Neural Language Models, Word Embeddings and Deep learning for NLP

Matthieu Labeau - Associate Professor, Telecom Paris, IPP

DS-Telecom-20: Natural Language Processing - 11/03/2024

Outline

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

Neural Probabilistic Language Models

Reminder: Difficulties with counting

- 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*.

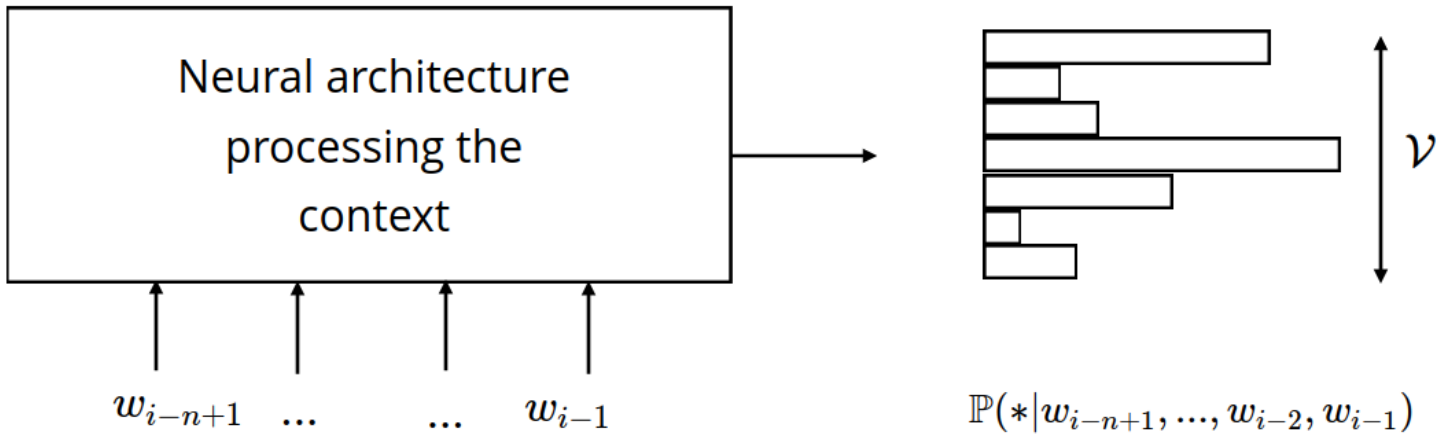
$$\text{word} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \in \mathbb{N}^{|\mathcal{V}|} \qquad \text{word} = \begin{bmatrix} 0.212 \\ 0.792 \\ -0.177 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{bmatrix} \in \mathbb{R}^{d}$$

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

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

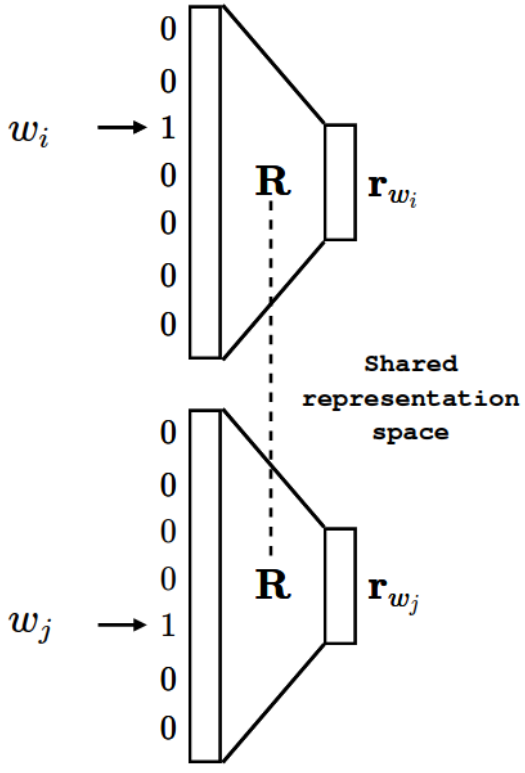
  in the training data will increase the probability of all 'neighbor' sentences. Having also the following sentence:

  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.

