



n -grams and fundamentals of language models

2026–27

Ion Androutsopoulos

<http://www.aueb.gr/users/ion/>

Contents

- n -gram language models.
- Estimating probabilities from corpora.
- Entropy, cross-entropy, perplexity.
- Greedy, top- k , top- p sampling.
- Beam-search decoding.
- Application in spelling correction.
- Sub-word tokenizers.

Language models

- **How probable** is it to encounter (e.g., in news articles) the following sentences (word sequences)?
 - *The government announcement new austerity metrics hopping to decrease the deficit.*
 - *The government announced new austerity measures hoping to reduce the deficit.*
- In many cases, **candidate alternative sentences** are produced. We wish to keep the **most probable** ones.
 - Speech recognition, optical character recognition, (statistical) machine translation, smartphone keyboards, spelling and syntax checking...

n -gram language models

- **Notation** for sequences of words:

$$\langle w_1, w_2, \dots, w_k \rangle = w_1^k$$

- **n -gram**: sequence of **n consecutive words**.
 - **Trigrams**: “the government announced”, “government announced new”, “announced new austerity”, ...
 - **Bigrams**: “the government”, “government announced”, “announced new”, “new austerity”, ...
 - In other cases, sequences of **n consecutive characters**.
- **Chain rule**:

$$P(w_1^k) = P(w_1, \dots, w_k) = P(w_1) \cdot P(w_2 \mid w_1) \cdot \\ P(w_3 \mid w_1, w_2) \cdot P(w_4 \mid w_1^3) \cdots P(w_k \mid w_1^{k-1})$$

How do we estimate the probabilities?

- Simplest approach: **maximum likelihood estimates** from a **corpus** of C tokens:

$$P_{MLE}(\text{the}) = \frac{c(\text{the})}{C}$$

$$P_{MLE}(\text{government} \mid \text{the}) = \frac{c(\text{the, government})}{c(\text{the})}$$

$$P_{MLE}(\text{announced} \mid \text{the, gov}) = \frac{c(\text{the, gov, announced})}{c(\text{the, gov})}$$

- Many n -grams (esp. 4-grams, 5-grams, ...) will be **very rare** or **may not occur** even in a **large corpus**.
 - Very **poor** or **zero** probability **estimates**.
 - Leading to a **zero chain product**...

Markov assumption

- **Bigram** language model:

$$P(w_1^k) = P(w_1, \dots, w_k) = P(w_1) \cdot P(w_2 | w_1) \cdot \\ P(w_3 | w_1, w_2) \cdot P(w_4 | w_1^3) \cdots P(w_k | w_1^{k-1}) \simeq$$

$$P(w_1 | start) \cdot P(w_2 | w_1) \cdot P(w_3 | w_2) \cdots P(w_k | w_{k-1})$$

- **Trigram** language model:

$$P(w_1 | start_1, start_2) \cdot P(w_2 | start_2, w_1) \cdot P(w_3 | w_1, w_2) \cdot \\ P(w_4 | w_2, w_3) \cdots P(w_k | w_{k-2}, w_{k-1})$$

- **Stationarity:** We assumed probabilities do not depend on where the n -grams are encountered. E.g., in $P(\text{announced} | \text{the, government})$, we do not examine if “announced” occurs as the 3rd or 4th or ... word in the sentences of the corpus.
- Strictly speaking, we also need an *end* pseudo-token. See study exercises.

Laplace smoothing

- Even with a Markov assumption, we will still have many ***n*-grams** that **do not occur in the corpus**.
- **Laplace smoothing for unigrams:** if we have $|V|$ vocabulary words (distinct words),

$$P_{Laplace}(W = w) = \frac{c(w) + 1}{C + |V|}$$

Add a pseudo-occurrence of each vocabulary word. More generally, of each possible value of the random variable (here W).

- Similarly, e.g., for **trigrams**:

$$P_{Laplace}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + 1}{c(w_{k-2}, w_{k-1}) + |V|}$$

Add a pseudo-occurrence of each possible trigram that starts with w_{k-2}, w_{k-1} . There are $|V|$ such trigrams in total.

- But we **over-estimate** rare bigrams, trigrams, ...

Add- α smoothing

- For **unigrams**: if we have $|V|$ vocabulary words,

$$P_{Laplace}(W = w) = \frac{c(w) + \alpha}{C + \alpha \cdot |V|}$$

We tune α ($0 \leq \alpha \leq 1$), e.g., on held-out data (see below).

- Similarly, e.g., for **trigrams**:

$$P_{Laplace}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + \alpha}{c(w_{k-2}, w_{k-1}) + \alpha \cdot |V|}$$

- Better, but still **poor estimates** for language models.
 - In practice, Laplace and add- α smoothing are **not used in language models** (but often work well in classification tasks).
 - See **optional reading slides for better estimates** for n -gram LMs (e.g., Knesser-Ney smoothing, backoff models).

Linear interpolation

- We use a **linear combination** of estimates from ***n*-gram language models** with different *n* values.

$$P_{\text{int}}(w_k \mid w_{k-2}, w_{k-1}) = \lambda_1 \cdot P(w_k \mid w_{k-2}, w_{k-1}) + \lambda_2 \cdot P(w_k \mid w_{k-1}) + \lambda_3 \cdot P(w_k) \quad \text{with } \sum_{i=1}^3 \lambda_i = 1$$

- We can tune $\lambda_1, \lambda_2, \lambda_3$, e.g., on held-out data (see below).

LMs as next word predictors

- **Sequence probability** using a **bigram LM**:

$$P(w_1^k) = P(w_1, \dots, w_k) = P(w_1) \cdot P(w_2 | w_1) \cdot \\ P(w_3 | w_1, w_2) \cdot P(w_4 | w_1^3) \cdots P(w_k | w_1^{k-1}) \simeq$$

$$P(w_1 | start) \cdot P(w_2 | w_1) \cdot P(w_3 | w_2) \cdots P(w_k | w_{k-1})$$

- We can think of the **LM** as providing **at each position** ($1, \dots, k$) a **distribution** $P(w|w_{i-1})$, showing how probable it is for every word $w \in V$ to be the **next one**.
 - Or a **distribution** $P(w|w_{i-2}, w_{i-1})$ for a **trigram LM**.
 - Or $P(w|h_1^{i-1})$ for an LM that considers all the “**history**” (previous words) h_1^{i-1} , e.g., in an **RNN LM** (next lectures).

LMs as next word predictors (cont.)

