Neural Language Models and Word Embeddings

Matthieu Labeau - Associate Professor, Telecom Paris, IPP

SD-TSIA-214: Introduction to Language Processing - 05th April 2024

Outline

- **Neural Probabilistic Language Models**
- **Learning word embeddings**
- **NLP from scratch**
- **Neural architectures for sequence processing**

Reminder: Difficulties with symbolic representations

- Vectors can get huge: memory and computation issues
- Vectors are **sparse**
- All dimensions (representing words) have the same importance
- Skewed frequency is always a challenge

- We can avoid all of these issues by using **dense**, **distributed** representations: we would like to replace:

this . . .   . . . *by this.*

$$\mathbf{word} = \boxed{\begin{matrix}0\\0\\1\\0\\0\\0\\0\end{matrix}} \in \mathbb{N}^{|\mathcal{V}|} \qquad \mathbf{word} = \boxed{\begin{matrix}0.212\\0.792\\-0.177\\0.109\\-0.542\\0.349\\0.271\end{matrix}} \in \mathbb{R}^{d}$$

- With $d << |\mathcal{V}|$
- Words appear in the **same space** and a similarity between them can be interpreted.

Better representations: dimension reduction

- Cosine similarity on symbolic representations does not work well: should all dimensions (represented by words) matter the same ?
- How can we represent documents by doing **more than counting words** ?

Idea: take advantage of the latent structure in the association between the set of words and the set of documents

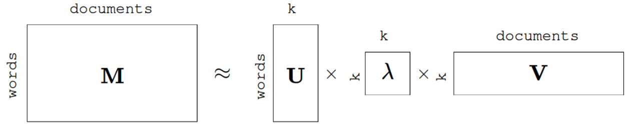
- First method: linearly reduce the dimension to put forward higher-order relationships
  $\rightarrow$ Use Singular Value Decomposition (SVD)

$$\mathbf{M} = \mathbf{U}\lambda\mathbf{V}^{\intercal}$$

  - λ diagonal matrix, with eigenvalues - ordered
  - $\mathbf{U}, \mathbf{V}$ orthogonal; eigenvectors
  - Keep the k first columns of $\mathbf{U}$, for the k largest eigenvalues, to obtain embeddings $\in \mathbb{R}^k$
  - But very costly (quadratic in memory, cube in flops)

Latent Semantic Analysis

This method is called **Latent Semantic Analysis**:



- The new space is interpreted as **topics**: this is the first method for *topic modeling*
- Reminder: SVD rotates the axis along directions of largest variations among the documents (generalized least-squares method)
- Useful for information retrieval, but also if we need **higher-order features than word counts**
- Can help to represent documents in topic space for classification !
  - Besides the cost, it must be re-run if you add new documents.

More on Topic Modeling

Other topic modeling methods: mostly **generative models** - we model the generation of words as *random, following a distribution*

- **Probabilistic LSA**: the generation of words follows a mixture of *conditionally independant multinomial distributions*, given topics:

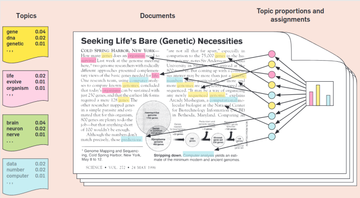
$$P(w,d) = \sum_t P(t)P(d|t)P(w|t) = P(d)\sum_t P(t|d)P(w|t)$$

  where $P(d|t)$ relates to V and $P(w|t)$ to U: non-negative values

- **Latent Dirichlet Allocation (LDA)**:

  We assume that the topic distribution has a *Dirichlet prior* (a family of continuous multivariate distributions)

Probablistic Topic Models (Blei, 2012)

