\theta)$$
$$= \min_\theta \sum_t \sum_{o \in O_t} -\log p(o|c_t;\theta)$$

# Word2vec: Neighbours and Probability

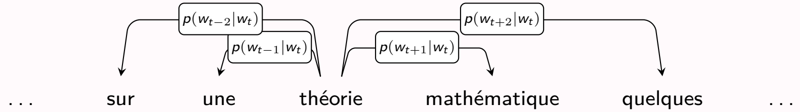
## Window

- Given a position $t$ in text, we write $w_t$, the token at this position
- A window of size $m$ centered at position $t$, noted $O_t$ consists of all words $w_i, \forall (t - m) \leq i \leq (t + m)$

## Word2vec defines probability distributions

- Probability for all words $w'$ to be in a window (of predefined size $m$) of a specific word $w$
- In the remainder, we call $p(w'|w)$ *neighbourhood probability*

## Example



# Word2vec : Maximum Likelihood Estimation (MLE)

General method to cast a ML problem (*eg* optimized by SGD) into a probabilistic framework. Probabilities are computed from scores given by a network with parameters $\theta$

## Definition (Likelihood)

A function returning a corpus probability (assume iid observations) from given parameters:

$$V(\theta) = \prod_{t \in \mathcal{T}} p(O_t | w_t; \theta) = \prod_{t \in \mathcal{T}} \prod_{t-m \leq j \leq t+m} p(w_j | w_t; \theta)$$

## Intuition

Observations must be more probable that any event that we did not observed, and because observations *happened* they must have a probability 1. We want to *maximize* $V$:

$$\theta^* = \arg\max_\theta V(\theta) = \arg\max_\theta \prod_t \prod_{t-m \leq j \leq t+m} p(w_j | w_t; \theta)$$

# Word2vec: From MLE to continuous optimization

## Product → Sum

$V(\theta)$ is a product: difficult to optimize. $\log$ is monotone so we can write:

$$\theta^* = \arg\max_\theta \log V(\theta) = \arg\max_\theta \sum_t \sum_{t-m \leq j \leq t+m} \log p(w_j | w_t; \theta)$$

## Minimization

We prefer minimizing a loss function:

$$\theta^* = \arg\min_\theta - \log V(\theta) = \arg\min_\theta \sum_t \sum_{t-m \leq j \leq t+m} - \log p(w_j | w_t; \theta)$$

## Finally $-\log V(\theta)$ is a loss function

- positive
- zero when no error

# Parametrization of probabilities

## Similarity between words

- We want to measure similarity between token $w_t$ et context token $w_o$
- Similarity of their associated vectors $\rightarrow$ inner product
- Issue with inner product: $v_1^\top v_2$ is maximal when $v_1 = v_2$. we want to avoid trivial solution: $v_i = v_j \forall i, j$
- Solution: define 2 vectors per token mot $m$:
  1. $v_m$ when $m$ is the center position of the window
  2. $u_m$ when $m$ is a context position in the window.

## From similarity to proability

- in $p(w'|w)$, use inner product $u_{w'}^\top v_w$
- we use softmax:

  for a vector $a = [a_1 \dots a_n]$ softmax return a vector:

$$\text{softmax}(a)[i] = \frac{\exp(a[i])}{\sum_j \exp(a[j])}$$

# Parametrization of probabilities (2)

## Inner product + softmax =

$$p(w'|w) = \frac{\exp(u_{w'}^\top v_w)}{\sum_{w''} \exp(u_{w''}^\top v_w)}$$

Softmax implements the idea that the more context $w'$ is similar to center $w$, that the higher $u_{w'}^\top v_w$ the more probable it is to appear in a window.
How to implement?