- If we are asked to **estimate the probability of a particular word sequence** $w_1, w_2, \dots, w_n$, using a **bigram model**,
 - e.g., $w_1, \dots, w_n$ = “the government announcement new austerity metrics hopping to decrease the deficit”,
 - get $P(\text{the}|\textit{start})$ from distribution $P(w|\textit{start})$,
 - get $P(\text{government}|\text{the})$ from distribution $P(w|\text{the})$,
 - ...
 - get $P(\text{deficit}|\text{the})$ from distribution $P(w|\text{the})$,
 - **multiply** them (in practice **add their logarithms**).
 - It would be better to also **multiply** $P(\textit{end}|\text{deficit})$.
- **Similarly for trigram or RNN LMs.**

LMs as text generators

- LMs can also **generate text**. E.g., with **greedy decoding**:
 - **Select as w_1** the word with the **highest $P(w|start)$** .
 - **Select as w_2** the word with the **highest $P(w|w_1)$** .
 - ...
 - Keep adding words **until you select *end*** (also in V).
 - **Similarly for trigram or RNN LMs.**
- Alternatively, with **sampling** (for more varying texts):
 - **Select w_1 by sampling from $P(w|start)$** .
 - **Select w_2 by sampling from $P(w|w_1)$** .
 - ... **until you select *end*** (also in V).
 - **Similarly for trigram or RNN LMs.**

How do we sample?

- **Segment the (0,1) interval** into segments:
 - One segment **per vocabulary word** $w \in V$.
 - **Length of segment** for each w : $P(w|h)$
 - h is the **history** (e.g., prev. 2 words in a bigram model).
 - **Draw** a number $x \sim \text{Uniform}(0,1)$.
 - **Select the word** whose **segment** x falls in.
- If the vocabulary has **only three words** (a, b, c):

$P(a h) = 0.2$	$P(b h) = 0.5$	$P(c h) = 0.3$
----------------	----------------	----------------



How do we sample with temperature?

Original distribution:

$P(a h) = 0.2$	$P(b h) = 0.5$	$P(c h) = 0.3$
----------------	----------------	----------------

New distribution with **high-temp softmax** ($t = 10$):

$P(a h) = 0.33$	$P(b h) = 0.34$	$P(c h) = 0.33$
-----------------	-----------------	-----------------

New distribution with **low-temp softmax** ($t = 0.07$):

$P(a h) = 0.01$	$P(b h) = 0.93$	$P(c h) = 0.05$
-----------------	-----------------	-----------------



How do we sample with temperature?

- **Before segmenting** the $(0,1)$ interval, **pass the probabilities** $P(w|h)$ (or log-probs) **through softmax** with temperature t .
 - **softmax** converts a given **set of real numbers** (not necessarily probs) to a **(new) probability distribution**.
 - For very **large t** , all numbers are given almost the **same (new) probability** (same segment length).
 - For very **small t** , the **largest given number** gets almost all the probability (in effect, greedy decoding).

$$\text{softmax}\left(\begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_{|V|} \end{bmatrix}, t\right) = \begin{bmatrix} \frac{\exp(p_1/t)}{\sum_{j=1}^{|V|} \exp(p_j/t)} \\ \frac{\exp(p_2/t)}{\sum_{j=1}^{|V|} \exp(p_j/t)} \\ \dots \\ \frac{\exp(p_{|V|}/t)}{\sum_{j=1}^{|V|} \exp(p_j/t)} \end{bmatrix}$$

Top-k sampling

Suppose $k = 3$

Take the words and renormalize (so the probabilities sum up to 1).

Token	Prob.
<i>cat</i>	0.40
<i>dog</i>	0.25
<i>then</i>	0.15
<i>water</i>	0.10
<i>book</i>	0.06
<i>ran</i>	0.04

Token	Prob.
<i>cat</i>	$0.40/0.8 = 0.50$
<i>dog</i>	$0.25/0.8 = 0.31$
<i>then</i>	$0.15/0.8 = 0.19$

Dong Nguyen (2026)

From the slides of Don Nguyen's talk "How language models work: From n-grams to instruction-tuned LLMs" at AthensNLP 2026
(<https://athnlp.github.io/2026/presentations.html>).

Top- p Sampling

Similar to top- k sampling, but select the smallest set of tokens (V_{sel}) such that:

$$\sum_{w \in V_{sel}} P(w \mid \mathbf{w}_{<t}) \geq p$$

Like before, renormalize and then sample.

cat: $0.40/0.65 = 0.62$

dog: $0.25/0.65 = 0.38$

$$p = 0.6$$

Token	Prob.
<i>cat</i>	0.40
<i>dog</i>	0.25
<i>then</i>	0.15
<i>water</i>	0.10
<i>book</i>	0.06
<i>ran</i>	0.04

Dong Nguyen (2026)

From the slides of Don Nguyen's talk "How language models work: From n-grams to instruction-tuned LLMs" at AthensNLP 2026
(<https://athnlp.github.io/2026/presentations.html>).

Top- k and top- p sampling

- Instead of simple sampling, with **top- k sampling**:
 - **Select w_1 by sampling from $P(w|start)$ after keeping in the distribution only the words with the top k probabilities and renormalizing.**
 - **Select w_2 by sampling from $P(w|w_2)$ after keeping in the distribution only the words with the top k probabilities and renormalizing.**
 - **... until you select *end* (also in V).**
 - **Similarly for trigram or RNN LMs.**
- **Top- p or “nucleus” sampling** is similar but **keeps in the distribution of each step the top words whose probabilities add up to p (e.g., 90%), before renormalizing.**

top- k and top- p can be combined with temperature sampling.

Spelling correction/normalization

- The words we see:

w_1^k : He **pls gd ftball**.

- Possible candidate corrections (**hypotheses**):

t_1^k : He **please god football**.

t_1^k : He **plays god football**.

t_1^k : He **plays good football**.

t_1^k : He **players good football**.

...

t_1^k : He **pleases god ball**.

The **green words** are **vocabulary words** with **small distance** (e.g., edit distance) from the **out-of-vocabulary words**.

A **language model** estimates **how well** the **words** of each candidate hypothesis **fit together**.

We often have too many hypotheses

- If we have **very few wrong words** in the input, we may also have **very few hypotheses**.
 - Then we could **exhaustively consider all the hypotheses** and keep the one with the **best LM score**.
- But in practice the **hypotheses** may be **too many**!
- The **wrong words** may actually be **vocabulary words**!
 - **1st type**: “he plays good football” → “he **pls gd ftball**”.
 - **2nd type**: “he plays good football” → “he **please god ftball**”.
 - To cope with the **2nd type of error**, we need to **treat every word** (even vocabulary words) as **possibly needing replacement**. We get **many more hypotheses**.

Correcting errors of both types

- The words we see:

w_1^k : He pls god ftball.

- Possible corrections (**hypotheses**):

t_1^k : He please god football.

t_1^k : He plays god football.

t_1^k : He plays good football.

t_1^k : Her players good football.

...

t_1^k : Her pleases god ball.

