Lecture 6: Language modeling

LTAT.01.001 – Natural Language Processing

Kairit Sirts (kairit.sirts@ut.ee)

22.03.2022

Plan for today

- N-gram language modeling
- Smoothing
- Feedforward neural language model
- Evaluating language models – perplexity

N-gram language modeling

The task of language modeling

The cat sat on the mat

The mat sat on the cat

The cat mat the on sat

The task of language modeling

Two main purposes:

- Estimate the grammaticality or fluency of a sentence or text in order to pick one of several possible options
- Generate fluent and grammatical sentence or text

Which sentence is better?

The more probable sentence is better

P(The cat sat on the mat) > P(The mat sat on the cat)

P(The mat sat on the cat) > P(The cat mat the on sat)

How to compute the sentence probability?

P(The cat sat on the mat) $= \frac{\#(\text{The cat sat on the mat})}{\#\ all\ sentences} = ?$

P(The mat sat on the cat) $= \frac{\#(\text{The mat sat on the cat})}{\#\ all\ sentences} = ?$

P(The cat mat the on sat) $= \frac{\#(\text{The cat mat the on sat})}{\#\ all\ sentences} = ?$

- the number or count of such sentences

That's clearly not doable in general!

How to compute the sentence probability?

Word: w Sentence: $\boldsymbol{w} = w_1 w_2 \dots w_n$

Factorize the joint probability:

- In general:

$$p(x, y, z) = p(x)p(y|x)p(z|x, y)$$

- Similarly:

$$p(w_1, w_2, \dots, w_n) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2) \dots p(w_n|w_1, w_2, \dots, w_{n-1})$$

- **It still does not solve the problem!**

How to compute the sentence probability?

- Cannot estimate directly:

$$p(w_1 w_2 \dots w_n) = \frac{\#(w_1 w_2 \dots w_n)}{\#\, all\ sentences}$$

- Cannot use the factorization:

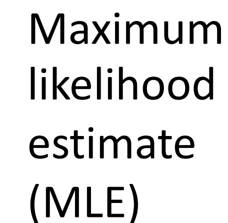
$$p(w_1 w_2 \dots w_n) = \prod_{i=1}^{n} p(w_i | w_1 \dots w_{i-1})$$

How to compute the sentence probability?

But word probabilities are doable:

- Take a large text corpus (millions/billions of words)
- Compute the probability for each word type (unique word)

$$p(w) = \frac{\#(w)}{\#\ all\ words\ in\ the\ corpus}$$



How to compute the sentence probability?

- What if we treat each word as independent of other words? Then:

$$p(\boldsymbol{w}) \cong p(w_1) \times p(w_2) \times \cdots \times p(w_n)$$

P(The cat sat on the mat) = P(The mat sat on the cat)

P(The mat sat on the cat) = P(The cat mat the on sat)

How to compute the sentence probability?

- Maybe add some context?

$$p(\mathbf{w}) \cong p(w_1) \times p(w_2|w_1) \times p(w_3|w_2) \times \cdots \times p(w_n|w_{n-1})$$

P(The cat sat on the mat)

$$= p(The)p(cat|The)p(sat|cat)p(on|sat)p(the|on)p(mat|the)$$

P(The mat sat on the cat)

$$= p(The)p(mat|The)p(sat|mat)p(on|sat)p(the|on)p(cat|the)$$

P(The cat mat the on sat)

$$= p(The)p(cat|The)p(mat|cat)p(the|mat)p(on|the)p(sat|on)$$

Sentence probability – Markov property

Independence assumption or **Markov assumption** (in the context of language modeling):

- **The next word only depends on the (few) last word(s).**

- This is precisely the model we had on the previous slide and it is called **bigram language model** because we are looking at the word bigrams.

N-gram language model

In general, we talk about **n-gram language models**, where the next word depends on a fixed history of **n-1** words.

- Unigram model – all words are independent
- Bigram model – the next word depends on the last word only
- Trigram model – the next word depends on two last words
- 4-gram model – the next word depends on the three last words
- 5-gram model – the next word depends on the four last words
- etc

Computing n-gram probabilities

- Unigrams: w_i $$p(w_i) = \frac{\#w_i}{\#\, all\ words}$$
- Bigrams: $w_{i-1}w_i$ $$p(w_i|w_{i-1}) = \frac{\#(w_{i-1}, w_i)}{\#(w_{i-1})}$$
- Trigrams: $w_{i-2}w_{i-1}w_i$ $$p(w_i|w_{i-2}, w_{i-1}) = \frac{\#(w_{i-2}, w_{i-1}, w_i)}{\#(w_{i-2}, w_{i-1})}$$

Sentence probability according to the n-gram model

- If

$$p(w_i|w_1, w_2, \dots, w_{i-1}) \cong p(w_i|w_{i-k}, \dots, w_{i-1})$$

- Where $k = ngram\, rank\ - 1$:
    - Unigrams: $k = 0$
    - Bigrams: $k = 1$
    - Trigrams: $k = 2$, etc
- Then

$$p(\boldsymbol{w}) = \prod_{i=1}^{n} p(w_i|w_1, w_2, \dots, w_{i-1}) \cong \prod_{i=1}^{n} p(w_i|w_{i-k}, \dots, w_{i-1})$$

Bigram language model: example

An example corpus:

1. the cat saw the mouse
2. the cat heard a mouse
3. the mouse heard
4. a mouse saw
5. a cat saw