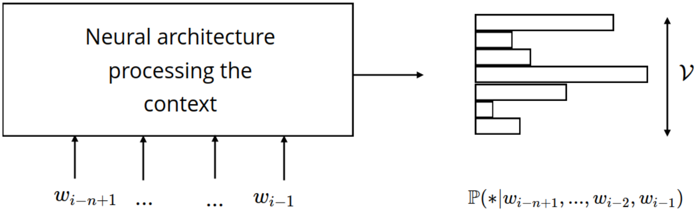


Neural Probabilistic Language Models

n-gram *neural* models

- Instead of computing $\mathbb{P}_\theta(w_i | w_{i-n+1}, ..., w_{i-1})$ with corpus statistics, we can teach a neural network to **predict** these probabilities

- This is divided in two parts:
  - Processing the context words - how it is done depends on the neural architecture used:



  - Obtaining an output probability distribution for the next word - it's the same whatever the model, we are classifying over $\mathcal{V}$

NPLM: A first model

A Neural Probabilistic Language Model (Bengio et al, 2003)

A similar model was first applied to speech recognition (Schwenk and Gauvain, 2002) and machine translation.

Main ideas:

- **Continuous word vectors**: Each input and output word is represented by a vector of dimension $d \ll |\mathcal{V}|$ taking values in $\mathbb{R}$, rather than being discrete
- **Continuous probability function**: The probability of the next word is expressed as a continuous function of the features of the word in the current context - using a neural network
- **Joint learning**: The parameters of the word representations, and the probability function are learnt jointly.

NPLM: Joint learning

Why should it work ?

- **Continuity**: the probability function is smooth, implying that a small change in the context or word vector will induce a small change in word probability.
- **Distributional hypothesis**: words appearing in similar contexts should have similar representations.

- Hence, the updates caused by having the following sentence:

  A dog was running in a room

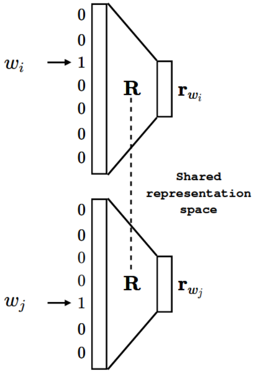
  in the training data will increase the probability of all 'neighbor' sentences. Having also the following sentence:

  The cat was running in a room

  will make the features of the words (dog, cat) get close to each other.

Neural model: projecting words

- Create a layer that is the vocabulary $\mathcal{V}$: the input is a **one-hot vector**.
- This layer is densely connected to a smaller continuous layer, of dimension d_w
- The parameters of the weight matrix $\mathbf{R}$ are what we call the **word embeddings**.
- Now, we assume working with a 3-gram (w_{i-2}, w_{i-1}, w_i): we need to project the two first words to predict the third.

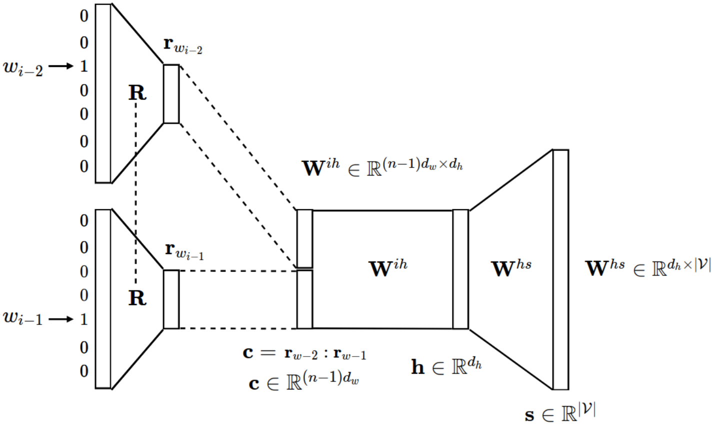


Neural model: Obtaining scores

- Given the context representation $\mathbf{c}$, create a hidden representation:

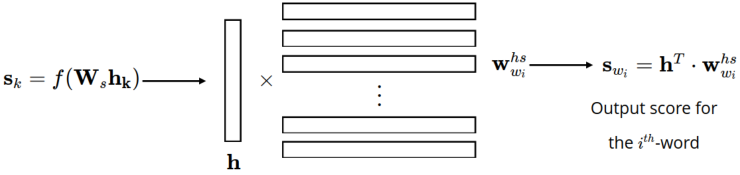
$$\mathbf{h} = \phi(\mathbf{W}^{ih}\mathbf{c})$$