We now **replace every word** (even vocabulary words) by **close vocabulary words** (treating also the original word as possible “replacement”, if it’s a vocabulary word).

Again, a **language model** estimates **how well** the **words** of each hypothesis **fit together**.

A noisy channel model

- We assume that **all the words** were **initially correct**, but were **transmitted** through a **noisy channel**.
 - Here the channel distorts the words by occasionally **inserting**, **deleting**, or **replacing** letters.
- We try to **guess the initial** (correct) words **from the observed** ones, i.e., we “**decode**” the channel output.
 - Initial (correct) words: $t_1^k = \langle t_1, t_2, \dots, t_k \rangle$
 - Observed words: $w_1^k = \langle w_1, w_2, \dots, w_k \rangle$
- We seek the **most probable initial words**:

$$\hat{t}_1^k = \arg \max_{t_1^k} P(t_1^k | w_1^k) = \arg \max_{t_1^k} \frac{P(t_1^k) \cdot P(w_1^k | t_1^k)}{\cancel{P(w_1^k)}}$$

The most probable initial words

- For a **given observed sequence** (e.g., sentence) w_1^k :

$$\hat{t}_1^k = \arg \max_{t_1^k} P(t_1^k | w_1^k) = \arg \max_{t_1^k} P(t_1^k) \cdot P(w_1^k | t_1^k)$$

All the **hypotheses** (candidate initial word sequences)

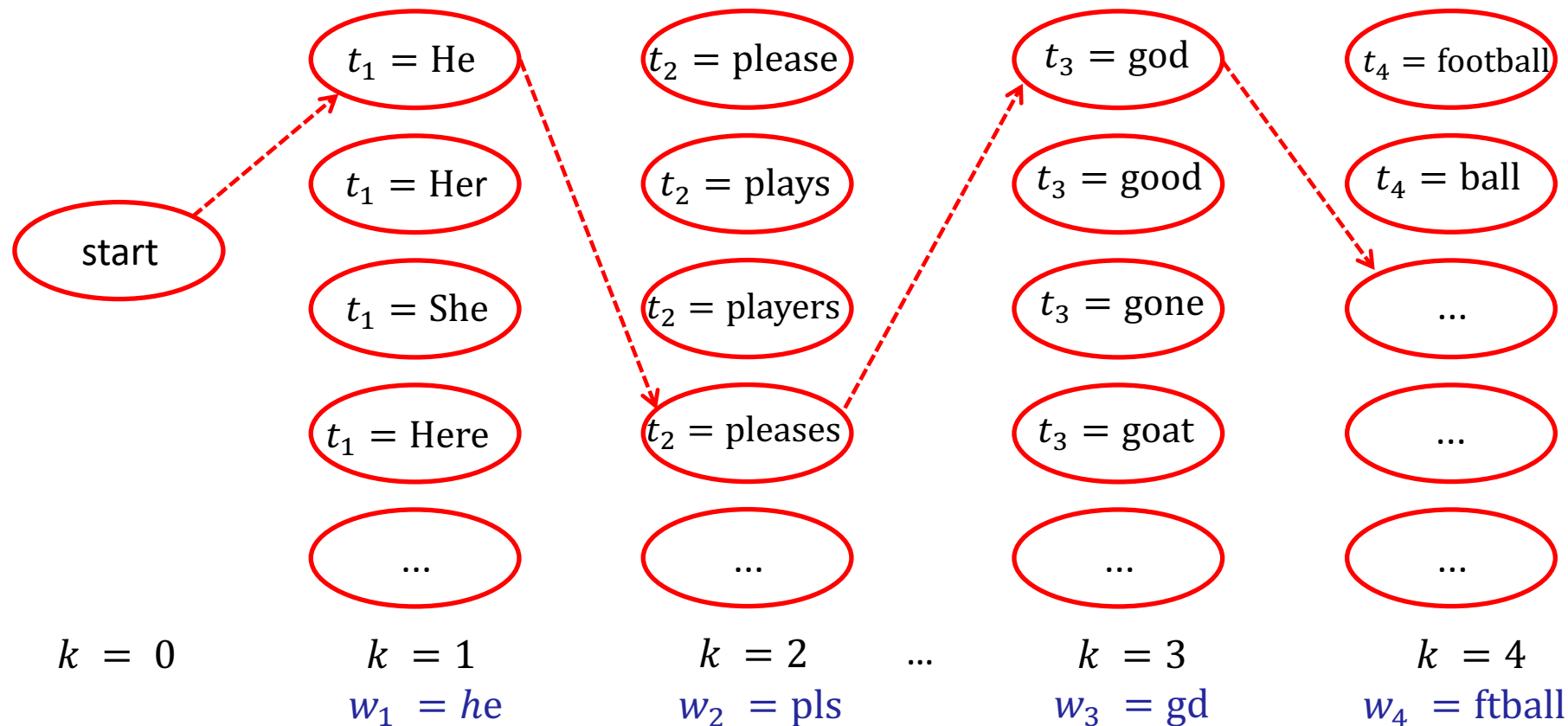
Language model
(e.g., trigram model)

Simplest approach: “probabilities” **inversely proportional to edit distance** ($1/[ED(w_i, t_i) + \varepsilon]$). See **J&M** for better ideas.

$$\begin{aligned} P(w_1^k | t_1^k) &= P(w_1 | t_1^k) \cdot P(w_2 | w_1, t_1^k) \cdot \dots \cdot P(w_k | w_1^{k-1}, t_1^k) \\ &\simeq P(w_1 | t_1) \cdot P(w_2 | t_2) \cdot \dots \cdot P(w_k | t_k) = \prod_{i=1}^k P(w_i | t_i) \end{aligned}$$

- The **best candidate sequence** $t_1, \dots, t_k$ can be found with **heuristic search** (e.g., beam search) or **dynamic programming**.