Bigram	Count	Unigram	Count	Bigram prob
START the		START		
the cat		the		
cat saw		cat		
saw the		saw		
the mouse		the		
mouse END		mouse		
cat heard		cat		
heard a		heard		
a mouse		a		
START a		START		
mouse saw		mouse		
saw END		saw		
a cat		a		

Bigram language model: example

An example corpus:

1. the cat saw the mouse
2. the cat heard a mouse
3. the mouse heard
4. a mouse saw
5. a cat saw

Bigram	Count	Unigram	Count	Bigram prob
START the	3	START	5	0.6
the cat	2	the	4	0.5
cat saw	2	cat	3	0.67
saw the	1	saw	3	0.33
the mouse	2	the	4	0.5
mouse END	2	mouse	4	0.5
cat heard	1	cat	3	0.33
heard a	1	heard	2	0.5
a mouse	2	a	3	0.67
START a	2	START	5	0.4
mouse saw	1	mouse	4	0.25
saw END	2	saw	3	0.67
a cat	1	a	3	0.33

Bigram language model: example

P(the cat heard) = ?

P(the mouse saw the cat) = ?

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Bigram language model: example

P(the cat heard) =

= P(the|START) x P(cat|the) x P(heard|cat)

x P(END|heard)

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Bigram language model: example

$P(\text{the cat heard}) =$

$= P(\text{the}|\text{START}) \times P(\text{cat}|\text{the}) \times P(\text{heard}|\text{cat})$

$\times P(\text{END}|\text{heard})$

$= 0.6*0.5*0.33*0.5 = 0.0495$

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Bigram language model: example

P(the mouse saw the cat) = ?

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Bigram language model: example

P(the mouse saw the cat) =

=P(the|START) x P(mouse|the) x P(saw|mouse)

x P(the|saw) x P(cat|the) x P(END|cat)

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Bigram language model: example

P(the mouse saw the cat) =

=P(the|START) x P(mouse|the) x P(saw|mouse)

x P(the|saw) x P(cat|the) x P(END|cat)

= 0.6*0.5*0.25*0.33*0.5***0 = 0**

Bigram	Bigram prob
The|START	0.6
cat|the	0.5
saw|cat	0.67
the|saw	0.33
mouse|the	0.5
END|mouse	0.5
heard|cat	0.33
a|heard	0.5
mouse|a	0.67
a|START	0.4
saw|mouse	0.25
END|saw	0.67
cat|a	0.33
END|heard	0.5

Smoothing

Sparsity issues

Natural languages are sparse!