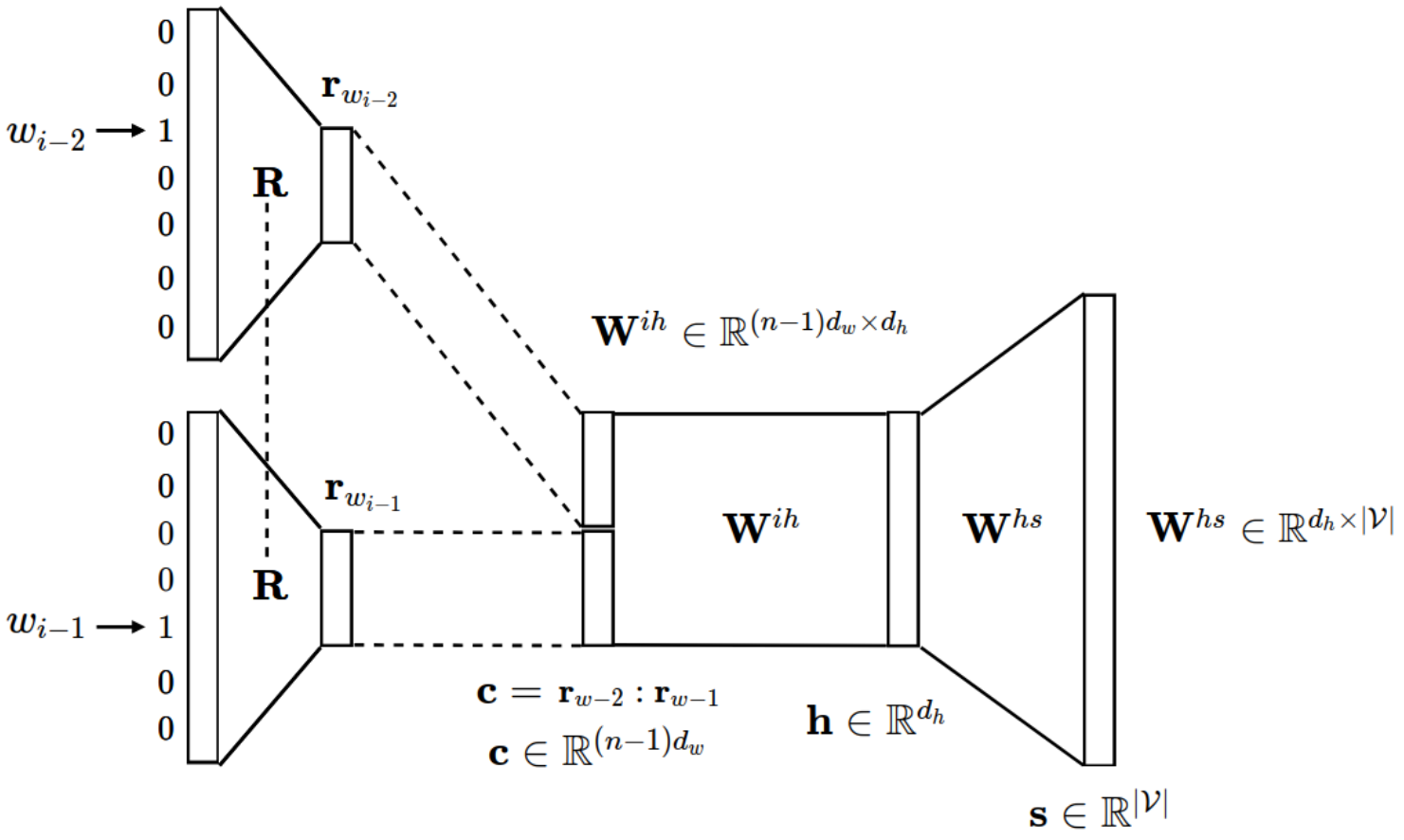


Neural model: Obtaining scores

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

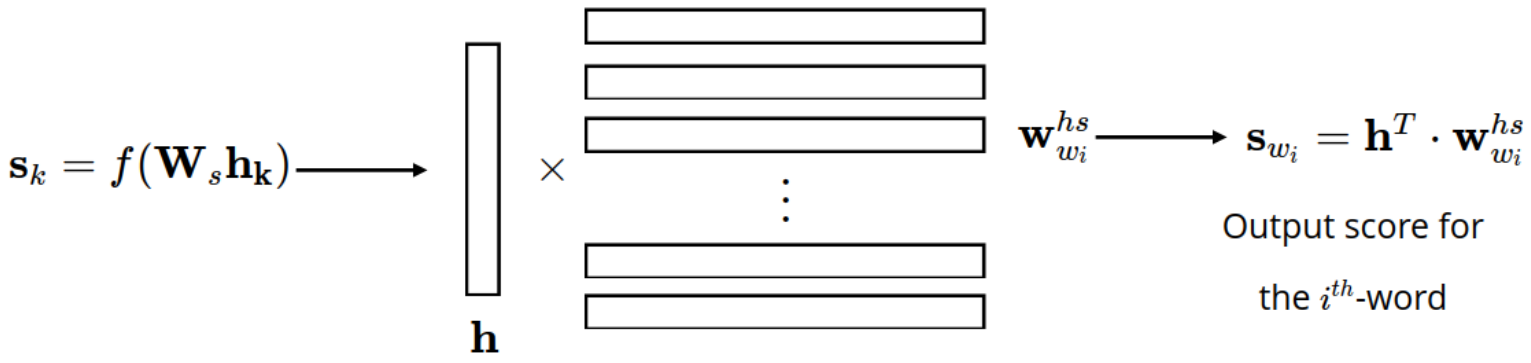
$$\mathbf{h} = \phi(\mathbf{W}^{ih}\mathbf{c})$$

- 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 $[\mathbf{w}_k^{hs}]_{k=1}^{|\mathcal{V}|}$:

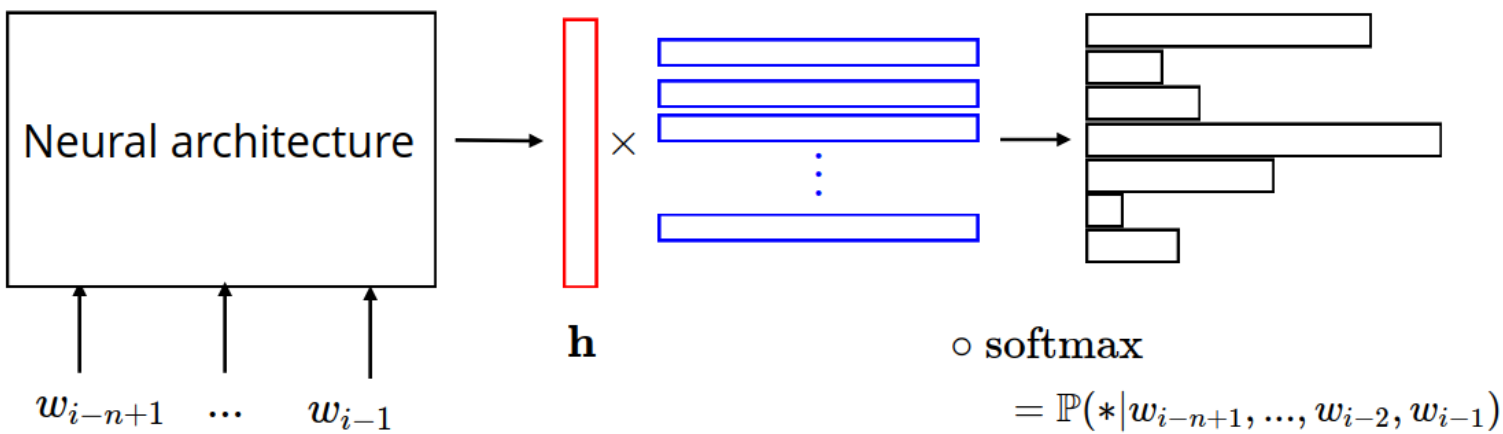


Neural model: Computing probabilities

- 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:

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

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:

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

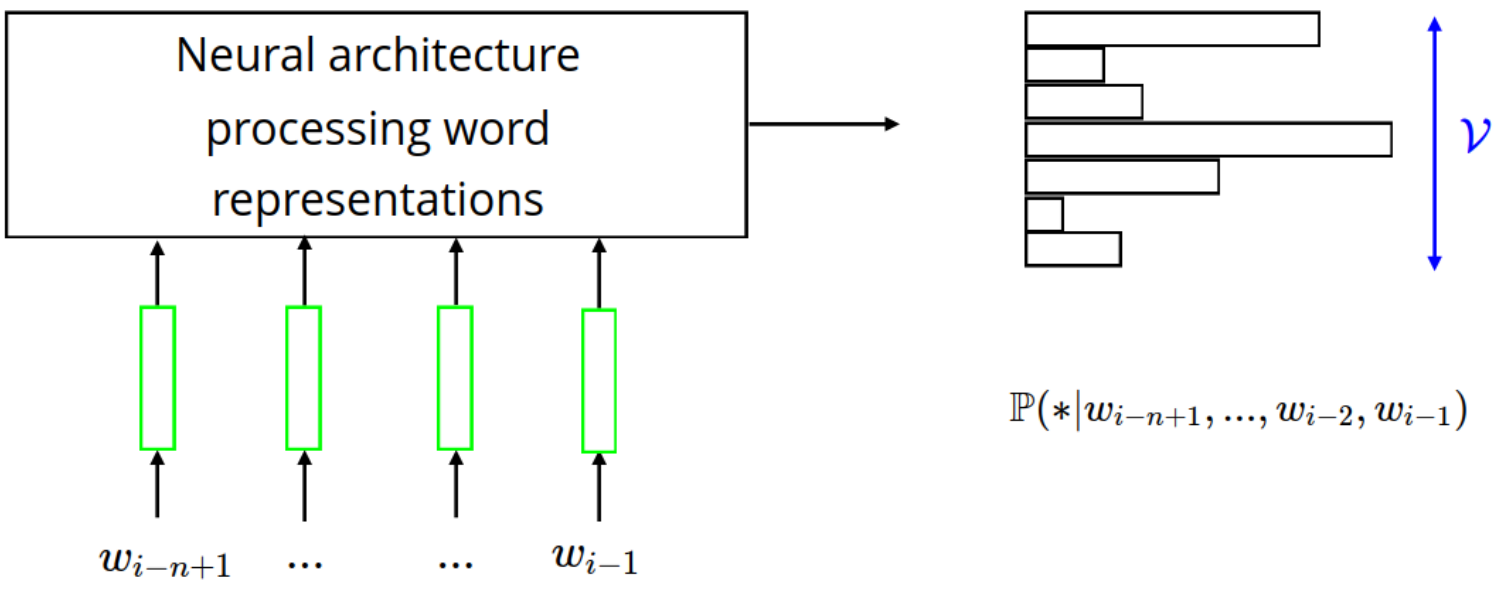
- Remark: minimizing the cross-entropy is equivalent to minimizing the Kullback-Leibler divergence
- We can note again that the perplexity is a simple function of the cross-entropy:

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

$\rightarrow$ We are directly minimizing perplexity when training

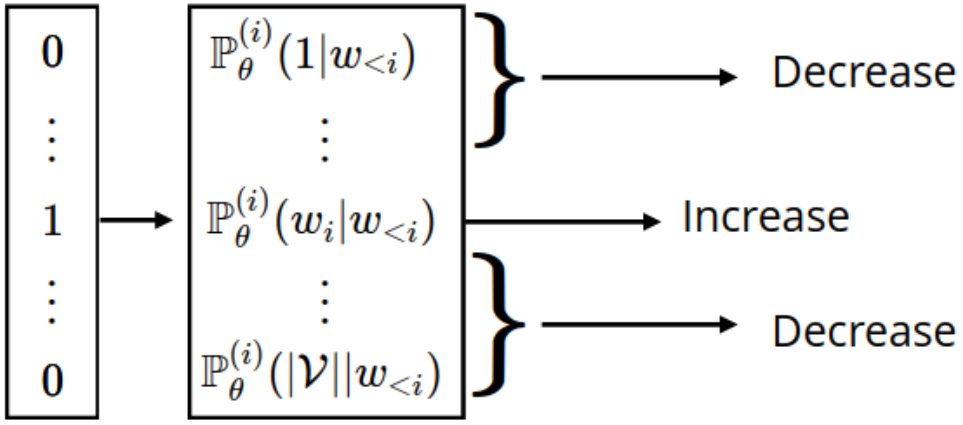
Neural model: Assessment

- 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:

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

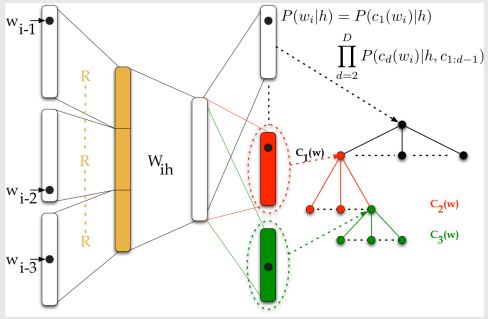
- The first term increases the conditional log-likelihood of w_i given $w_{<i}$
- The second decreases the conditional log-likelihood of all the other words $w \in \mathcal{V}$ - and implies a double summation on $\mathcal{V}$

$\rightarrow$ The softmax causes this computational bottleneck !

Parenthesis: dealing with the bottleneck ?

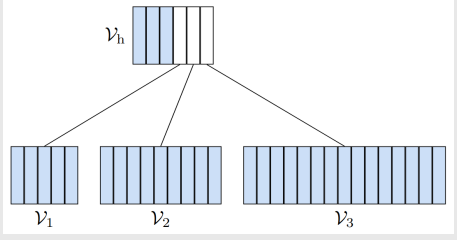
- Making hierarchical predictions: replace complexity in $O(|\mathcal{V}|)$ by $O(\log |\mathcal{V}|)$

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)

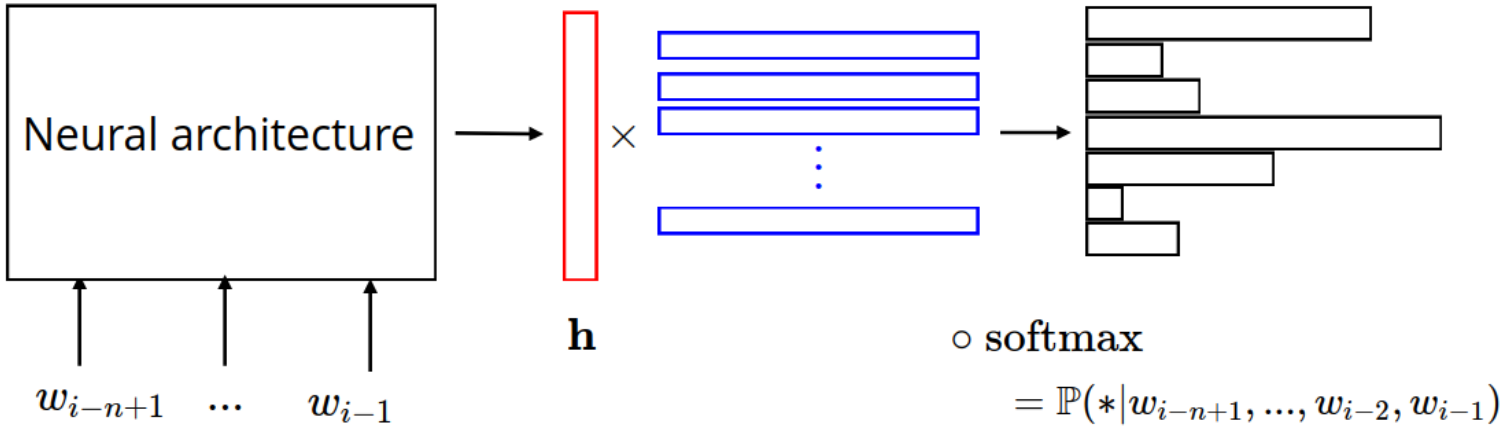
  - From text-document matrices (*TF-IDF, LSA..*)
  - From word-context matrices (*Co-occurences, PPMI..*)
  - From more complex relationnal patterns, that can handle word order

- 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 ?

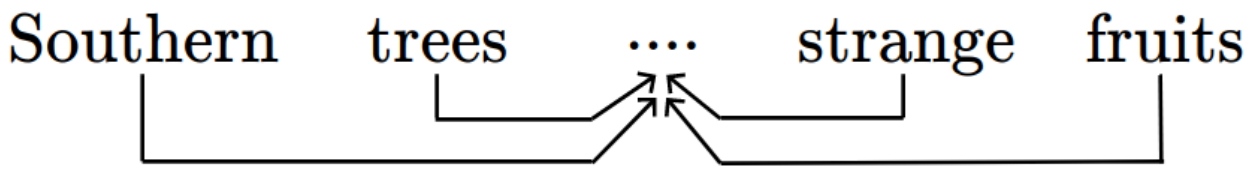


- We can work on two things: the **architecture** and the **softmax computation**
  - Hypothesis: the distribution of the context is what matters $\rightarrow$ **simplify the architecture** !
  - Goal: learning representation $\rightarrow$ **no need for a proper softmax** !

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:

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

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

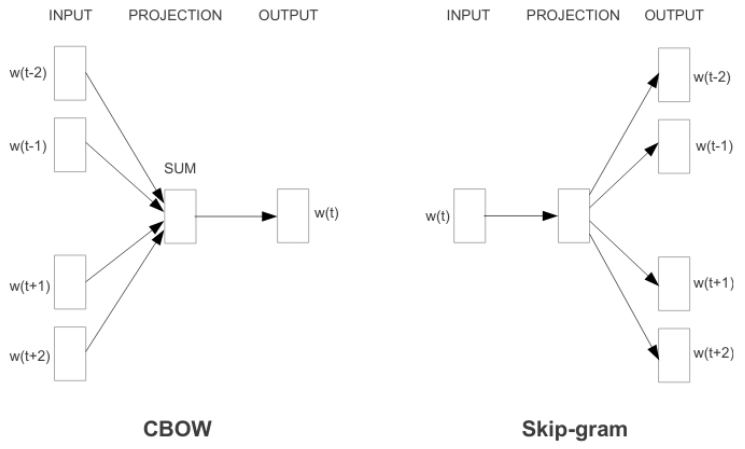
Learning Word Embedding: objective functions

- 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**:

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



(*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 ?