The search space

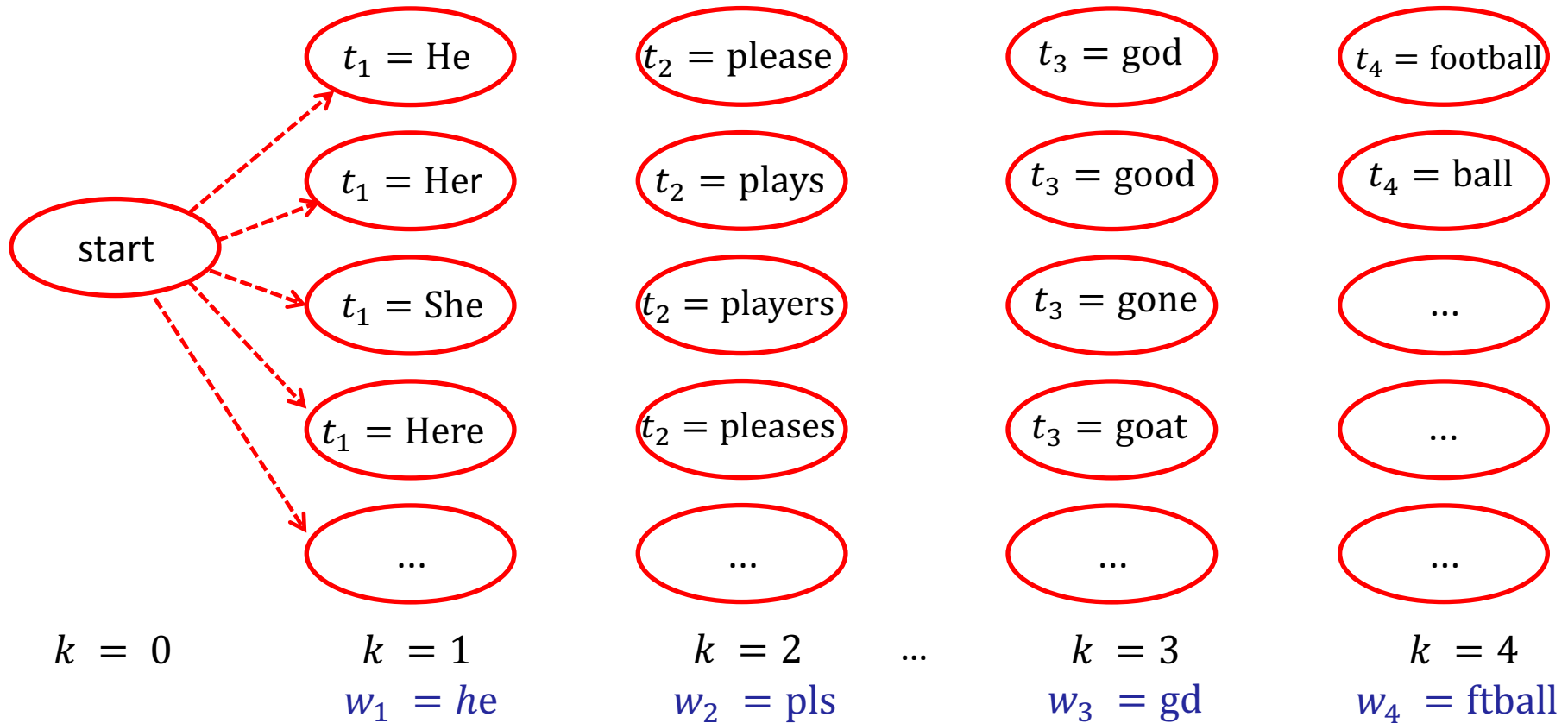


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$
 For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

Greedy search may not find the path with the highest overall probability.