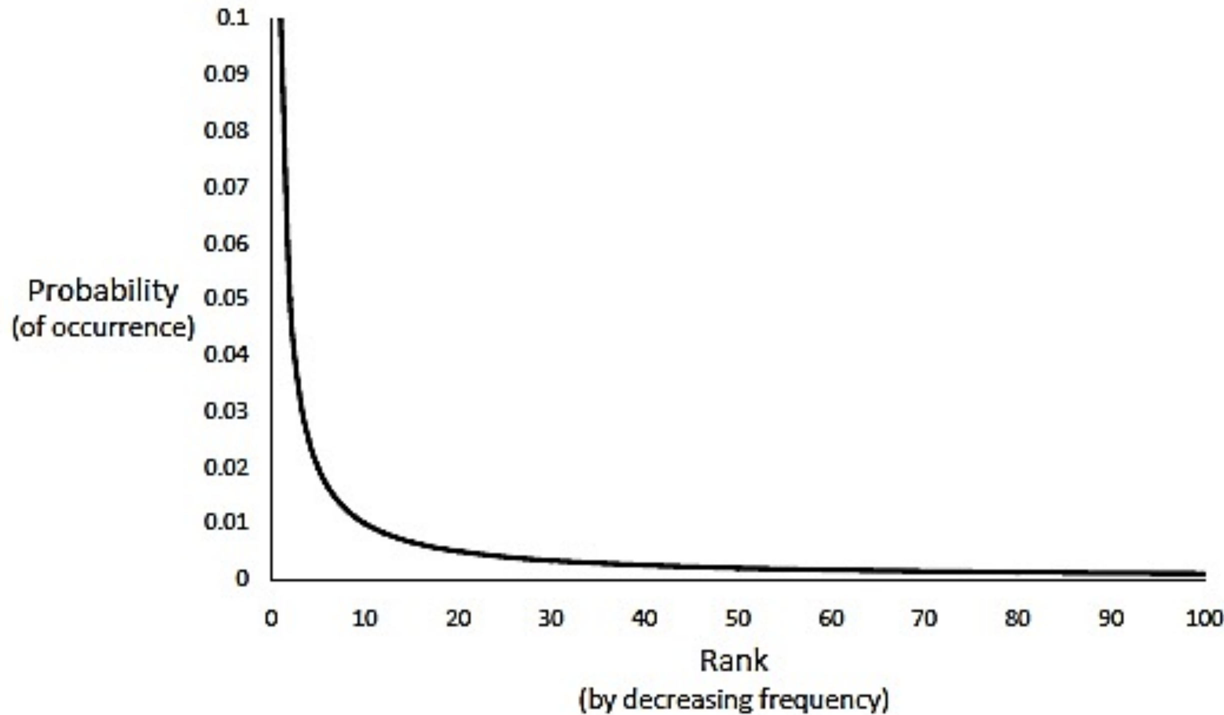
Consider vocabulary of size 60000

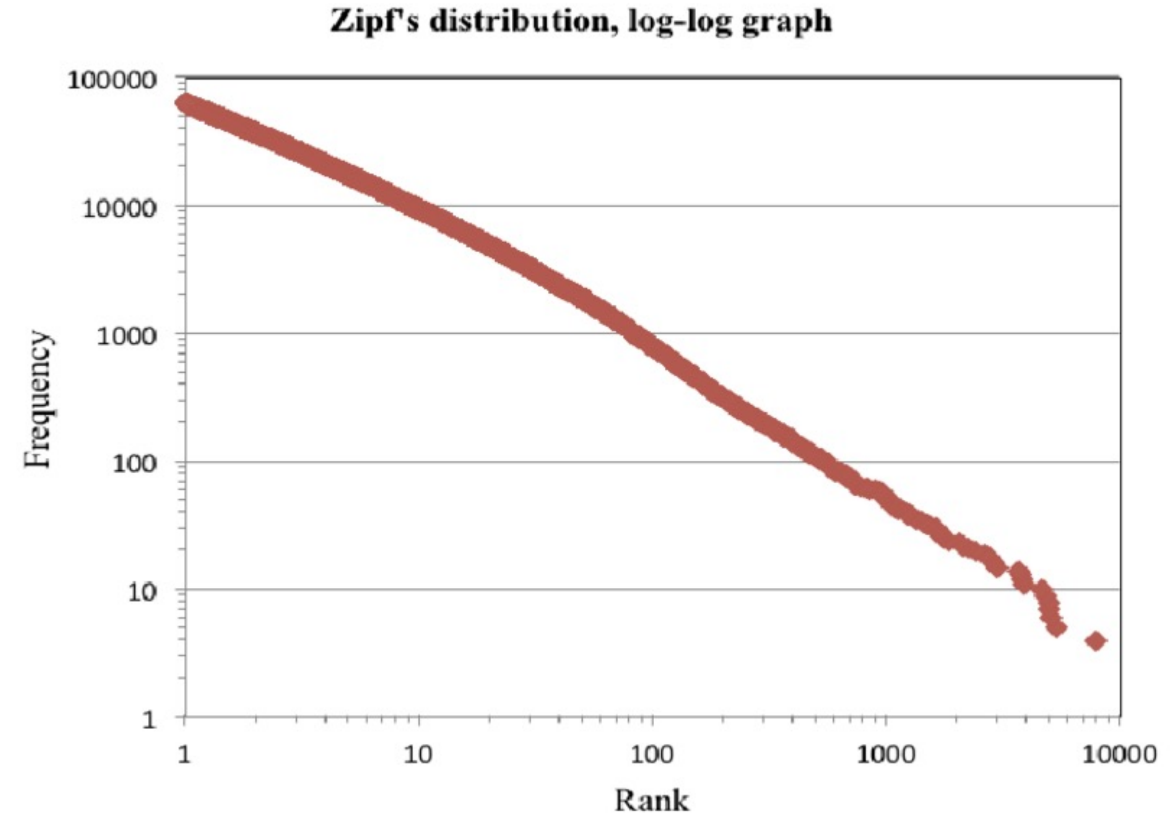
- How many possible unigrams, bigrams, trigrams are there?
- How large a text corpus do we need to obtain reliable statistics for all n-grams?
- Can we solve the problem with more data?

Zipf's law

- Given some corpus of natural language text, the frequency of any word is inversely proportional to its rank in the frequency table
  - The most frequent word will occur approximately twice as often as the second most frequent word
  - The second most frequent word will occur approximately twice as often as the third most frequent word etc

Zipf’s law





Masrai and Milton, 2006.

How to solve the sparsity problems?

- Smoothing
- Sub-word segmentation
- Neural models

Smoothing

The general idea: Find a way to fill the gaps in counts

- Take care not to change the original distribution too much
- Make sure that the smoothed model is still a proper probabilistic model
- Fill in the gaps only as much as needed: as the corpus grows larger there are less gaps to fill.

Smoothing methods

- Add-1 method – simple but stupid
  - add-k method – still simple but a bit less stupid
- Interpolation – various methods, can be quite complex
- (Modified) Kneser-Ney – one of the most successful and widely used methods
- There are others

Add-k method

Assume all n-grams occur k more times than they actually do.

- **Usual bigram probability:**

$$p(w_i|w_{i-1}) = \frac{\#(w_{i-1}, w_i)}{\#(w_{i-1})}$$

- Add $0 < k \leq 1$ to all bigram counts:

$$p_k(w_i|w_{i-1}) = \frac{\#(w_{i-1}, w_i) + k}{\#(w_{i-1}) + k|V|}$$

- Special case $\mathrm{k} = 1$: **add-one smoothing** or **Laplace smoothing**

$$p_{Laplace}(w_i|w_{i-1}) = \frac{\#(w_{i-1}, w_i) + 1}{\#(w_{i-1}) + |V|}$$

Add-k method

- Advantages
  - Very simple
  - Easy to apply
- Disadvantages
  - Performs poorly
  - All unseen events receive the same probability
  - The count of all n-grams is increased by k

Interpolation (Jelinek-Mercer smoothing)

If the bigram $w_{i-1}\, w_i$ is unseen:

- Originally its probability would be 0:

$$p(w_i | w_{i-1}) = 0$$

- Instead of 0 we could use the probability of the shorter n-gram (unigram):

$$p(w_i)$$

- We must make sure that the total probability mass remains the same
- Thus, interpolate between the unigram and bigram distribution

$$p_{JM}(w_i | w_{i-1}) = \lambda p(w_i | w_{i-1}) + (1 - \lambda) p(w_i)$$
$$0 < \lambda < 1$$

Interpolation (Jelinek-Mercer smoothing)

- Recursive formulation:
  - *n*th-order smoothed model is defined recursively as linear interpolation between the *n*th-order maximum likelihood (ML) model and the (*n-1*)th-order smoothed model

$$p_{JM}(w_i|w_{i-k},\dots,w_{i-1})$$
$$= \lambda_{i-k} p(w_i|w_{i-k},\dots,w_{i-1}) + (1-\lambda_{i-k}) p_{JM}(w_i|w_{i-k+1},\dots,w_{i-1})$$

- Can ground the recursion with:
  - 1st order unigram model
  - 0th order uniform model

$$p(w) = \frac{1}{|V|}$$

Software for language modelling

- KenLM: https://github.com/kpu/kenlm
- SRILM: http://www.speech.sri.com/projects/srilm/

- Using an existing model with KenLM python API

```
import kenlm
model = kenlm.Model('lm/test.arpa')
print(model.score('this is a sentence .', bos = True, eos = True))
```

Storing and using an n-gram language model

- An n-gram language model is simply a set of n-grams with their probabilities
  - Computations with n-gram language models are typically very quick
  - However, it can take quite a lot of room in memory
- Using an n-gram language model involves:
- For scoring:
  - segmenting the sentence into overlapping n-grams
  - computing the probability of each word in the sentence given its n-gram context
  - multiplying/summing the logs of these probabilities
- For generating:
  - starting with a suitable context: a sentence start symbol or a desired first word etc
  - sample a next ngram from the probability distribution conditioned on the previous context
  - Continue until the desired end is reached: a special end of sentence symbol, a desired length etc

Feedforward neural language model

Feedforward neural language model

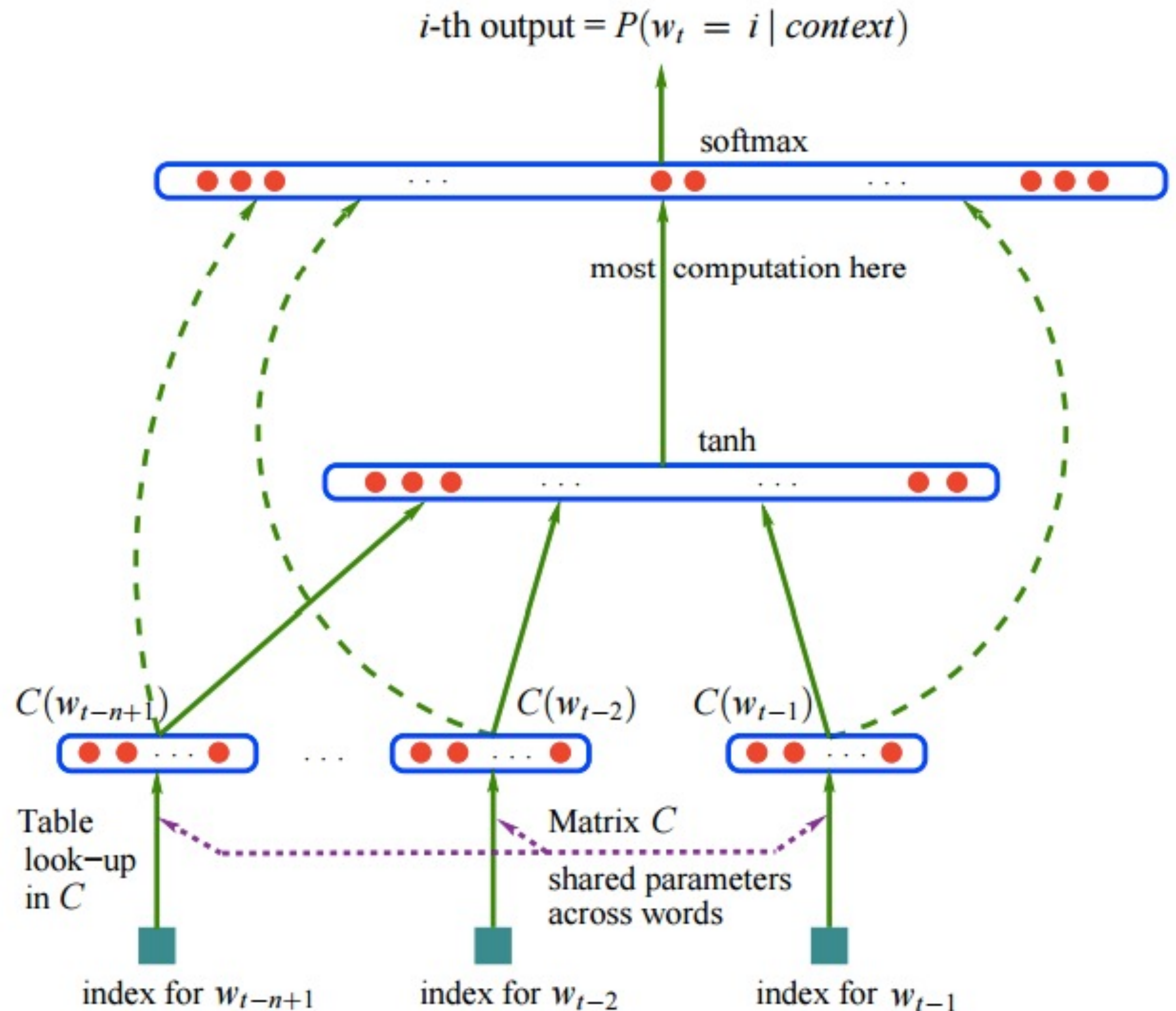
- Is based on feedforward neural network
- Still uses the Markov approximation, i.e. the next word is predicted based on context of fixed length
- Words are represented with word embeddings.
- Therefore:
  - The model can represent arbitrary n-grams (no smoothing needed)
  - The model can better generalize to unseen contexts (because the embeddings of words with similar or related meaning tend to be similar)