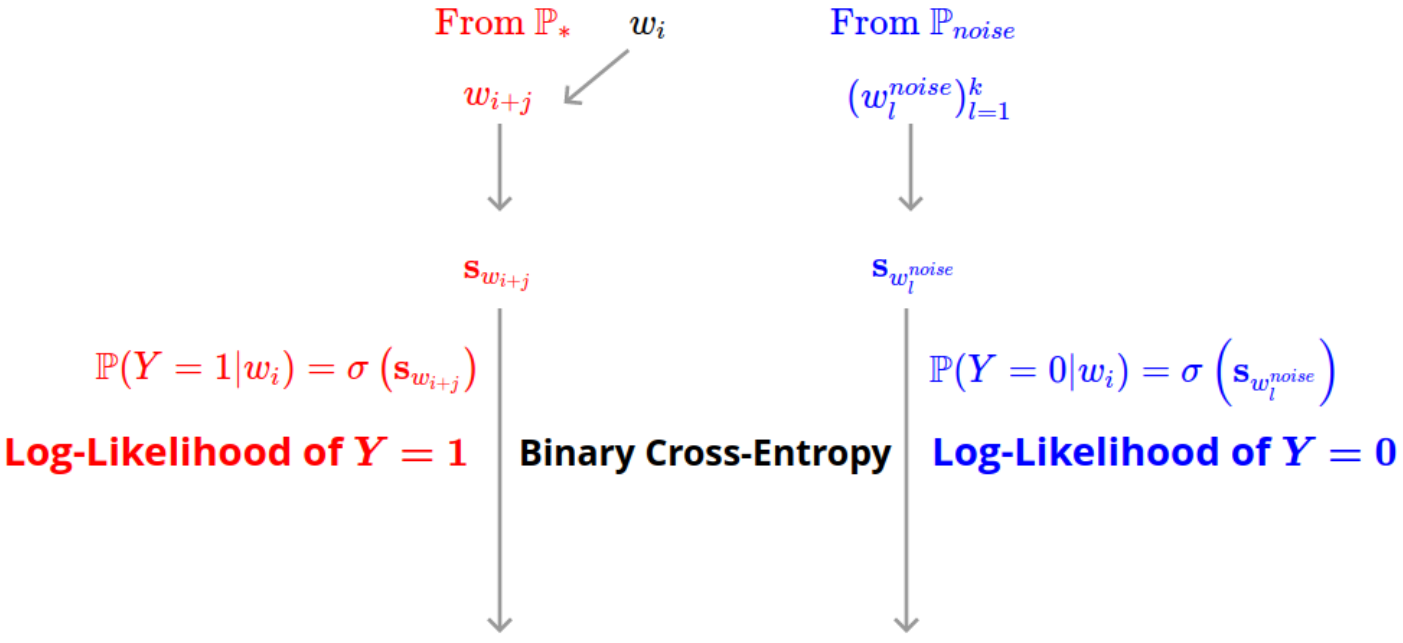
$$\mathbb{P}_\theta(w_{i+j}|w_i) = \sigma(\mathbf{s}_{w_{i+j}})$$

- 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



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

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

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

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

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

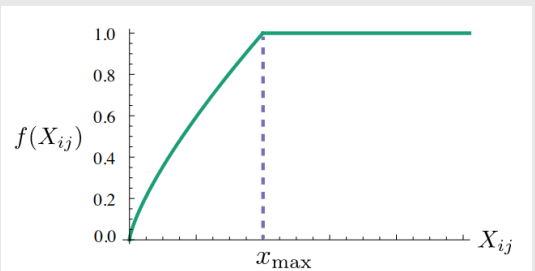
Central idea: **directly learn word embeddings by predicting word co-occurence counts** !

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

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

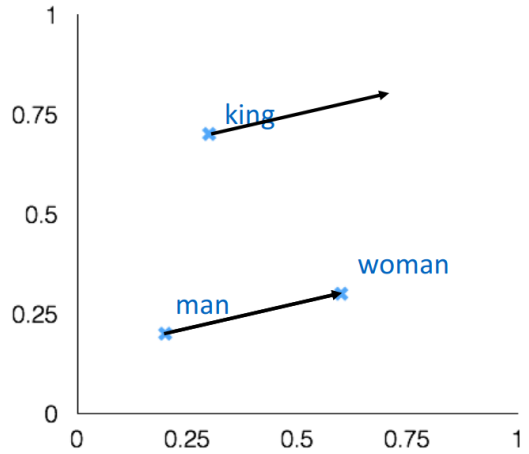


- Very fast training
- Scales to very large corpora

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

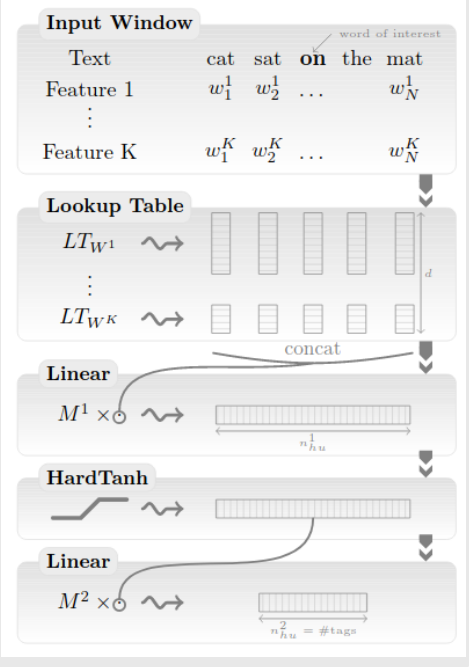
NLP from scratch

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

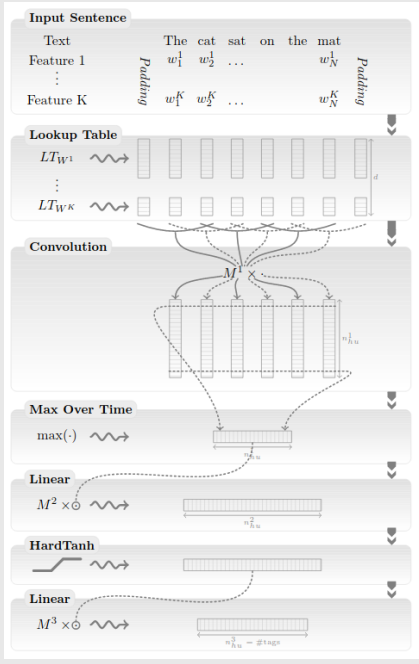
- *NLP from scratch*: first occurence of the idea
- Word embeddings obtained with word2vec and GloVe: hugely popular, improvements on many (many) tasks

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 our data !**

Neural architectures for sequence processing

Representing a sequence of words

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

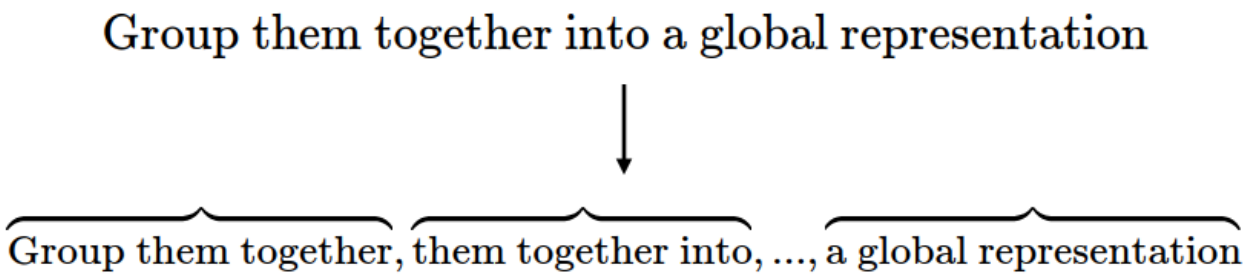
- We can use **dense word representations**
  - Infinitely many sentences: learn word vectors and learn composition
  - Principle of **compositionality**: derive meaning from word meaning and combination rules
  - Not so easy: *figurative* language, implicitness, sarcasm..
  - Basic model: simple, communative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

How to do better ?
$\rightarrow$ Convolutional and Recurrent neural networks

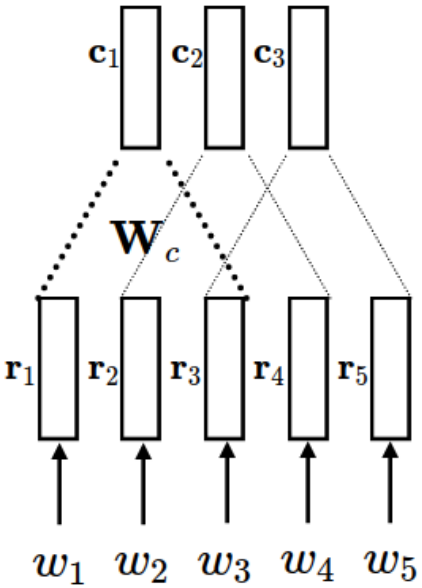
Convolutional approach in general

- Classification tasks on sequences necessitate a **global** representation
- Idea: consecutively agglomerate features obtained on windows of n-grams
  - Compute representations for each window (= possible subsequence of a certain length)$\rightarrow$ **convolution**
  - Group them together into a global representation $\rightarrow$ **pooling**