Beam search decoder

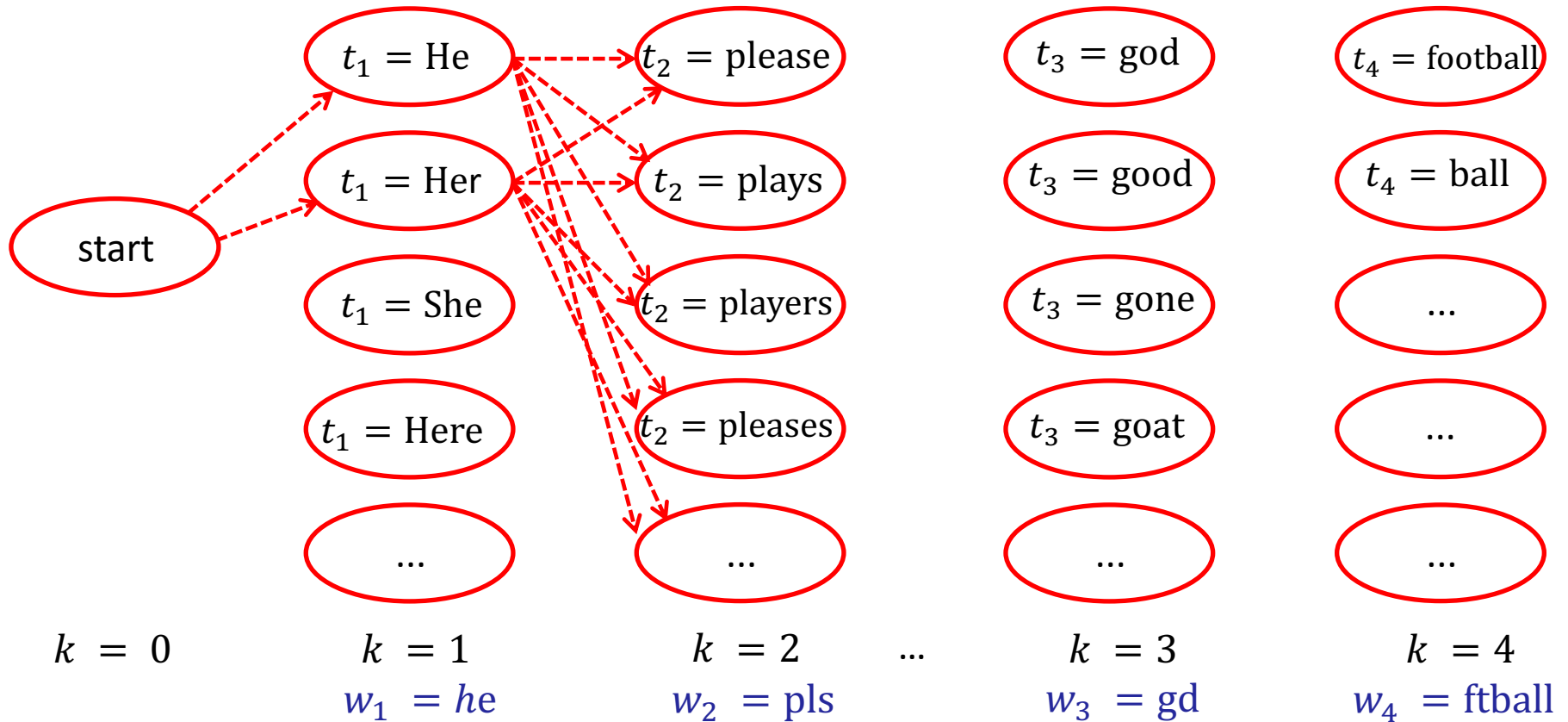


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

Beam search decoder

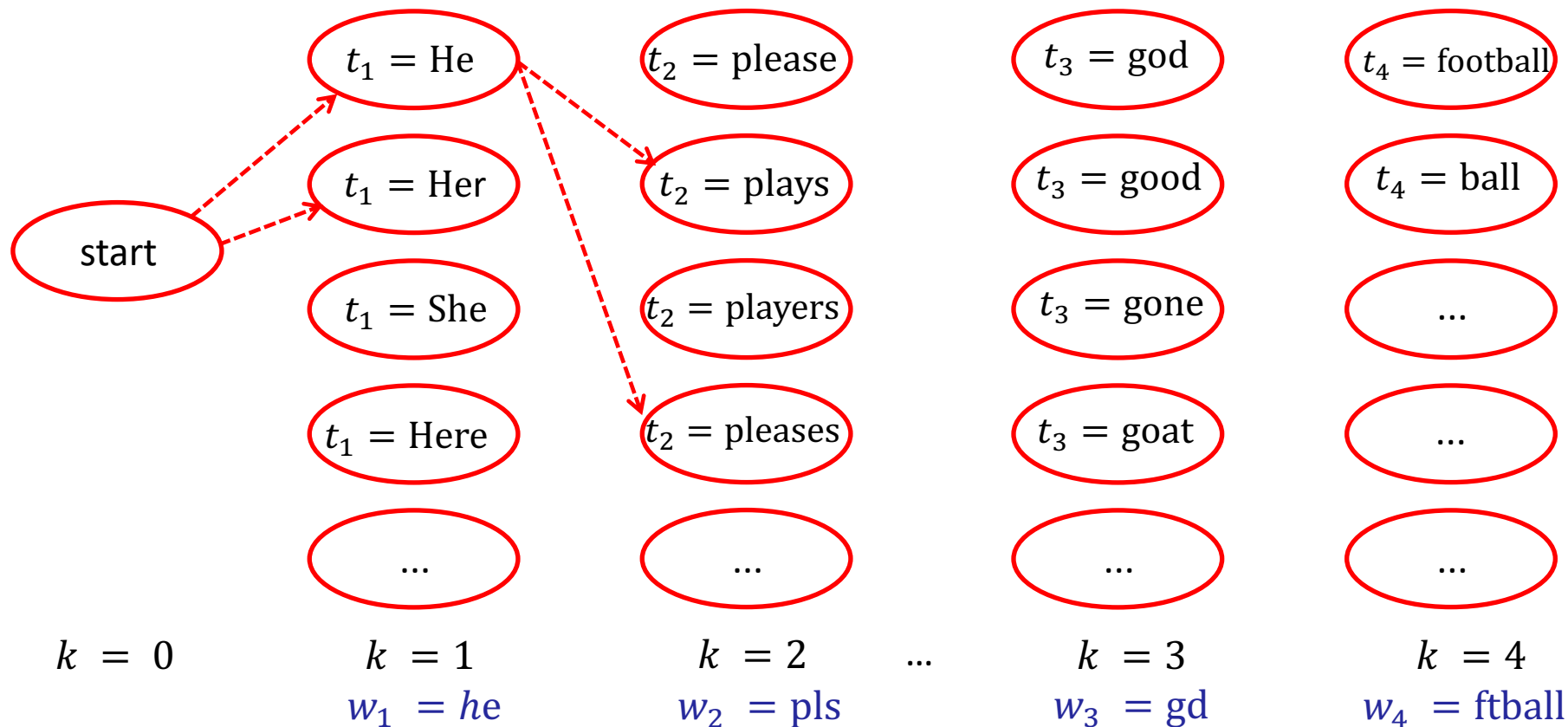


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$
 For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

Beam search decoder



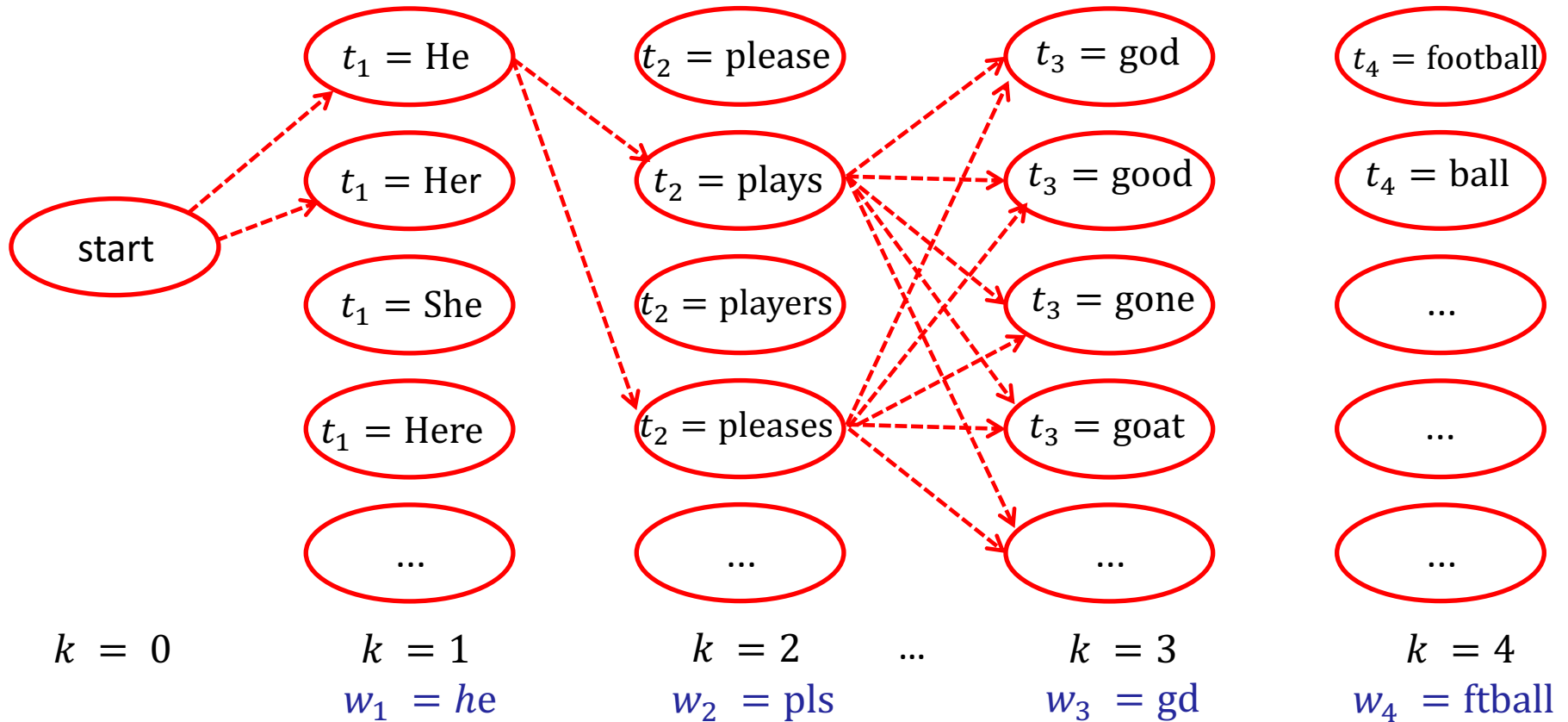
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