Feed-forward neural language model (Bengio et al., 2003)

$$\boldsymbol{x} = [C(w_{t-n+1}); \dots; C(w_{t-2}); C(w_{t-1})]$$

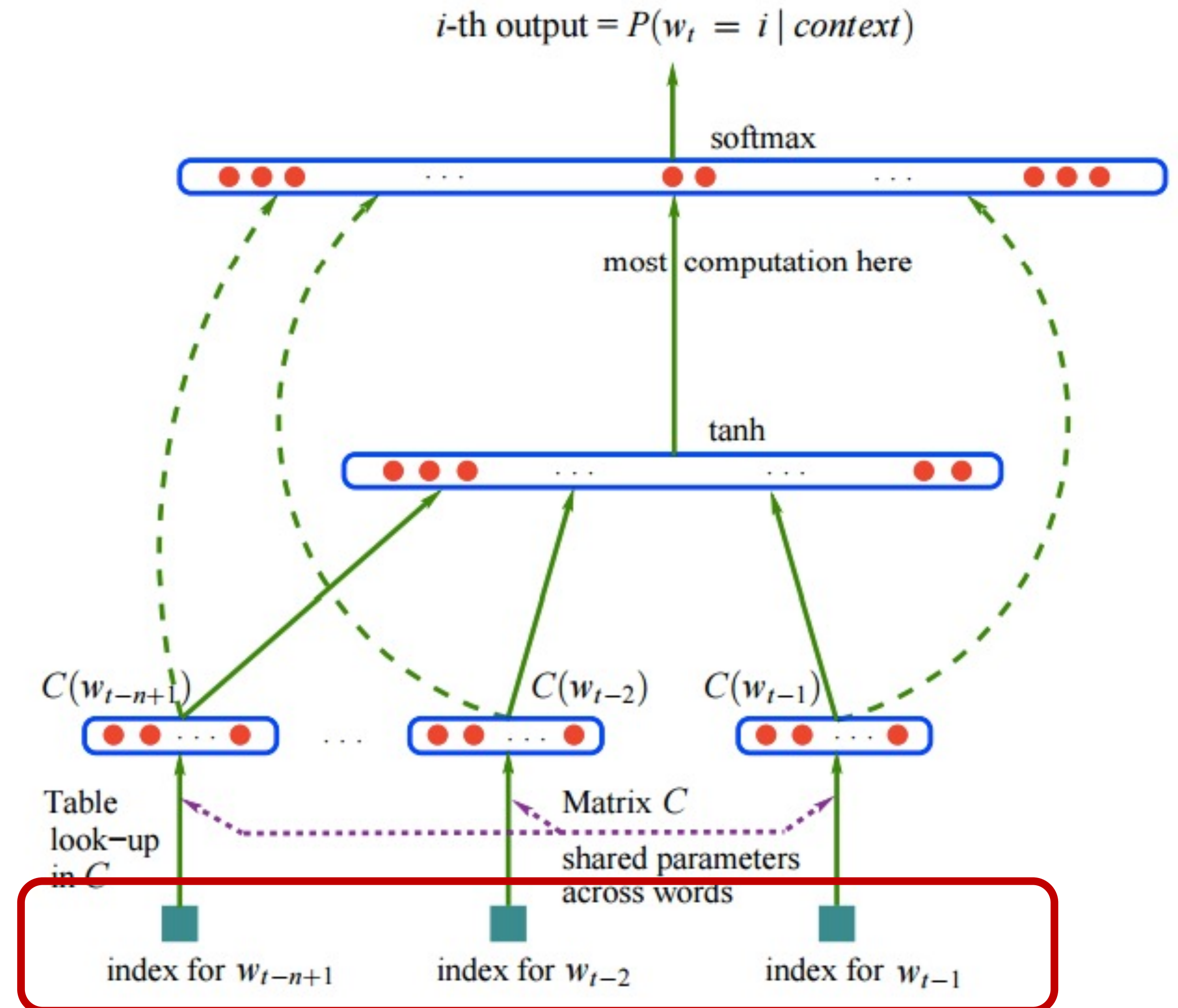
$$\boldsymbol{h} = g(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}^{W})$$

$$p(w_t | w_{t-n+1}, \dots, w_{t-2}, w_{t-1}) = \text{softmax}(\boldsymbol{h}\boldsymbol{U} + \boldsymbol{b}^{U})$$



Input to the model

- Input words are encoded with one-hot representations
- Each word is represented with a vector of length |V|
- Each word has an index $i = 1 \dots |V|$
- The vectors are filled with zeros, there is only one 1 in the position of the index of the word