- Then, obtain scores for all words in $\mathcal{V}$ given $\mathbf{h}$: $\mathbf{s} = \mathbf{W}^{hs}\mathbf{h}$



Neural model: Obtaining scores

The prediction layer can be seen as a dot product between $\mathbf{h}$ and output word embeddings $[\mathbf{w}_k^{hs}]_{k=1}^{|\mathcal{V}|}$:

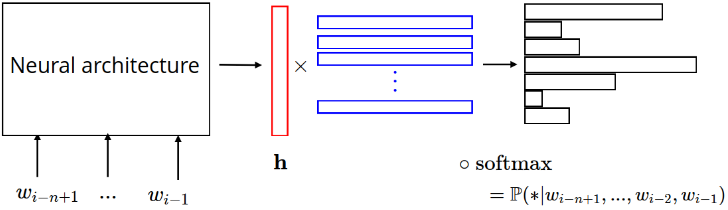


Neural model: Computing probabilities

- The goal of the neural architecture is here to get a vector representation $\mathbf{h}_i$ for the context input words $w_{i-n+1}, ..., w_{i-1}$.

- We use this representation against vector representations of all possible output o words $\mathbf{w}_o^{hs}$ in $\mathcal{V}$ to estimate their probabilities. We use the softmax function:

$$\mathbb{P}(o|w_{i-n+1}, ..., w_{i-1}) = \frac{\exp(\mathbf{h}_i^T \mathbf{w}_o^{hs})}{\sum_{l=1}^{|\mathcal{V}|} \exp(\mathbf{h}_i^T \mathbf{w}_l^{hs})}$$



Neural model: Training

- The input representations, hidden weights, and output representations are learned jointly: $\theta = (\mathbf{R}, \mathbf{W}^{ih}, \mathbf{W}^{hs})$

- We want the model output probability distribution $\mathbb{P}_\theta$, at timestep (i), to get close to the ground truth:

$$\mathbb{P}_\theta^{(i)}(*|w_{<i}) \rightarrow \text{ one-hot}(w_i) = \mathbb{P}_*^{(i)}$$

  since we know that the next word is w_i.

- We look to minimize the distance between the two distributions:

$$d\left(\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbb{P}_\theta^{(i)}(1|w_{<i}) \\ \vdots \\ \mathbb{P}_\theta^{(i)}(w_i|w_{<i}) \\ \vdots \\ \mathbb{P}_\theta^{(i)}(|\mathcal{V}||w_{<i}) \end{bmatrix} \right) ? \longrightarrow \begin{aligned} &\text{Cross-entropy}(\mathbb{P}_*^{(i)}, \mathbb{P}_\theta^{(i)}(*|w_{<i})) \\ &= -\sum_{k=1}^{|\mathcal{V}|} \mathbb{P}_*^{(i)}(k) log(\mathbb{P}_\theta^{(i)}(k|w_{<i})) \\ &= -log(\mathbb{P}_\theta^{(i)}(w_i|w_{<i}))\ ! \end{aligned}$$

Neural model: Training

- By minimizing this cross-entropy, at each step, we minimize the **negative log-likelihood** of the data sample $(w_i, w_{<i})$
- On all data samples $\mathcal{D} = (w_i)_{i=1}^m$, this is the Maximum-Likelihood Estimation (MLE) objective:

$$NLL(\theta) = -\sum_{i=1}^{m} \log(\mathbb{P}_\theta^{(i)}(w_i|w_{<i}))$$

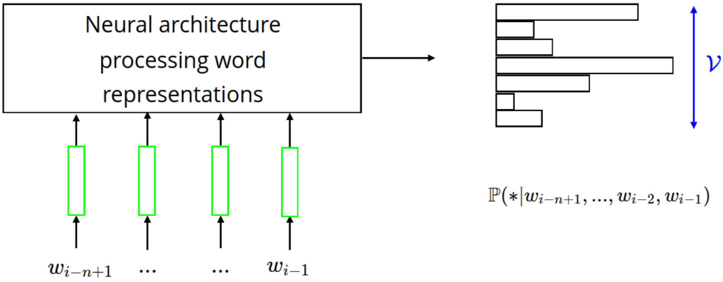
- Remark: minimizing the cross-entropy is equivalent to minimizing the Kullback-Leibler divergence
- We can note again that the perplexity is a simple function of the cross-entropy:

$$\text{Perplexity}(w_1, ..., w_m) = \sqrt[n]{\prod_{i=1}^{m} \frac{1}{\mathbb{P}_\theta^{(i)}(w_i|w_{<i})}} = 2^{-\frac{1}{m}\sum_{i=1}^{m} \log(\mathbb{P}_\theta^{(i)}(w_i|w_{<i}))}$$

$\rightarrow$ We are directly minimizing perplexity when training

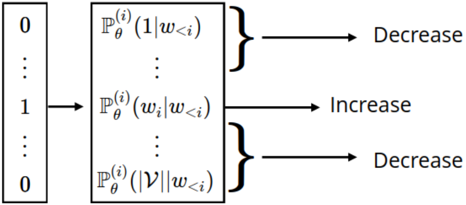
Neural model: Assessment

- Probability estimation is based on the similarity among the word vectors
- Projecting in continuous spaces reduces the **sparsity issue**
- Increasing the number of input words does not change much the complexity of the model
- The bottleneck is the output vocabulary size !



Neural model: Learning bottleneck

- During learning, probabilities are modified as follow:



The updates are computed using the following gradient:

$$\frac{\delta}{\delta\theta}\log(\mathbb{P}_\theta^{(i)}(w_i|w_{<i})) = \frac{\delta}{\delta\theta}\mathbf{s}_{w_i} - \sum_{w\in\mathcal{V}} \mathbb{P}_\theta^{(i)}(w|w_{<i})\frac{\delta}{\delta\theta}\mathbf{s}_w$$

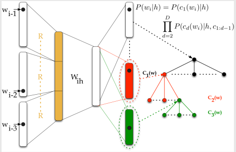
- The first term increases the conditional log-likelihood of w_i given $w_{<i}$
- The second decreases the conditional log-likelihood of all the other words $w \in \mathcal{V}$ - and implies a double summation on $\mathcal{V}$

$\rightarrow$ The softmax causes this computational bottleneck !

Parenthesis: dealing with the bottleneck ?

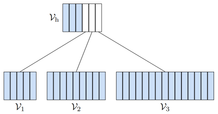
- Making hierarchical predictions: replace complexity in $O(|\mathcal{V}|)$ by $O(\log|\mathcal{V}|)$

Structured Output Layer neural network Language Model (Le et al, 2012)



- Using **sampling-based** methods, to replace the sum over $|\mathcal{V}|$ by a sum over k samples ($k \ll |\mathcal{V}|$)
- More recently: implement a class-based softmax, based on word frequencies, and attribute more parameters to frequent words

Efficient softmax approximation for GPUs (Grave et al, 2016)



Learning Word Embeddings

Learning Word Embeddings: why ?

- Text vector representations exist in several forms:

From Frequency to Meaning: Vector Space Models of Semantics (Turney et al, 2010)

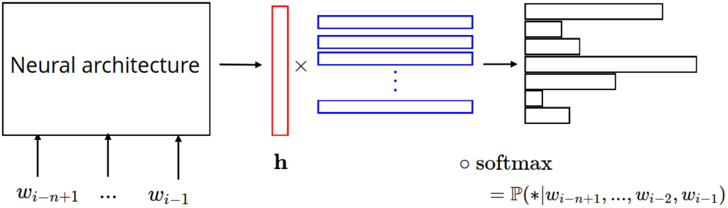
  - From text-document matrices (*TF-IDF, LSA..*)
  - From word-context matrices (*Co-occurences, PPMI..*)
  - From more complex relationnal patterns, that can handle word order

- All based on simply storing frequencies in a big tensor
  $\rightarrow$ Still, all of these are **costly**

- But Neural n-gram Language models **learn good dense representations**.
  Can we use them ?