Beam search decoder

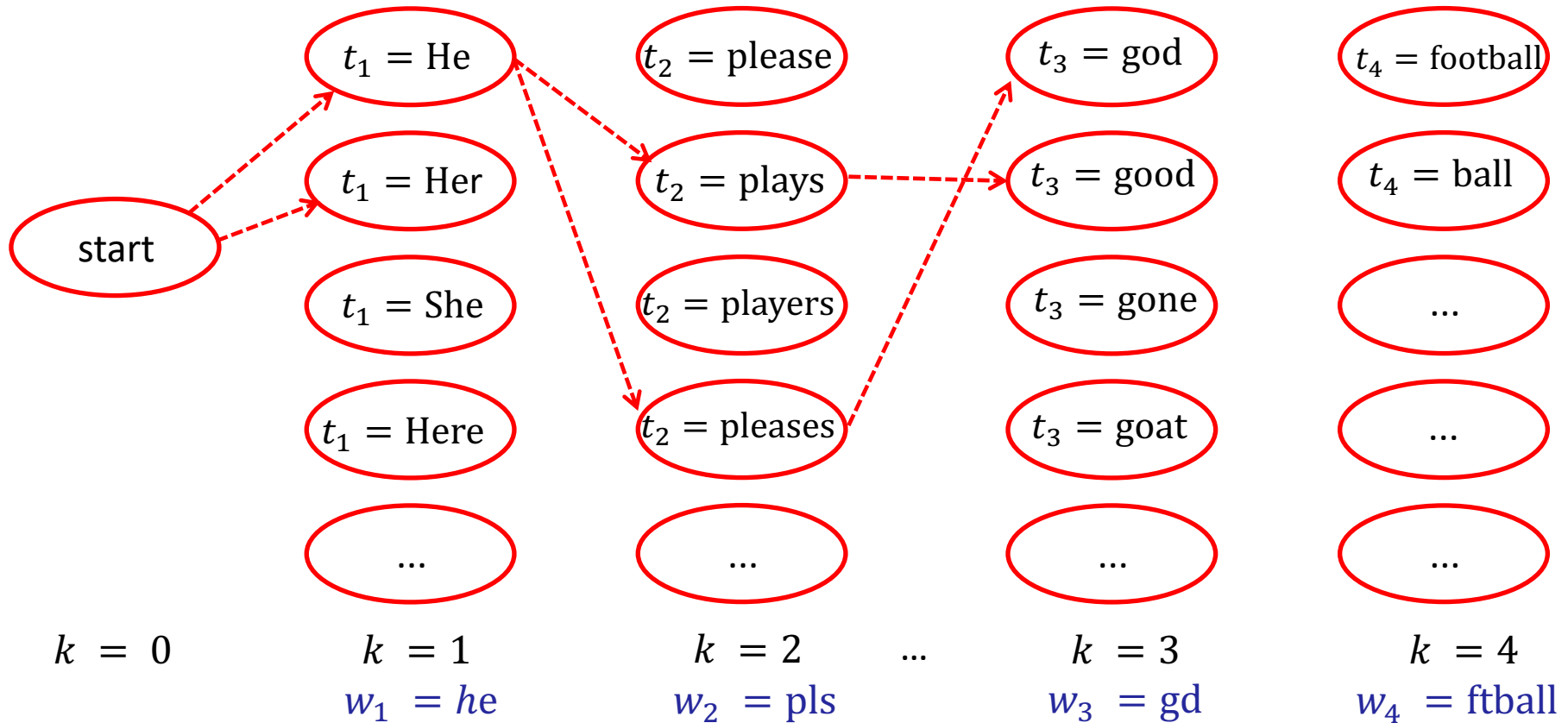


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$
 For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

Beam search decoder



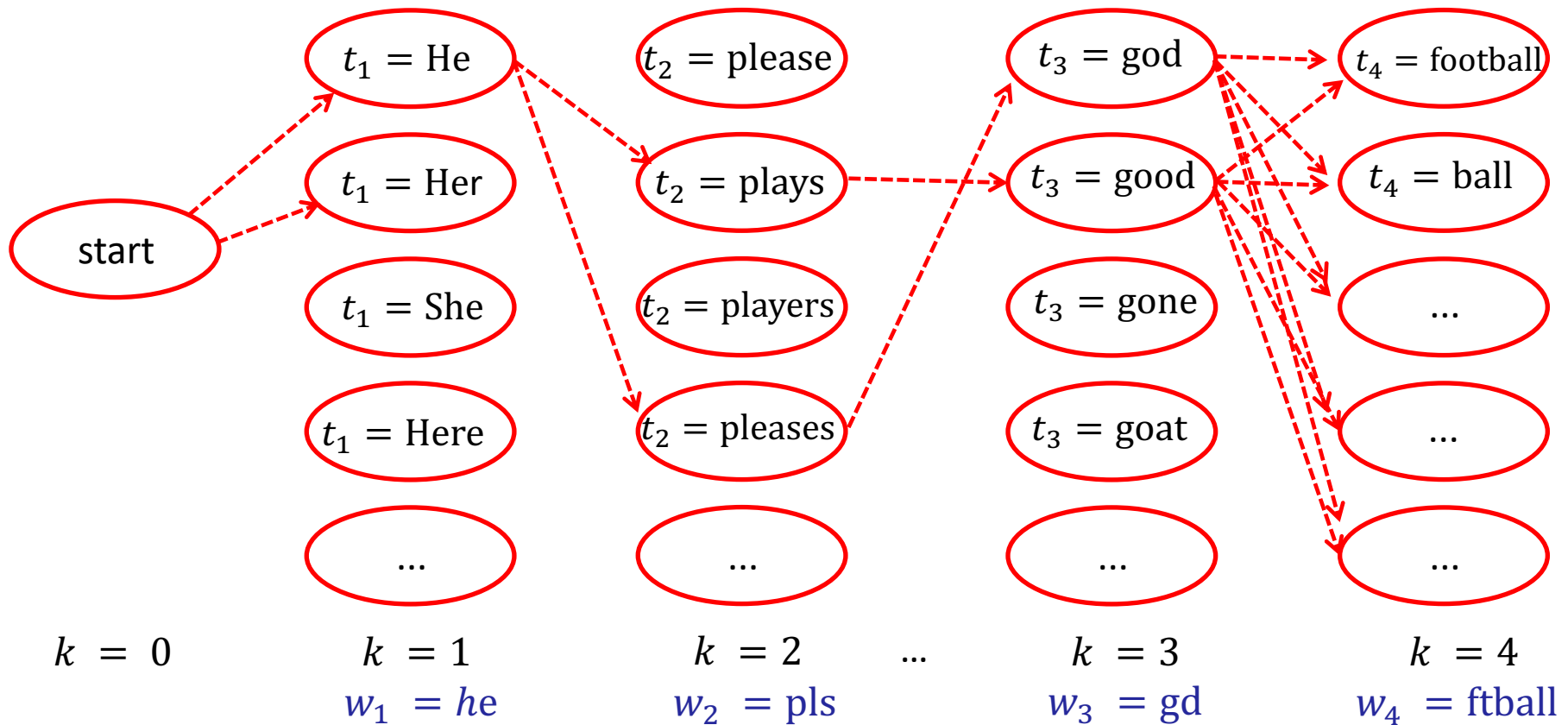
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