# First implementation

In lab, you will code an efficient batched version

```
import torch

class Word2Vec(torch.nn.Module):
    def __init__(self, corpus, lexicon_size, vec_size):

        # 2 tables to define from data:
        #word2idx table token -> index
        #idx2word table index -> token

        U = torch.nn.Embedding(lexicon_size, vec_size)
        V = torch.nn.Embedding(lexicon_size, vec_size)

    # returns the sim. score for all token as contexts of w_c
    def forward(self, idx_c):
        # inner product with all vectors in U
        logits = self.U.weight @ self.V(idx_c)
        return logits
```

# First implementation (2)

```
import torch

def train(net, opt, train_set, nb_epoch):

    loss_fn = torch.nn.CrossEntropyLoss()
    # to adapt for lab sessions
    # to deal with batched input
    for epoch in range(nb_epoch):
        train_set.shuffle()
        for (o,c) in train_set:
            logits = net(c)
            loss = loss_fn(logits, o)
            loss.backward()
            opt.step()
            opt.zero_grad()

w2v = Word2Vec(corpus, 10000, 20)
# TODO: conversion between data and training set
#optimizer = ...
train(w2v, optimizer, train_set, 20)
```

# Some issues with this version

Softmax on vocabulary

$$p(w'|w) = \frac{\exp(u_{w'}^{\top}, v_w)}{\sum_{w''} \exp(u_{w''}^{\top}, v_w)}$$

Will not scale for large vocabularies:

- complexity (time/memory) (denominator)
- rounding errors

## In the *real* Word2vec

- many modifications to improve efficiency
- less rounding errors

## Coming up next:

- a better implementation
- interpretation/explanation of the implementation
- beaucoup d'équations 🤯

# Topic

In `Pytorch` learning with MLE uses a function called *cross entropy* 🤔

## Definition (Entropy of a distribution)

a measure of the *randomness* of a distribution

$$H(p) = -\sum_{c \in \mathcal{C}} p(c) \log p(c)$$

- $H(c) \geq 0$
- If $p$ uniform (*ie* random) $H(p) = \log(|\mathcal{C}|)$
- If $p$ deterministic ($p(c_i) = 1$), $H(p) = 0$

# MLE and Cross Entropy (2)

## Definition (Conditional Entropy / Cross Entropy)

Allows to compute a kind of *distance* between 2 distribution 🤔

$$\mathtt{CE}(q, p) = -\sum_c q(c) \log p(c)$$

## Definition (Empirical distribution $p_e$)

- *distribution* of observed data $s = c_i$ always $0/1$
- for a multinomial, if for $p_e(c_i) = 1$ then $\forall j \neq i, p(c_j) = 0$
- (generalizezs to mean for several examples)

## Equivalence

Let us compute the cross-entropy between:

- $q$ the empirical distribution
- $p$ the distribution computed by our model

# MLE and Cross Entropy (3)

$$
\begin{aligned}
\mathtt{H}(q, p) &= -\sum_{c} q(c) \log p(c) \\
&= -\, q(c_i) \log p(c_i) - \sum_{c \neq c_i} q(c) \log p(c) \\
&= -\, 1 \times \log p(c_i) - \sum_{c \neq c_i} 0 \times \log p(c) \\
\mathtt{H}(q, p) &= -\, \log p(c_i)
\end{aligned}
$$

We recover $-\log p(x)$, the loss we defined for MLE 🤯

# Cross-Entropy in Pytorch (1)

```
# classifier for input of size 10, 5 classes
# 8 hidden neurons (cf. DATA MINING)
net = MLP(10,[8],5)

# let us pretend the following tensor contains
# the input for 3 examples
inputs = torch.rand(3,10)

# we obtain the 5 scores for all 3 examples
preds = net(inputs)

# let us suppose that the correct classes were:
empirical = torch.tensor([0,2,1], dtype=torch.long)

loss_function = torch.nn.CrossEntropyLoss()

# note that we do not compute log softmax (inside CE)
loss = loss_function(preds, empirical)

loss.backward() # and the rest...
```

# Cross-Entropy in Pytorch (2)

After training, prediction:

```
# let us pretend the following tensor contains
# the input for 3 examples
inputs = ...

# we obtain the 5 scores for all 3 examples
predictionss = net(inputs)

#apply the argmax for each line:
outputs = torch.argmax(predictionss, dim=1)
```

pas de *cross-entropy*, pas de *softmax* : pourquoi ?