Learning Word Embeddings: why ?

- Idea: **prediction-based representation learning**
- Basically, language modeling - but we don't care about generating text
- What was costly in our neural language model ?

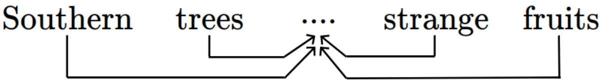


- We can work on two things: the **architecture** and the **softmax computation**
  - Hypothesis: the distribution of the context is what matters $\rightarrow$ **simplify the architecture** !
  - Goal: learning representation $\rightarrow$ **no need for a proper softmax** !

Learning Word Embeddings: Architecture

First, the architecture: for *'Southern trees bear strange fruits'*

- Use all context: $\mathbb{P}(w_t | w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$?



- Let's simplify it as much as possible. Assuming:
  - Context word representations $\mathbf{C} \in \mathbb{R}^{d \times |\mathcal{V}|}$
  - Output word representations $\mathbf{W} \in \mathbb{R}^{d \times |\mathcal{V}|}$

$$\mathbf{h} = \mathbf{C} \times \left(\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right) \quad (\in \mathbb{R}^d)$$

$$\mathbf{o} = softmax(\mathbf{h} \times \mathbf{W}) \quad (\in \mathbb{R}^{|\mathcal{V}|})$$

Learning Word Embeddings: CBOW

This is the **Continuous bag-of-words** (CBOW) architecture

- Output:

$$\mathbb{P}(\text{bear}|w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}) = \mathbf{o}_{bear} = \frac{\exp(\mathbf{h}^T \mathbf{c}_{bear})}{\sum_{l=1}^{|\mathcal{V}|} \exp(\mathbf{h}^T \mathbf{c}_l)}$$

- Training:

$$\mathbb{P}(\text{bear}|w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}) = \mathbf{o}_{bear} \rightarrow \text{ one-hot(bear)} = \mathbb{P}_*^{(\text{bear})}$$

- Noting $\theta = \{\mathbf{W}, \mathbf{C}\}$ the parameters of the model:

$$d\left(\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbb{P}_\theta(\text{word}_1) \\ \vdots \\ \mathbb{P}_\theta(\text{bear}) \\ \vdots \\ \mathbb{P}_\theta(\text{word}_{|\mathcal{V}|}) \end{bmatrix} \right) ? \longrightarrow \begin{aligned} &\text{Cross-entropy}(\mathbb{P}_*^{(\text{bear})}, \mathbb{P}_\theta) \\ &= -\sum_{k=1}^{|\mathcal{V}|} \mathbb{P}_*^{(\text{bear})}(k) \log(\mathbb{P}_\theta(k)) \\ &= -\log(\mathbb{P}_\theta(bear)) = -\log(\mathbf{o}_{\text{bear}}) \,! \end{aligned}$$

- Minimizing this cross-entropy $\Leftrightarrow$ minimize the negative log-likelihood of the data sample *'Southern trees bear strange fruits'*

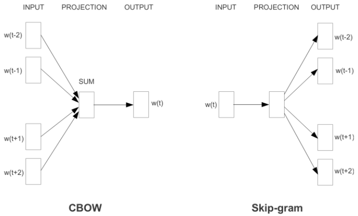
Learning Word Embedding: objective functions

- With a dataset $\mathcal{D} = (w_i)_{i=1}^{N}$ and a window of m words,

$$J_{MLE}^{CBOW}(\theta) = -\sum_{i=1}^{N} \log(\mathbb{P}_\theta(w_i | w_{i-m}, ..., w_{i-1}, w_{i+1}, ..., w_{i+m}))$$