Beam search decoder

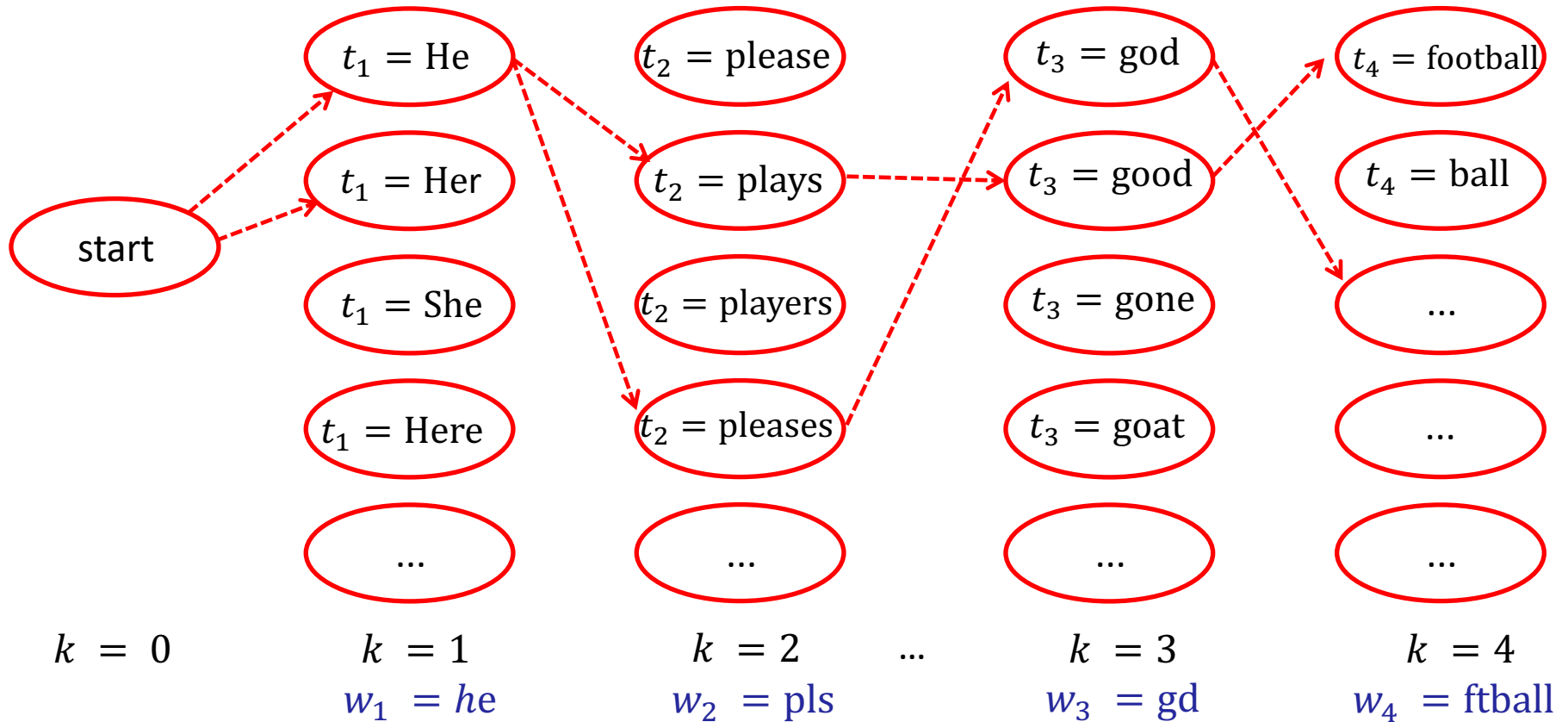


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$
 For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

Beam search decoder



We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^k P(w_i|t_i)$
 For a bigram language model: $\prod_{i=1}^k P(t_i|t_{i-1})$

For each k , we keep the b (here $b = 2$) best paths only.