- With text, the convolution is one-dimensional



Convolutional networks for NLP



- Inputs in the same window are concatenated:

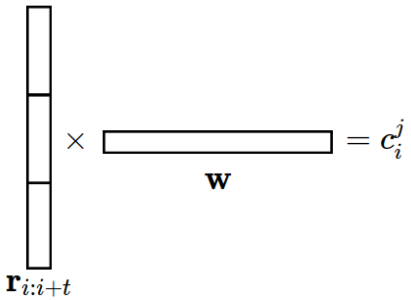
$$\mathbf{r}_{i:i+t} = [\mathbf{r}_i : .. : \mathbf{r}_{i+t}]$$

- A filter is a vector $\mathbf{w} \in \mathbb{R}^{td_w}$ applied to the window to obtain an individual feature:

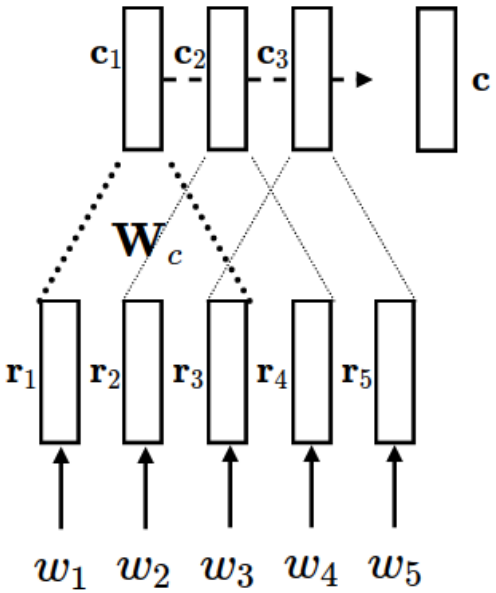
$$c_i^j = \phi(\mathbf{w}_j^T \mathbf{r}_{i:i+t})$$

- The weight matrix $\mathbf{W}_c$ re-groups the d_c filters, outputting a feature vector of size d_c:

$$\mathbf{c}_i = \phi(\mathbf{W}_c^T \mathbf{r}_{i:i+t})$$



Convolutional networks for NLP



- With max-pooling, capture the most important activation over time:

$$\mathbf{c} = max(\{\mathbf{c}_i\}_{i=1}^{n-t})$$

$$max\left(\begin{bmatrix}\vdots\\ \mathbf{c}_1^j\\ \vdots\end{bmatrix}, \begin{bmatrix}\vdots\\ \mathbf{c}_2^j\\ \vdots\end{bmatrix}, \begin{bmatrix}\vdots\\ \mathbf{c}_3^j\\ \vdots\end{bmatrix}\right) \rightarrow \begin{bmatrix}\\ \\ \end{bmatrix}$$

- Pooling can be used with other functions
- With overlapping (like here) or non-overlapping windows ($\rightarrow$ stride)

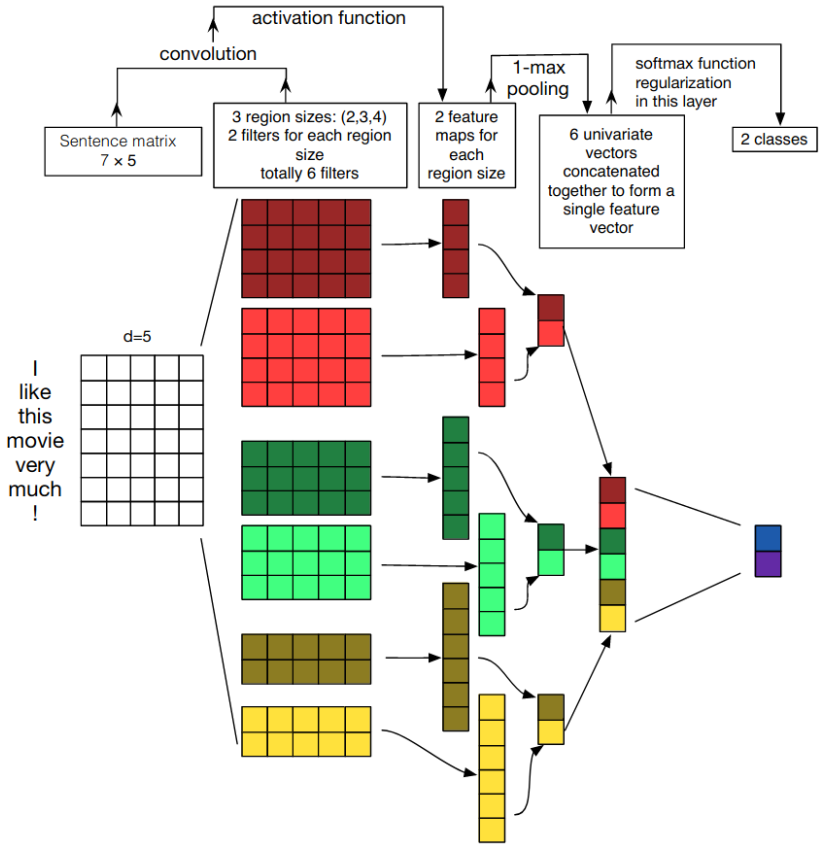
CNN: Applications in NLP

- Convolutional networks: lighter, faster - filters of various sizes

- Combine filters of different sizes to obtain a fixed-sized representation for the sentence + Classification

Convolutional Neural Networks for Sentence Classification (Kim, 2014)

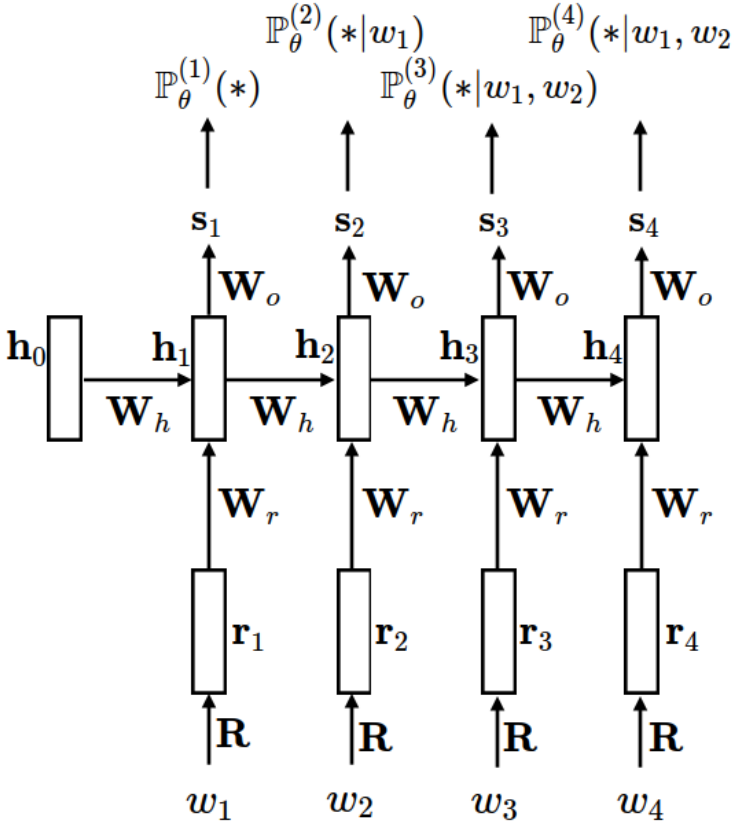
Very Popular for text sequence representations for classification $\longrightarrow$



Reccurent approach in general

With language models, we have the **long distance dependencies** issue: We need an architecture that can process inputs of any lengths.