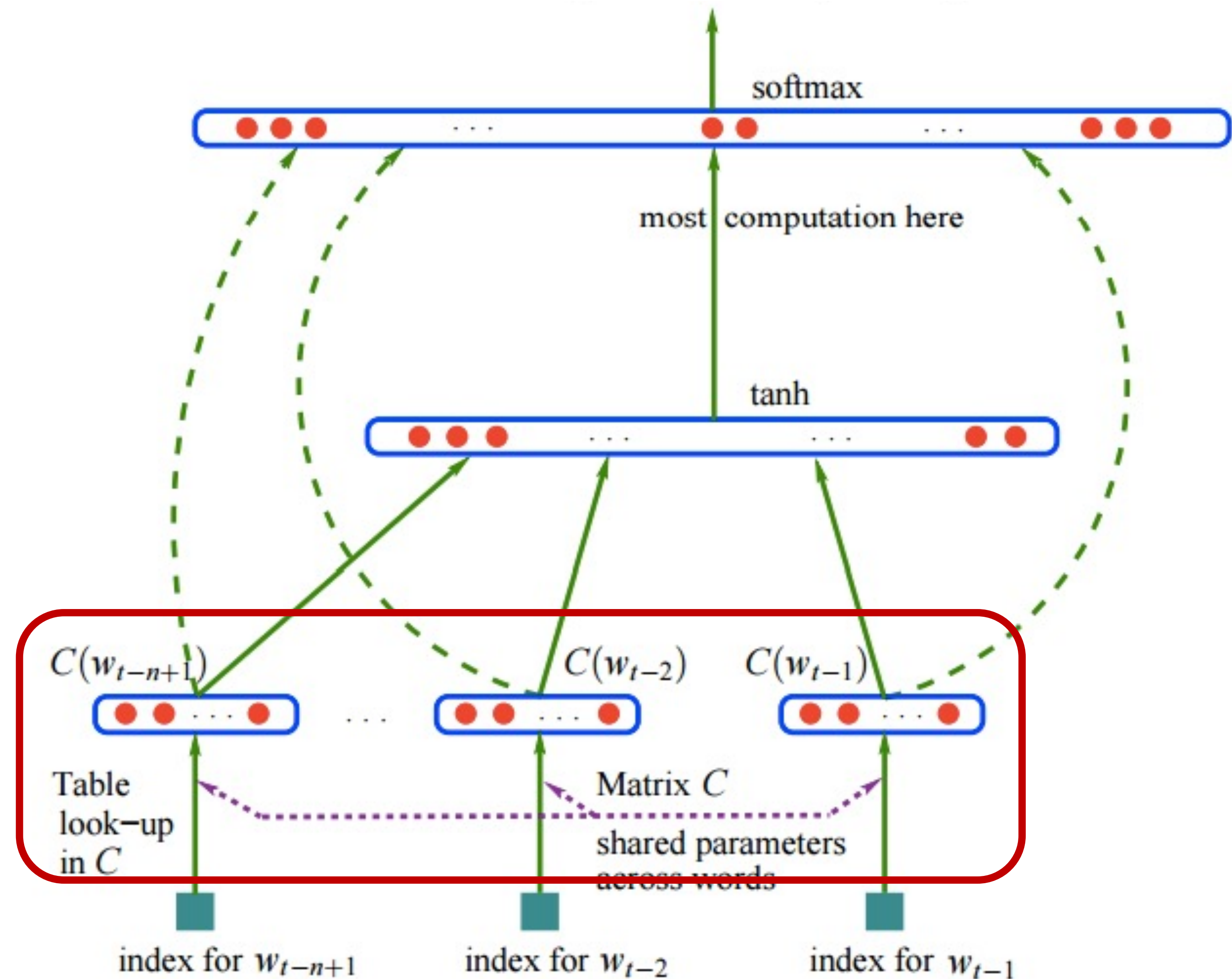


Embedding layer

- Embedding matrix (lookup table, word embeddings, embedding layer) C:

$$C \in \mathbb{R}^{|V| \times d}$$

- **This embedding matrix is used virtually in every neural model for NLP**



Computing context representation

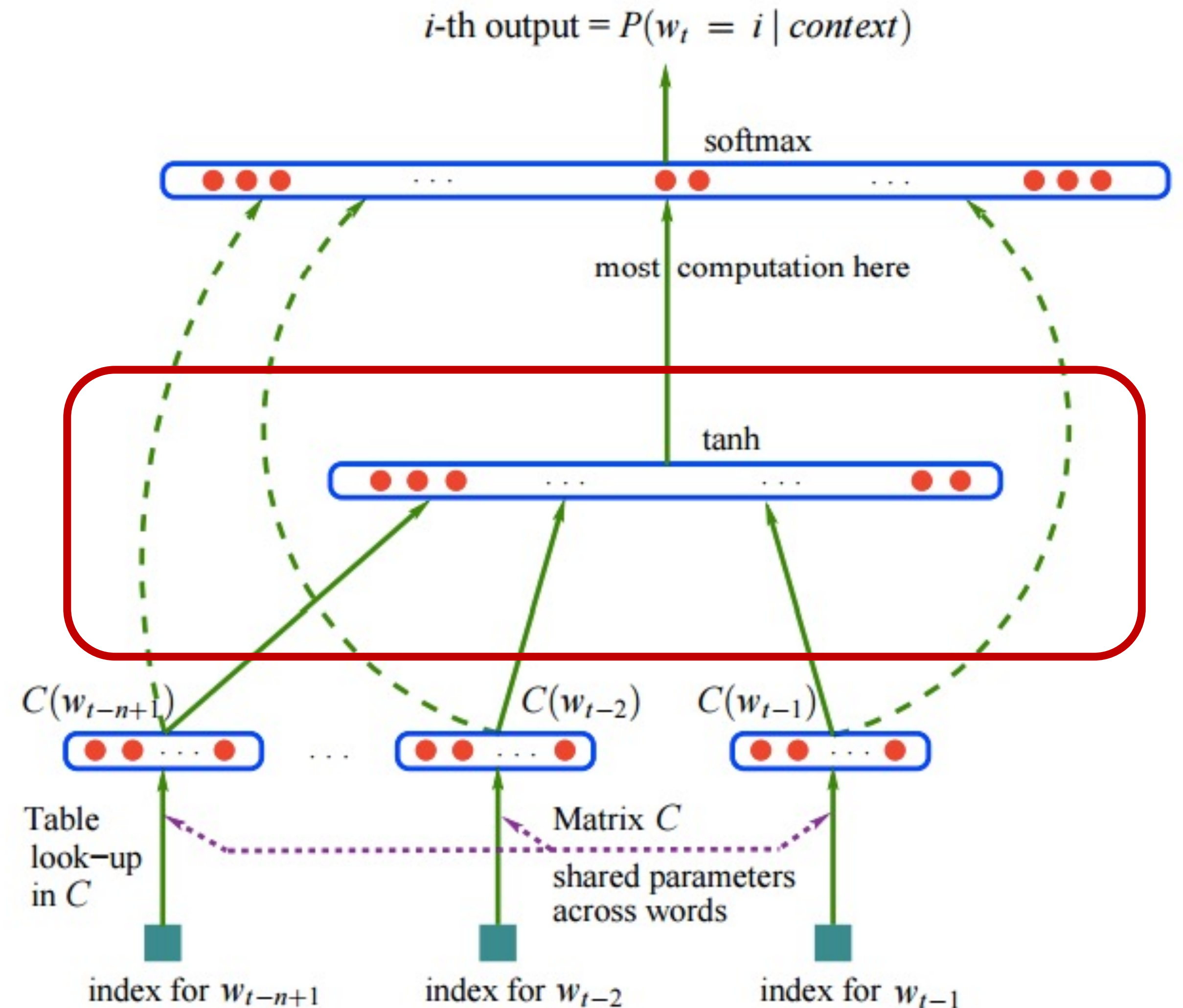
- Concatenate word embeddings

$$x = [C(w_{t-n+1}); \dots; C(w_{t-2}); C(w_{t-1})]$$

- Compute context representation

$$h = g(xW \quad + b^W)$$

g – non-linearity

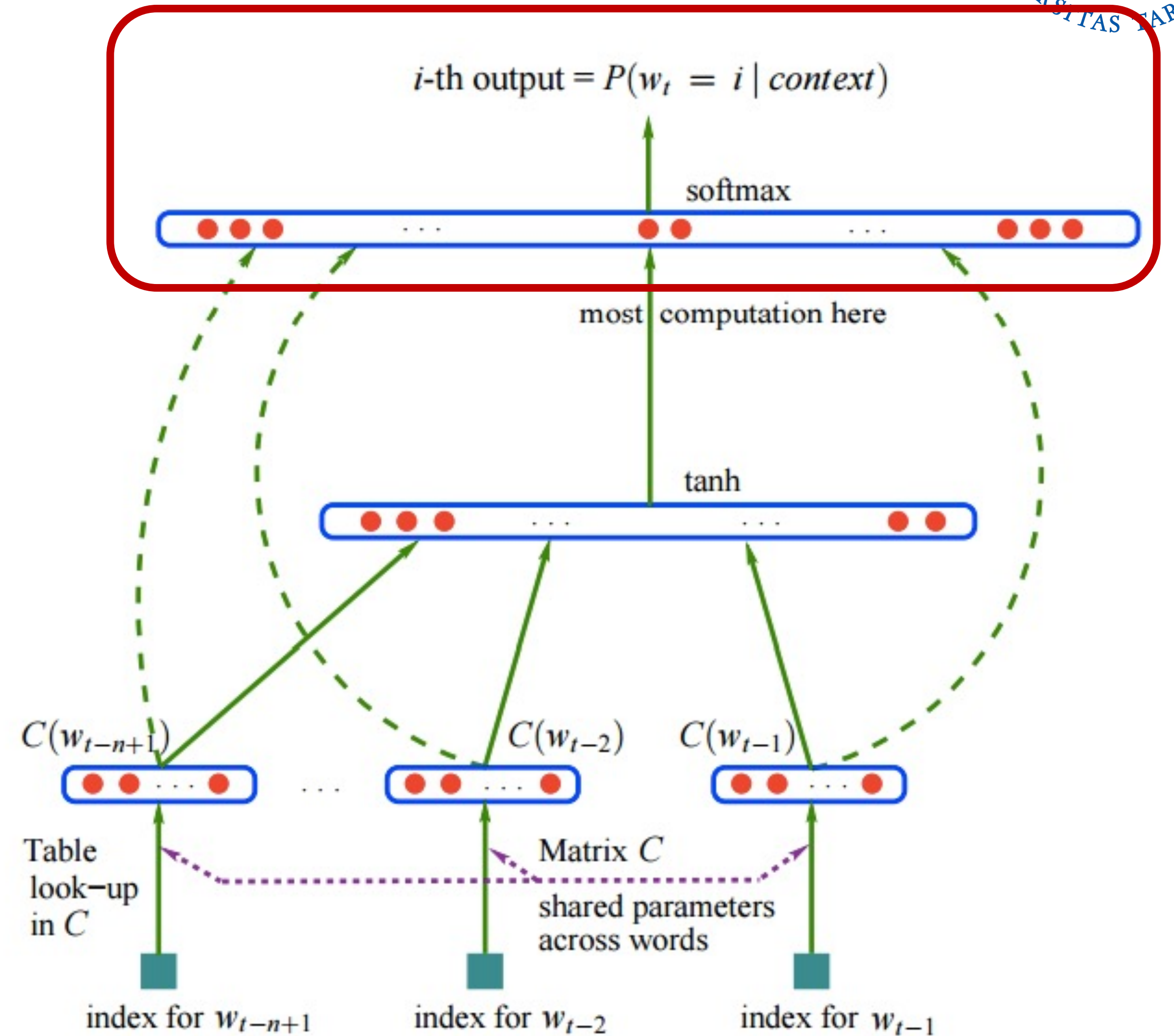


Classifier layer

$$p(w_t|w_1, \ldots, w_{t-2}, w_{t-1}) \cong$$

$$p(w_t|w_{t-n+1}, \ldots, w_{t-2}, w_{t-1}) =$$

$$\text{softmax}(\boldsymbol{hU} + \boldsymbol{b^U})$$



Softmax

- Convert real numbers into probabilities
  - exponentiate --> ensure that all values are positive
  - normalise → ensure that all values are less than 1 and sum to 1

- $\boldsymbol{x} = [x_1, x_2, \dots, x_n]$
- $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$
- $\text{softmax}(\boldsymbol{x}) = \left[\frac{e^{x_1}}{Z}, \frac{e^{x_2}}{Z}, \dots, \frac{e^{x_n}}{Z}\right] \; Z = \sum_{j=1}^{n} e^{x_j}$

Training the language model with cross-entropy loss

$$L_{cross-entropy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{|V|} y_i \log p(\hat{y}_i)$$

Consider the word in context: *P*(flower|the girl with a)

Compute the cross-entropy loss for this training item.

	the	girl	with	flowers	is	cute	are	were	flower	…	…
$\boldsymbol{y}$	0	0	0	0	0	0	0	0	1	0	0
$p(\hat{\boldsymbol{y}})$	0.06	0.15	0.009	0.07	0.0003	0.001	0.005	0.003	0.63	…	…
$\log p(\hat{\boldsymbol{y}})$	-2.81	-1.90	-4.71	-2.66	-8.11	-6.91	-5.30	-5.81	-0.46		

Training the language model with cross-entropy loss

$$L_{cross-entropy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{|V|} y_i \log p(\hat{y}_i) = -\log p(\hat{y}_t)$$

|V| - the vocabulary size

t - index of the correct word in the vocabulary

$\boldsymbol{y}$ - correct prediction, one-hot vector, one only in the position t

$\hat{\boldsymbol{y}}$ - vector of predicted probabilities

Evaluation - perplexity

Language model evaluation

- Intrinsic evaluation
  - Perplexity
  - Quick and simple
  - Improvements in perplexity might not translate into improvements in downstream tasks
- Extrinsic evaluation
  - In a down-stream task (like machine translation, speech recognition etc)