- Other possible architecture, the **Skip-gram**:

$$J_{MLE}^{SG}(\theta) = -\sum_{i=1}^{N} \sum_{j \neq 0}^{-m<j<m} \log(\mathbb{P}_\theta(w_{i+j} | w_i))$$



(*What would be the interest of using this one ?*)

Learning Word Embedding: Too slow

Reminder: **softmax gradient updates are slow and costly**: with the skip-gram,

$$\frac{\delta}{\delta\theta}\log(\mathbb{P}_\theta(w_{i+j}|w_i)) = \frac{\delta}{\delta\theta}\mathbf{s}_{w_{i+j}} - \sum_{k=1}^{|\mathcal{V}|}\mathbb{P}_\theta(w_k|w_j)\frac{\delta}{\delta\theta}\mathbf{s}_{w_k}$$

- The first term increases the conditional log-likelihood of w_{i+j} given w_i
- The second decreases the conditional log-likelihood of all $w_k \in \mathcal{V}$

How to avoid computing any $\sum_{k=1}^{|\mathcal{V}|}$?

- Replace the task by **binary classification**: predicting the right word ?

$$\mathbb{P}_\theta(w_{i+j}|w_i) = \sigma(\mathbf{s}_{w_{i+j}})$$

- Only provides the **positive contribution** to the conditional log-likelihood:

$$\frac{\delta}{\delta\theta}\log(\mathbb{P}_\theta(w_{i+j}|w_i)) = (1-\sigma(\mathbf{s}_{w_{i+j}}))\frac{\delta}{\delta\theta}\mathbf{s}_{w_{i+j}}$$

- Let's add the negative contribution by **sampling** $k \ll |\mathcal{V}|$ **wrong words**

Learning Word Embedding: Negative sampling

From $\mathbb{P}_*$ $\quad w_i$ $\quad\quad$ From $\mathbb{P}_{noise}$

w_{i+j} $\quad\quad (w_l^{noise})_{l=1}^{k}$

$\mathbf{s}_{w_{i+j}}$ $\quad\quad \mathbf{s}_{w_l^{noise}}$

$\mathbb{P}(Y = 1|w_i) = \sigma\left(\mathbf{s}_{w_{i+j}}\right)$ $\quad\quad \mathbb{P}(Y = 0|w_i) = \sigma\left(\mathbf{s}_{w_l^{noise}}\right)$

Log-Likelihood of $Y = 1$ $\quad$ **Binary Cross-Entropy** $\quad$ **Log-Likelihood of $Y = 0$**

$$J_{NS}^{SG}(\theta) = -\sum_{i=1}^{N}\sum_{j\neq 0}^{-m<j<m}\left(\log\sigma(\mathbf{s}_{w_{i+j}}) + \sum_{l=1}^{k}\log\sigma(\mathbf{s}_{w_l^{noise}})\right)$$

$$= -\sum_{i=1}^{N}\sum_{j\neq 0}^{-m<j<m}\left(\log\sigma(\langle\mathbf{w}_{w_{i+j}}, \mathbf{c}_{w_i}\rangle) + \sum_{l=1}^{k}\log\sigma(-\langle\mathbf{w}_{w_l^{noise}}, \mathbf{c}_{w_i}\rangle)\right)$$

- Very efficient, but requires some tricks: **subsampling of frequent words** in the noise distribution (*why ?*)

$$\mathbb{P}_{noise}(w) = (\mathsf{freq}(w))^{\frac{3}{4}}$$

Count-based vs Prediction-based

- **Prediction-based** methods, like word2vec, are:
  - Fast and scale well with available data
  - Are dense and capture complex patterns

But, they require a lot of data and are not using all the statistical information available

- **Count-based** methods can also give us **dense representations** !
  $\rightarrow$ For example, apply SVD to a PMI matrix, like we did for LSA
  - This is pretty fast
  - It uses efficiently all available information and works with little data

But, it does not scale well (in memory) and large frequencies create issues

- Can we get the best of both worlds ?