# Topic

# Constrastive Sampling Approach of Mikolov et al. (1)

Softmax inefficient $\rightarrow$ alternative method to train word vectors.

## From *word* probability to *class* probability

Define probability $p(D = d|w', w)$ for 2 classes $d$:

- $d = 1$ if $w'$ is a word that belongs to a possible context for $w$
- $d = 0$ otherwise
- of course we have: $p(D = 0|w', w) = 1 - p(D = 1|w', w)$

## How to model binomial distribution? (2 classes)

- softmax possible but...
- sigmoid more *natural* $\sigma(x) = \frac{1}{1+\exp(-x)}$
- combine sigmoid and similarity:

$$p(D = 1|w', w) = \frac{1}{1 + \exp(-u_{w'} \cdot v_w)}$$

# Constrastive Sampling Approach of Mikolov et al. (2)

## What we want to do

- train vector for tokens from a corpus organised in sentences...
- ... by using a loss function that involves a *similarity* beetween word vectors
- efficient: we want to avoid iterating through the lexicon

## Maximum Likelihood Estimation

For each position $t$ in the training corpus:

- we want to correctly classify $(w', w_t)$ (to 0 or 1) for all $w'$
- we write $k(c, w)$ the class of context candidate $c$ for token $w$ (set to 0 or 1) in the training set. MLE amounts to:

$$\theta^* = \max_\theta \sum_t \mathbb{E}_{w \sim P_{w_t}(.)}[\log p(D = k(w, w_t)|w, w_t)]$$

# Constrastive Sampling Approach of Mikolov et al. (3)

**Maximum Likelihood Estimation: Decompose on positive/negative class expectations:**

$$\theta^* = \max_\theta \sum_t \mathbb{E}_{w\sim P_{w_t}(.)}[\log p(D = k(w, w_t)|w, w_t)] \quad (1)$$

$$= \max_\theta \sum_t \sum_w P_{w_t}(w) \log p(D = k(w, w_t)|w, w_t) \quad (2)$$

$$= \max_\theta \sum_t \sum_w \sum_{i=0}^{1} P(D = i) \times P_{w_t}(w|D = i) \log p(D = k(w, w_t)|w, w_t) \quad (3)$$

$$= \max_\theta \sum_t \sum_{i=0}^{1} \sum_w P(D = i) \times P_{w_t}(w|D = i) \log p(D = k(w, w_t)|w, w_t) \quad (4)$$

$$= \max_\theta \sum_t \sum_{i=0}^{1} P(D = i) \sum_w P_{w_t}(w|D = i) \log p(D = k(w, w_t)|w, w_t) \quad (5)$$

$$= \max_\theta \sum_t \sum_{i=0}^{1} P(D = i)\mathbb{E}_{w\sim P^i_{w_t}(\cdot)}[\log p(D = k(w, w_t)|w, w_t)] \quad (6)$$

$$= \max_\theta \sum_t \sum_{i=0}^{1} P(D = i)\mathbb{E}_{w\sim P^i_{w_t}(\cdot)}[\log p(D = i|w, w_t)] \qquad \text{why ?} \quad (7)$$

🤯

# Constrastive Sampling Approach of Mikolov et al. (4)

Add the following assumptions:

## There are more negative examples than positive examples

- $\exists \kappa = 1, 2, \ldots, N$ such that $P^{-}(D = 0) = \kappa \times P^{+}(D = 1)$
- in other words, the negative class is $\kappa$ times more likely than the positive class

## Expectations can be approximated by sampling efficiently (Monte-Carlo!)

For a position $t$, we want to maximize:

$$\mathbb{E}_{w \sim P_t^1(.)}[\log p(D = 1|w, w_t)] + \kappa \mathbb{E}_{w \sim P_t^0(.)}[\log p(D = 0|w, w_t)]$$

Expectations $\rightarrow$ Sampling:

$$\log p(D = 1|w^{+}, w_t)] + \sum_{i=1}^{\kappa} \log p(D = 0|w^{-i}, w_t)]$$

- $w^{+}$ drawn randomly from distribution $P^{+}_{w_c}$
- $w^{-i}$ drawn randomly from ditribution $P^{-}_{w_c}$