$\rightarrow$ **Recurrent neural network!**



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

$$\mathbf{h}_i = \phi(\mathbf{W}_r \mathbf{r}_{w_i} + \mathbf{W}_h \mathbf{h}_{i-1})$$

$$\mathbf{s}_i = f(\mathbf{W}_o \mathbf{h_i})$$

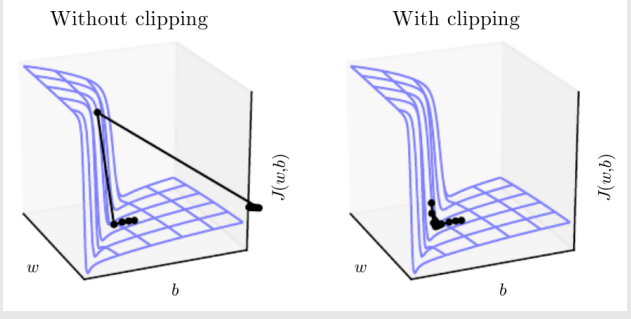
Training RNN Language Models

- Same objective: minimizing the negative log-likelihood on $\mathcal{D} = (w_i)_{i=1}^N$
- However, this time, updates go back to previous examples via reccurent links: we use *Backpropagation Through Time*

Main issues:

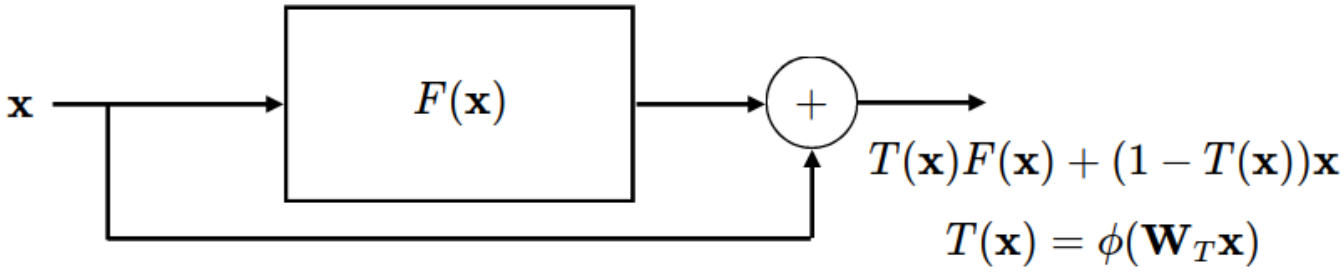
- **Vanishing/exploding** gradient:
  - Architectures dedicated to this issue: LSTM, GRU
  - Gradient clipping to $\gamma \longrightarrow$
- Long term memory not actually long term

From the *Deep Learning Book (Goodfellow et al, 2016)*



Innovations

- **Gating/Skipping**, similar to LSTM/GRU, but vertically:



  → Very useful for deep networks

- **Batch Normalization**: Scaling the output of the activation function to have zero mean and unit variance

Accelerating deep network training by reducing internal covariate shift (Ioffe et al, 2015)

- Make models less sensible to parameter initialization
- Reduce the difficulty with learning rate tuning

RNN-LMs: Assessment

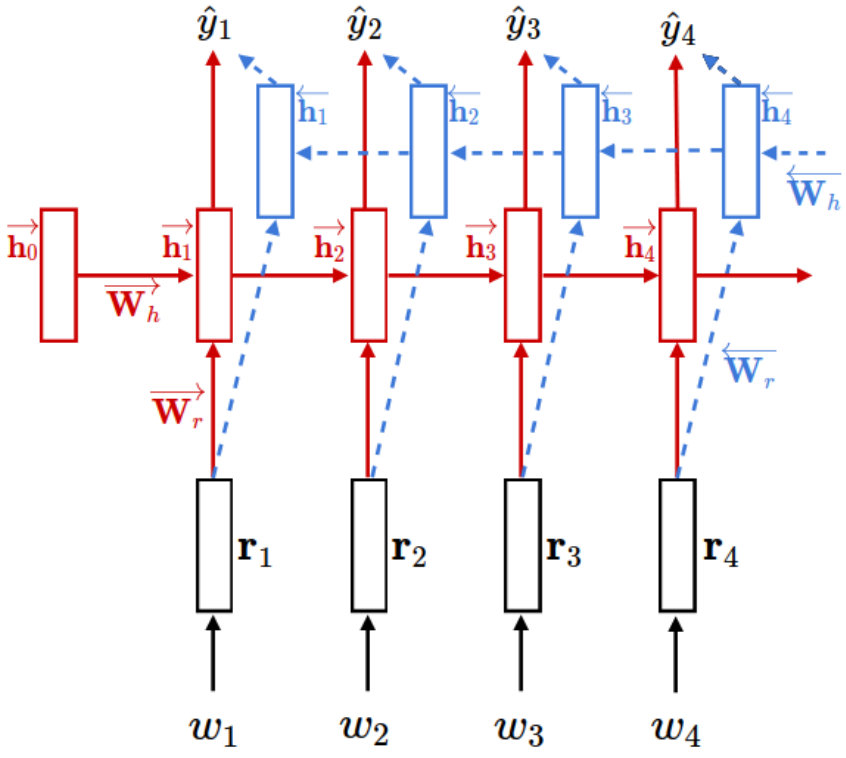
- **Results:** RNNs greatly improved results upon previous baselines

- **Many different architecures**, many different improvements, tweaks, and optimization procedures:
  - Difficult to keep track of and compare !
  - *As far as I know*, the state-of-the-art for LSTM-based is the ASGD (Average SGD) Weight Dropped (Dropout on Recurrent weights) LSTM, or AWD-LSTM

Regularizing and Optimizing LSTM Language Models (Merity et al, 2017)

Model	PTB Validation	PTB Test	WT2 Validation	WT2 Test
AWD-LSTM (tied)	60.0	57.3	68.6	65.8
– fine-tuning	60.7	58.8	69.1	66.0
– NT-ASGD	66.3	63.7	73.3	69.7
– variable sequence lengths	61.3	58.9	69.3	66.2
– embedding dropout	65.1	62.7	71.1	68.1
– weight decay	63.7	61.0	71.9	68.7
– AR/TAR	62.7	60.3	73.2	70.1
– full sized embedding	68.0	65.6	73.7	70.7
– weight-dropping	71.1	68.9	78.4	74.9

RNN as encoders - BiRNN



$$\overrightarrow{\mathbf{h}_i} = \phi(\overrightarrow{\mathbf{W}}_r \mathbf{r}_{w_i} + \overrightarrow{\mathbf{W}}_h \overrightarrow{\mathbf{h}_{i-1}})$$

$$\overleftarrow{\mathbf{h}_i} = \phi(\overleftarrow{\mathbf{W}}_r \mathbf{r}_{w_i} + \overleftarrow{\mathbf{W}}_h \overleftarrow{\mathbf{h}_{i+1}})$$

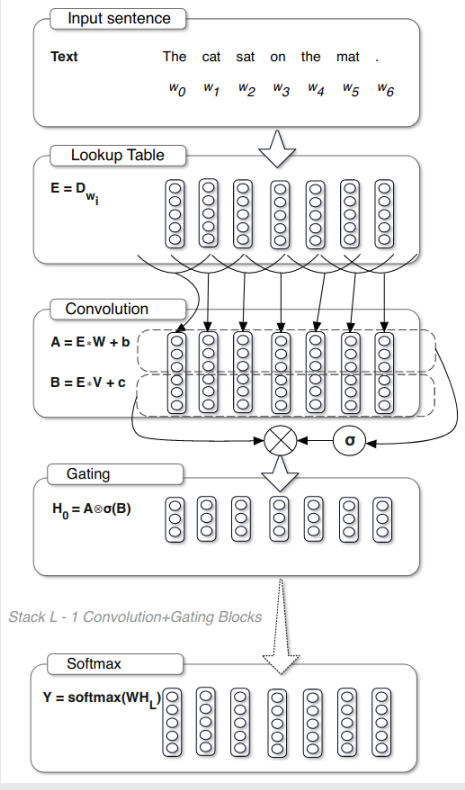
$$\hat{y}_i = f(\mathbf{W}_o[\overrightarrow{\mathbf{h}}_i : \overleftarrow{\mathbf{h}}_i])$$