GloVe

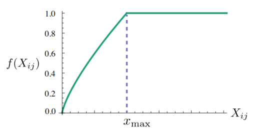
Central idea: **directly learn word embeddings by predicting word co-occurence counts** !

GloVe: Global Vectors for Word Representation (Pennington et al, 2014)

$$J_{Glove}(\theta) = \sum_{w_i, w_j \in \mathcal{V}} f\left(\mathbf{M}_{w_i,w_j}\right)\left(\mathbf{w}_{w_i}^{\top}\mathbf{w}_{w_j} - \log \mathbf{M}_{w_i,w_j}\right)^2$$

- $\theta = \{\mathbf{W}\}$
- f : scaling function - diminish the importance of frequent words

$$f(x) \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

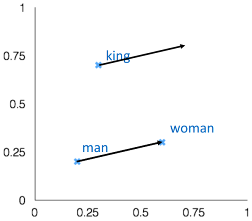


- Very fast training
- Scales to very large corpora

Properties: Analogy

- We can observe that **linear** word representations relation ships can capture meaning: for example,

$$\text{queen} = \underset{w_i \in \mathcal{V}}{\text{argmax}}[cos(\mathbf{w}_{w_i}, \mathbf{w}_{woman} - \mathbf{w}_{man} + \mathbf{w}_{king})]$$



- This also applies to other patterns in language. Let's check !
  $\rightarrow$ Look at the *visualization* notebook.

Properties: Bias

- Word embeddings have been found to reflect biases
- A lot of recent efforts dedicated to detect and reduce them
    - At first, focused on **gender bias**

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings (Bolukbasi et al, 2016)

- **Idea**: Find *gender axis* by reducing the dimension on a set of words differences ("she" - "he", "her" - "him", etc...)
- Project a word on this direction to quantify its bias
- Basis for WEAT (*Word Embeddings Association Test*)

- It does not work that well - a lot of work since: *Debiasing Methods Cover up Systematic Gender Biases in Word Embeddings But do not Remove Them (Gonen and Goldberg, 2019)*
- Bias is usually poorly defined in NLP - some recommendations on approaching it: *Language (Technology) is Power: A Critical Survey of "Bias" in NLP, Blodgett et al, 2020*

Difficulty: lexical ambiguity

How to account for the different meanings of **polysemous** words ?

- Same issue with **homonyms**
- Most word are monosemous but words with multiple senses tend to have higher frequency
- Senses can be very different or only subtly (*sense granularity*)
- For embeddings: **word meaning conflation** into a single representation. Solutions ?
  - **Sense embeddings**: one vector by word sense
  - **Sparse coding**: separating word senses *inside* the embedding
  - **Contextualized embeddings**: for next class !

Subword models

- Issue: vocabulary is closed - especially a problem when spelling varies
- Also, we are missing a lot of information linking words

- Linguistically motivated decomposition:
  - **Phonemes** (distinctive features in audio), **morphemes** (smallest semantic unit)
  - But it's costly !

- Let's use characters sequences: **Fastext** (Word2Vec + subwords)

Enriching Word Vectors with Subword Information (Bojanowski et al, 2016)

- Goal: improve on one of Word2vec main weakness, **rare words**
- Words are represented as character n-grams:

$$\text{where} \;\rightarrow\; < \text{wh}, \text{whe}, \text{her}, \text{ere}, \text{re} >$$

- The representations is simply the sum of the n-gram representations

NLP from Scratch

Pre-neural NLP: Feature engineering

- Supervised Learning in NLP:
  - Small datasets, for tasks that can be very difficult
  - Even with neural models, better performance from **task-specific features**

- Some examples:
  - Part-of-speech tagging (POS)

I	ate	the	spaghetti	with	meatballs	.
Pro	V	Det	N	Prep	N	PUN

  - Chunking

[NP I] [VP ate] [NP the spaghetti] [PP with] [NP meatballs]

  - Named-entity Recognition (NER)
  - Semantic-role labeling (SRL)