# Constrastive Sampling Approach of Mikolov et al. (5)

We also need to define distributions $P^+$ et $P^-$

## For the positive class

draw randomly a token from the window around position $t$

$$P_t^+(w) = \begin{cases} \frac{1}{|O_t|} & \text{if } w \in O_t, \\ 0 & \text{otherwise} \end{cases}$$

## For the negative class

Use the frequency of a word in training set as probability:

$$P_t^-(w) = \frac{\#w}{\sum_{w'} \#w'} = f(w)$$

In Mikolov's implementation, use a *flattened* version:

$$P_t^-(w) = \frac{\#w^{\frac{3}{4}}}{\sum_{w'} \#w'^{\frac{3}{4}}}$$

Finally we obtain the Mikolov's loss:

$$\max \sum_{t} \left( \log \sigma(u_{w+} \cdot v_{w_t}) + \sum_{i=1}^{\kappa} \log \sigma(-u_{w-_i} \cdot v_{w_t}) \right)$$

- This is a maximization: we get the loss function by multiplying by $-1$

A small trick to get a better model:

## Subsampling

Do not take into account all the words in the training set:

- remove all words that appear less than $n$ times (set $n = 2, 3, 4, 5 \ldots$)
- remove at random very frequent words in contexts. For each token $w$ in the lexicon, eliminate $w$ from $O_t$ with probability:

$$P_d(w) = ReLU(1 - \sqrt{\frac{t}{f(w)}})$$

with $t = 10^{-5}$

- in words, words with frequency equal to or above $t$ are always discarded.
- beware if we remove a word from context $O_t$, we still have to keep the size of the window $(2m)$: need to extend window boundaries

# Topic

- Introduction
- Word Vectors
- Wor2dVec
- Complements
- From MLE to Contrastive Learning
- Relation MLE / Contrastive
- The End

### Another way to look at contrastive estimation of word vectors

- a series of approximations/assumptions from the MLE model
- sheds new light on the relation between the two

Let us derive the gradient for one example:

$$
\begin{aligned}
\nabla L(\theta; w', w) &= \nabla - \log p(w'|w;\theta) = \nabla - \log \frac{\exp(u_{w'}^\top v_w)}{\sum_{w''} \exp(u_{w''}^\top v_w)} \\
&= \nabla \Big( \log \big( \sum_{w''} \exp(u_{w''}^\top v_w) \big) - u_{w'}^\top v_w \Big) \\
&= \nabla \Big( \log \big( \sum_{w''} \exp(u_{w''}^\top v_w) \big) \Big) - \nabla (u_{w'}^\top v_w) \\
&= \frac{\nabla \sum_{w''} \exp(u_{w''}^\top v_w)}{\sum_{w''} \exp(u_{w''}^\top v_w)} - \nabla u_{w'}^\top v_w \\
&= \frac{\sum_{w''} \nabla \exp(u_{w''}^\top v_w)}{\sum_{w''} \exp(u_{w''}^\top v_w)} - \nabla u_{w'}^\top v_w \\
&= \frac{\sum_{w''} \exp(u_{w''}^\top v_w) \nabla u_{w''}^\top v_w}{\sum_{w''} \exp(u_{w''}^\top v_w)} - \nabla u_{w'}^\top v_w \\
&= \sum_{w''} p(w''|w;\theta) \nabla u_{w''}^\top v_w - \nabla u_{w'}^\top v_w \\
&= \mathbb{E}_{w'' \sim p(.|w;\theta)} [\nabla u_{w''}^\top v_w] - \nabla u_{w'}^\top v_w
\end{aligned}
$$

# From Definition to Implementation: A New Objective

## Recap: we want $\nabla L(\theta; w', w) = 0$

- equivalently: $\mathbb{E}_{w'' \sim p(.|w;\theta)}[\nabla u_{w''}^\top v_w] - \nabla u_{w'}^\top v_w = 0$
- But this gradient is also the gradient of:

$$F(\theta; w', w) = \mathbb{E}_{w'' \sim \widehat{p(.|w;\theta)}}[u_{w''}^\top v_w] - u_{w'}^\top v_w$$