- $[\overrightarrow{\mathbf{h}}_i : \overleftarrow{\mathbf{h}}_i]$ represents w_i using both contexts !
- Powerful, but you need the entire input sequence
  $\rightarrow$ **Encoder** only, it can not be used for language modeling

RNN vs CNN: Assessment

- Advantages: Can process sequences of any length, constant number of parameters
- Disadvantages: **slow**, tough to memorize information
- Compared to CNNs: **very fast**, but not adapted to LMs $\rightarrow$
  - Hard to train, need residual connections
  - Less information on the position of words.
  - Not practical for left-to-right prediction

Language Modeling with Gated Convolutional Networks (Dauphin et al, 2017)

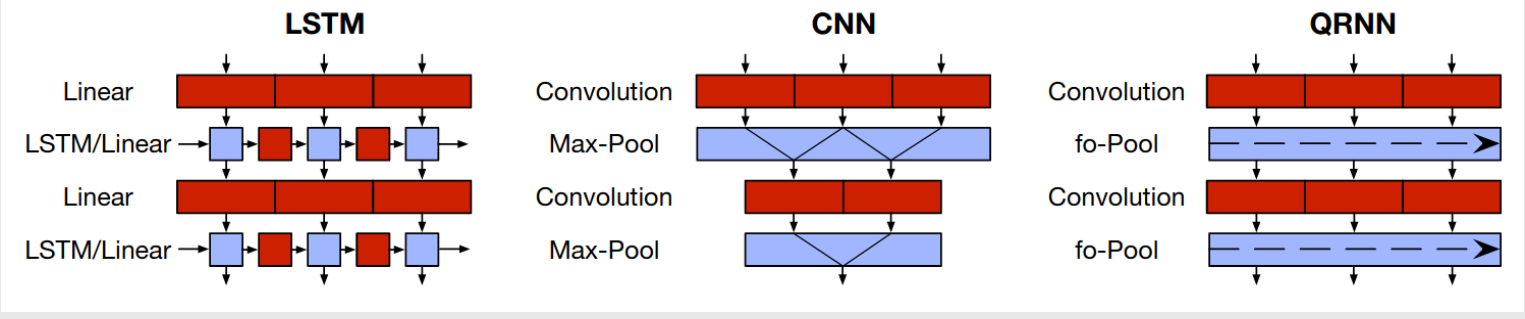


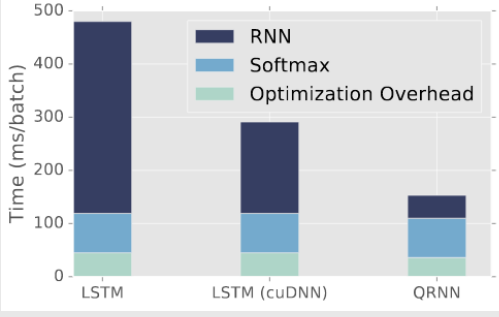
The best of both worlds ?

Performance of RNNs, being as fast as CNNs ?

Quasi-Recurrent Neural Networks (Bradbury et al, 2017)

Use convolutions and 'recurrent' pooling to minimize the recurrent part of the architecture and accelerate computation





		Sequence length				
		32	64	128	256	512
Batch size	8	5.5x	8.8x	11.0x	12.4x	16.9x
	16	5.5x	6.7x	7.8x	8.3x	10.8x
	32	4.2x	4.5x	4.9x	4.9x	6.4x
	64	3.0x	3.0x	3.0x	3.0x	3.7x
	128	2.1x	1.9x	2.0x	2.0x	2.4x
	256	1.4x	1.4x	1.3x	1.3x	1.3x

The specific case of Statistical Machine Translation

- Research on Machine Translation (**MT**) began early (1950s)
  - System were mainly **rule-based**

- Before neural approaches, MT was essentially Statistical MT
  - Data-driven: based on **parallel corpora**
  - Working at the sentence-level, and requiring **word alignement**
  - Mainly focused on **high-resource** languages

- Assuming a sentence in a language f that we want to translate as a sentence in a target language e. We are looking for:

$$\text{argmax}_e \mathbb{P}(e|f) = \text{argmax}_e \underbrace{\mathbb{P}(f|e)}\underbrace{\mathbb{P}(e)}$$

  where $\mathbb{P}(f|e)$ is the *translation model* and $\mathbb{P}(e)$ the *language model*

- Evaluation with **BLEU** score (measures n-gram overlap)

SMT: Many components

- The main translation models were phrase-based:
  - Segment the parallel sentences in phrases
  - Use alignement probabilities learnt with a separate model
  - Generating a translation (decoding) implies:
    - Using translation probabilities to obtain target sentences
    - Using a reordering model along the language model

$\rightarrow$ Each of these (and others) steps implies many complex design choices, and

- Feature engineering,
- Linguistic resources,
- Systems to maintain and update... for every different language pair

$\rightarrow$ Many other difficulties: using context, long-range dependancies, ...

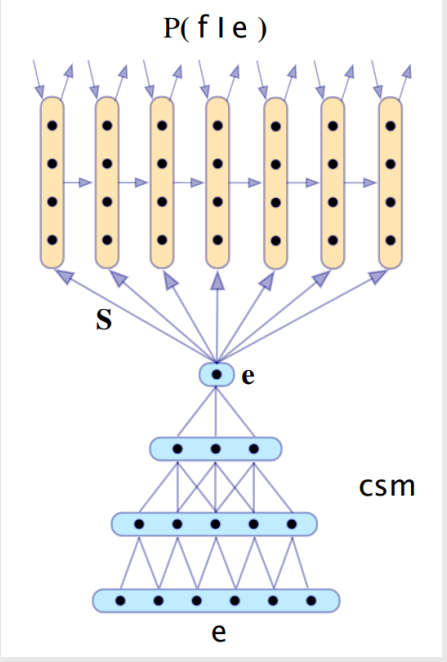
Towards sequence-to-sequence models

Our goal: an **end-to-end** model allowing to directly translate a text sequence into another language

Recurrent Continuous Translation Models (Kalchbrenner et al, 2013)

One of the first neural model for **end-to-end machine translation:**

- The input sentence is encoded into one global representation via convolution
- The representation is decoded into the target sentence with a RNN

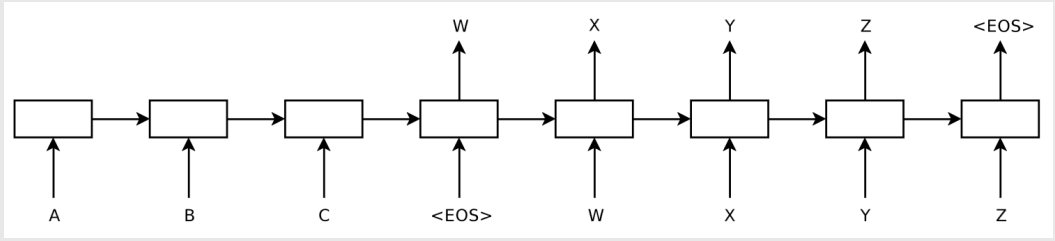


Sequence-to-sequence models

Sequence-to-sequence (Seq2seq) models are based on 2 RNNs:

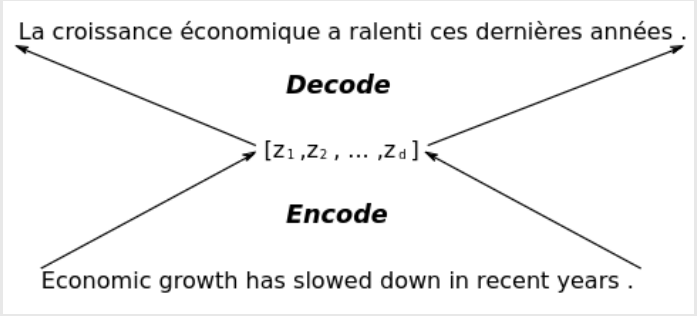
- An **encoder**, outputting a sentence representation
- A **decoder**, generating a word at each step

Sequence to Sequence Learning with Neural Networks, (Sutskever et al, 2014)



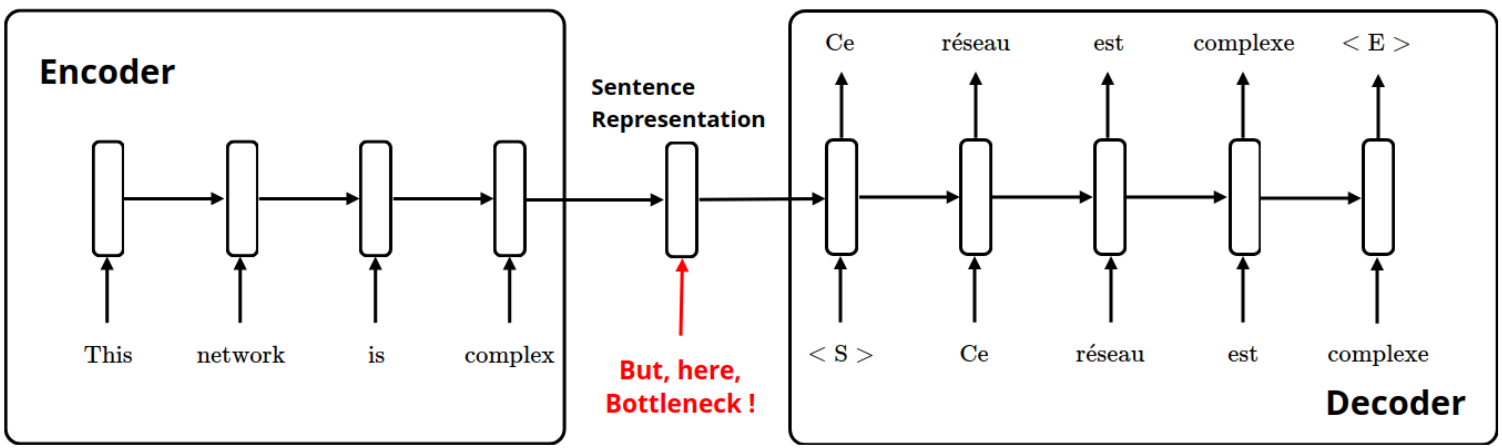
Motivation: End-to-end neural machine translation !

On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, (Cho et al, 2014)



Seq2Seq models: conditional LMs

- Can be seen as a conditional language model: we optimize the likelihood of each word given its previous context **plus the input of the encoder**
- Sequence-to-sequence is very versatile:
  - Summarization (long text $\rightarrow$ short text)
  - Dialogue (previous utterances $\rightarrow$ next utterance)
  - Parsing (input text $\rightarrow$ output parse as sequence)
  - Code generation (natural language $\rightarrow$ Python code)
- **Limitation**: one vector to represent the full encoded sentence



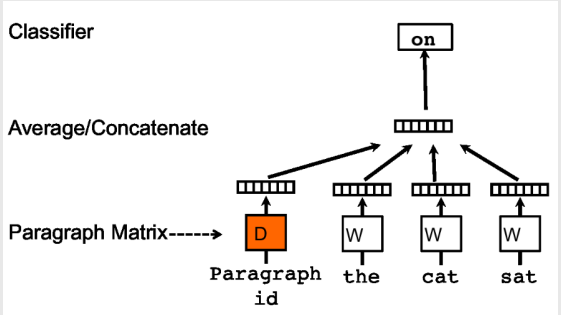
Seq2Seq models for NMT: Assesment

- NMT with Seq2seq provided huge gains in performance (score BLEU) compared to SMT
- Translation is more fluent, better use of the context
- Models are far easier to train and maintain and the same method is used for different language pairs
- However (with neural networks) NMT is less interpretable and difficult to control.
- Besides, some problems are still there:
  - The large ($\rightarrow$ slow training) and closed output vocabulary
  - It does not work well on low-resource language pairs
  - And **long-term dependencies**

Applications to sentence representations

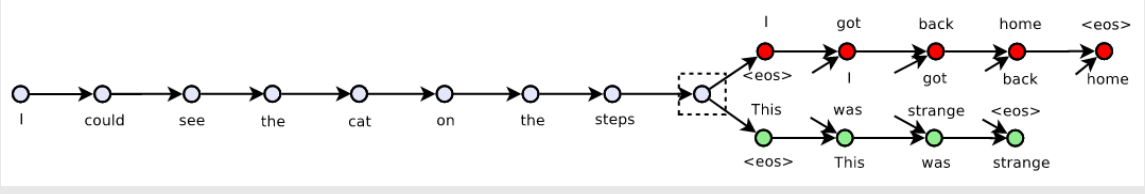
- A first, direct extension of CBOW: Paragraph2Vec

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



- Skip-thought: learn to predict the **previous** and **following** sentences with a seq2seq model

Skip-Thought Vectors (Kiros et al, 2015)



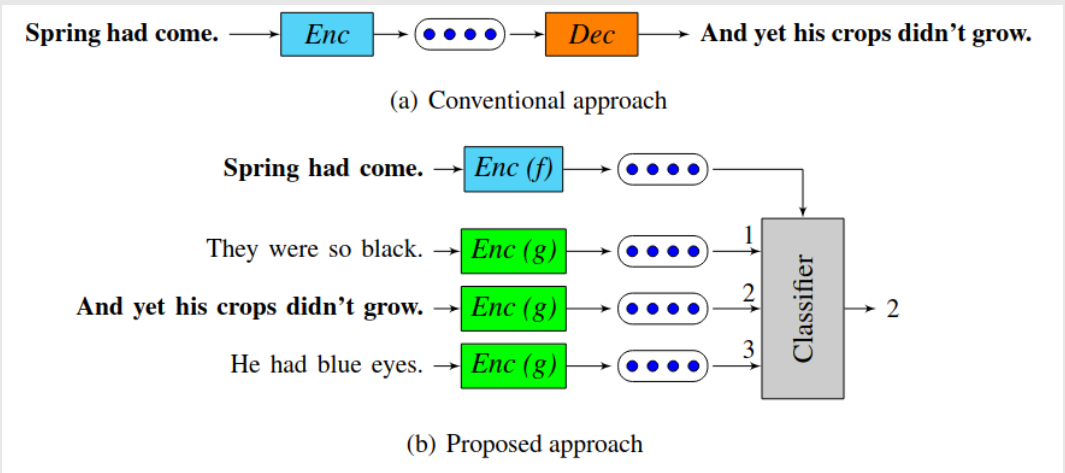
Applications to sentence representations

- **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



Summary

- Possible architectures depending on the *input-output configuration*:
  - Many-to-one: Sequence of words $w_{1:T}$ to single label y
    $\rightarrow$ **Encoder** architectures
  - Many-to-many: Sequence of words $w_{1:T}$ to sequence of labels $y_{1:T}$
  - One-to-many: Single words w to sequence $y_{1:T}$
    $\rightarrow$ **Decoder** architectures
  - Many-to-many: Sequence of words $x_{1:T_x}$ to sequence $y_{1:T_y}$
    $\rightarrow$ **Encoder-decoder** architectures

- Until now, the decoder is necessarily **recurrent**

- CNNs are not versatile; RNNs are slow; both fail at long term dependencies
  - Next: solving all of these with the **attention mechanism !**