(Agent Patient Source Destination Instrument)

John	drove	Mary	from	Austin	to	Dallas	in	his	DS Citroën
A		P		S		D			I

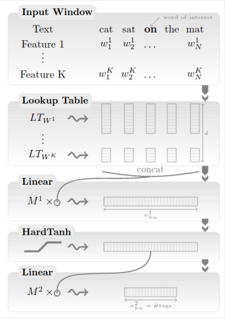
NLP from scratch

Natural language processing (almost) from scratch (Collobert et al, 2011)

- **Goal**: Build a model that excels on multiple benchmarks, without needing task-specific representations or engineering
- A **task-independent** approach is better because no single task can provide a complete representation of a text

Paper:

- Language-modeling-like training, to **rank unlabelled sentences**
- Then, train the model on the different tasks
- Features: n-grams ! Beating state-of-the art on most tasks



Framework: pre-training + fine-tuning

In summary: parameters of *text predicting* models **represent text very well**
$\rightarrow$ why not "pre-train" other model them to do the same before their task ?

- We can do **unsupervised pre-training** of a generative sequence prediction model:

$$\mathbb{P}_\theta(x_{1:T}) = \prod_{i=1}^{T} \mathbb{P}_\theta(x_i | x_{1:i-1})$$

- We follow-up with **supervised fine-tuning** of a classifier on the target task, with the annotated dataset we have:

$$\mathbb{P}_\phi(y | x_{1:T})$$

  - Initialize part of ϕ with part of θ
  - *Freeze* θ then extract representations, or *fine-tune* θ into ϕ

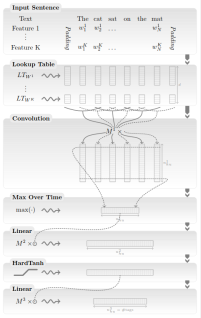
- *NLP from scratch*: first occurence of the idea
- Word embeddings obtained with word2vec and GloVe: hugely popular, improvements on many (many) tasks

What's next ?

Natural language processing (almost) from scratch (Collobert et al, 2011)

Paper, continued:

- Not working well on tasks requiring the full sentence
- Need to use the full sentence as input
- Going further than n-grams: how ?



- Next: **Neural architectures better adapted to textual data: in Deep Learning class**

And representing a sequence of words ?

- We can use **symbolic representations**
  - Bag of words: no word order, large, sparse
  - N-grams: large, even sparser

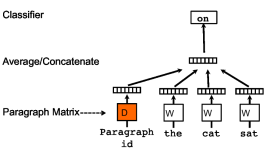
- We can use **dense word representations**
  - Infinitely many sentences: learn word vectors and learn composition
  - Principle of **compositionality**: derive meaning from word meaning and combination rules
  - Not so easy: *figurative* language, implicitness, sarcasm..
  - Basic model: simple, communative operation (*mean*)
    $\rightarrow$ Word order not taken into account

How to do better ?
$\rightarrow$ Deep learning architecture for composing words together !
$\rightarrow$ Or **directly learn sentence representations**

Applications to sentence representations

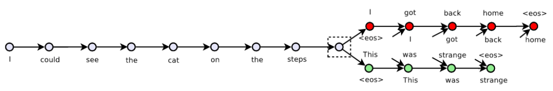
- A first, direct extension of CBOW: Paragraph2Vec

Distributed Representations of Sentences and Documents (Le and Mikolov, 2014)



- Skip-thought: learn to predict the **previous** and **following** sentences

Skip-Thought Vectors (Kiros et al, 2015)



Applications to sentence representations

- **Contrastive learning** is often used for learning sentence representations

An efficient framework for learning sentence representations (Logeswaran and Lee, 2018)

Main idea: **learn to recognize the next sentence**

- Use two encoders f and g (RNNs)
- Input s, classify between the right candidate s_c and samples s' from $\mathcal{S}_c$
- Use MLE objective on the sofmax, corresponding to a contrastive loss with (s, s_c) as positive pair and (s, s') as negatives