  - More difficult and time-consuming
  - More accurate evaluation (although beware of confounding with other factors)

Perplexity

- **Perplexity** is a measurement of how well a probability model predicts a sample.
- Language model is a probability model over text
- To evaluate a language model, compute the perplexity over a held-out set (test set)
- The lower the perplexity the better the language model, i.e., the less "surprised" the model is on seeing the evaluation data

Computing perplexity

- Perplexity pp

$$pp = P(w_1 w_2 \dots w_n)^{-\frac{1}{n}} =$$

$$= \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}}$$

- The probability of the text is computed according to a particular model
- For bigrams for instance:

$$pp = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{i-1})}}$$

Computing in the log-space

- Often or usually language modeling probabilities are computed in the log space because
  - The probabilities for individual words can be quite small
  - Multiplying many small probabilities makes the result even smaller --> numeric underflow

$$\log pp = -\frac{1}{n}\sum_{i=1}^{n} \log P(w_i|w_{i-1})$$

What does the perplexity value mean?

- Let's assume that the logarithm of the perplexity on a test set is 7.95
- This means that each word in the test set could be encoded with 7.95 bits
- The model perplexity would be $2^{7.95}$=247 per word
- This means that the model is as confused on test data as if it would have to choose uniformly at random from 247 possibilities for each word.

Valid use of perplexity

- Perplexity is corpus-specific: only the perplexities calculated on the same test set are comparable
- For meaningful comparison, the vocabularies of the two language models must be the same, e.g.
  - You can compare a bigram language model to a trigram language model that both use the same vocabulary of size 10000
  - You cannot compare a trigram language model using a vocabulary of size 10000 with a trigram language model using vocabulary of size 20000
- In an ”open-vocabulary” setting, typically the perplexity per character is computed
  - This way can compare for instance a character-based model with a sub-word-based model