- where $\widehat{x}$ means $x$ is considered a constant (null gradient)

Conclusion We can learn with $F$ instead of $L$ (they have the same solution)

## Wait... we still have to iterate over all words (expectation)

- Idea: sample only a few words, not all the lexicon

$$\text{With } w_k \sim p(.|w;\theta) : \qquad F(\theta; w', w) \approx \frac{1}{K} \sum_{k=1}^{K} [u_{w_k}^\top v_w] - u_{w'}^\top v_w$$

with $w_k$ sampled from $p$

# From Definition to Implementation: Efficient Sampling

$$\text{With } w_k \sim p(.|w;\theta): \qquad F(\theta; w', w) \approx \frac{1}{K}\sum_{k=1}^{K}[u_{w_k}^\top v_w] - u_{w'}^\top v_w$$

## Problems when sampling with $p$

To sample with $p$, we need to compute the denominator and iterate over the lexicon...

## Solution

- Find an efficient approximation to $p(.|.;\theta)$
- `word2vec` uses the frequency of words in training corpus

$$P_u(w) = \frac{\#w}{\sum_{w'} \#w'}$$

- or a flattened version

$$P'_u(w) = \frac{\#w^{0.75}}{\sum_{w'} \#w'^{0.75}}$$

# From Definition to Implementation: Rectification (1)

$$\text{With } w_k \sim P'_u(.) : \qquad F(\theta; w', w) \approx \frac{1}{K}\sum_{k=1}^{K}[u_{w_k}^{\top} v_w] - u_{w'}^{\top} v_w$$

## Recall : inner product ≈ vector similarity

- $u_{w'}^{\top} v_w$ must be positive (it's similar!)
- $u_{w_k}^{\top} v_w$ must be negative (it's not an observation!)

## Use a Rectifier

We ignore the result of an inner product if it does not have the expected sign

- We could use `ReLU(x)=max(x,0)` (maybe. . . )
- in `word2vec` use a differentiable variant `softplus`

$$\text{sp}(x) = \log(1 + \exp(x))$$

- pay attention to minus signs - in the definition of $F$

# From Definition to Implementation: Rectification (1)

$$F(\theta; w', w) \approx \frac{1}{K}\sum_{k=1}^{K}[-sp(-u_{w_k}^\top v_w)] - sp(u_{w'}^\top v_w)$$

$$\frac{1}{K}\sum_{k=1}^{K}[-\log(1+\exp(-u_{w_k}^\top v_w))] - \log(1+\exp(u_{w'}^\top v_w))$$

$$\frac{1}{K}\sum_{k=1}^{K}[\log(\frac{1}{1+\exp(-u_{w_k}^\top v_w)})] + \log(\frac{1}{1+\exp(u_{w'}^\top v_w)})$$

$$\frac{1}{K}\sum_{k=1}^{K}[\log(\sigma(u_{w_k}^\top v_w))] + \log(\sigma(-u_{w'}^\top v_w))$$

## Can we recover contrastive estimation??

With $w_k \sim P'_u(.)$: $$F(\theta; w', w) \approx \frac{1}{K}\sum_{k=1}^{K}[\log\sigma(u_{w_k}^\top v_w)] + \log\sigma(-u_{w'}^\top v_w)$$

One last step to recover the signs of contrastive estimation

# Topic

# Conclusion

## Word2vec

- Simple Idea, implements Firth's principle in an efficient way
- Probabilistic modelling : MLE/Constrative
- Many tricks to scale up:

## Extensions

*Dynamic* Vectors: Word Vectors + Functions (NNs) to adapt Word vectors to each context (ElMO, BERT...)

# Bibliography

Levy, Omer and Goldberg, Yoav (2014). *Neural Word Embedding as Implicit Matrix Factorization*, Curran Associates, Inc..
Mikolov, Tomas and Grave, Edouard and Bojanowski, Piotr and Puhrsch, Christian and Joulin, Armand (2018). *Advances in Pre-Training Distributed Word Representations*, European Language Resources Association (ELRA).