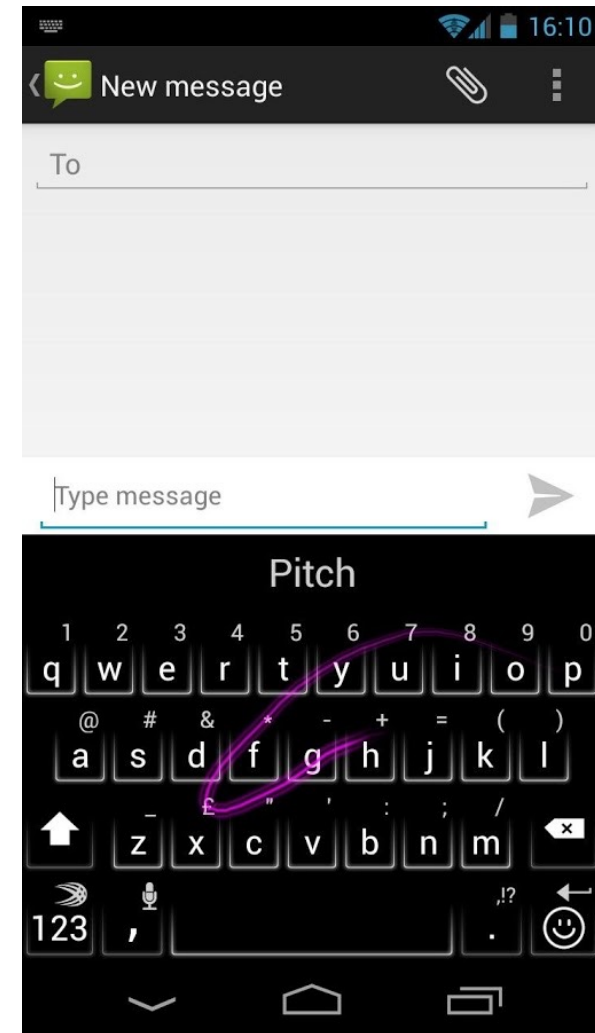
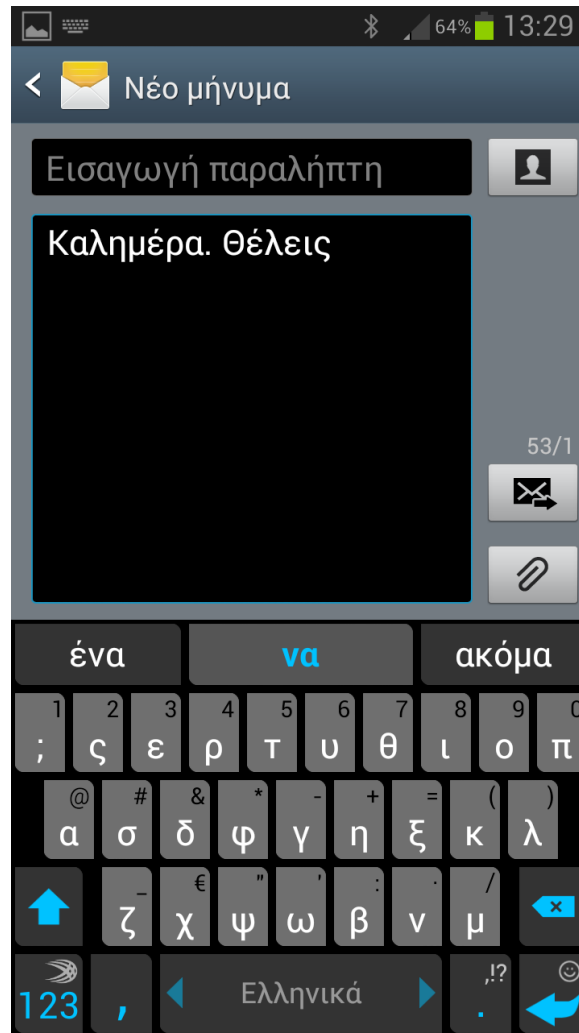
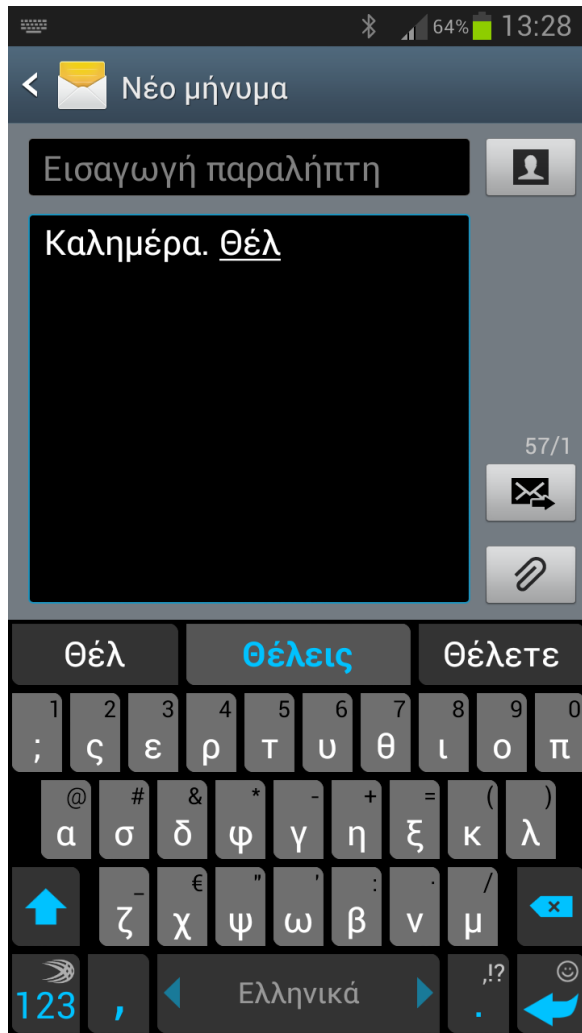
Further decoding notes

- **For n-gram models**, we can use **Viterbi decoding** (a dynamic programming algorithm) to find the **best (guaranteed) path**.
- **Viterbi cannot be used with most neural LMs**.
 - Because of their **infinite history** window.
- **Beam search is still heuristic search**. No guarantee it will find the most probable overall path.
 - But **more likely than greedy decoding**, esp. with large b .
- **Beam search is also common in machine translation**.
 - To be discussed later in the course.
- **Most modern LLMs use just sampling** (or top- k or top- p) with **temperature**.

Further decoding notes (continued)

- **All LMs prefer shorter sequences.**
 - Fewer **probabilities multiplied**, fewer logs added.
- **Problem** when **comparing probabilities** assigned by an LM to sequences of **different lengths**.
 - In the **spelling correction** example, we deliberately assumed that all the hypotheses had the **same length**.
 - **What if** candidate hypotheses have **different lengths**?
 - **Simplest “fix”**: divide the log probability of each candidate sequence by its length m , i.e., $\frac{1}{m} \sum_{i=1}^m \log P(w_i | h_1^{i-1})$.
 - More elaborate fixes also exist...

Smart keyboards



Images from: <http://www.swiftkey.net/>

Evaluating language models

- How do we **evaluate a language model**?
 - Among different language models (e.g., using different n or different smoothing), which one is the best?
 - We typically use **perplexity**, which is based on **entropy**, both discussed next.
 - This is a form of **intrinsic evaluation** (considering a model on its own).
 - One could also use **extrinsic evaluation** (considering the effect on a larger system the model is part of).

Encoding example and entropy

- Let the **possible values** of a random variable C be:
 - c_1 with $P(c_1) = 1/4$, c_2 with $P(c_2) = 1/4$, c_3 with $P(c_3) = 1/2$.
- A good **encoding**:
 - Use **fewer bits** for **more probable values**.
 - $c_1 \rightarrow 10$, $c_2 \rightarrow 11$. We use $-\log_2(1/4) = 2$ bits.
 - $c_3 \rightarrow 0$. We use $-\log_2(1/2) = 1$ bits.
 - Exp. number of transmitted bits: $1/4 \cdot 2 + 1/4 \cdot 2 + 1/2 \cdot 1 = 1.5$
- Information theory says the **ideal encoding** (lowest expected number of transmitted bits) uses **$-\log_2 P(c_i)$ bits for value c_i** .
 - We may need to use a slightly different number of bits in practice, if the $P(c_i)$ probabilities are not powers of 2.
- **With an ideal encoding** (as above), the **expected number of transmitted bits is the Entropy $H(C)$ of C** .
 - It shows **how uncertain we are about the value** of C , i.e., **how much information (in bits) we need to transmit** to let somebody know its value.

Entropy

- **Entropy** of a random variable C :
 - **How uncertain** we are about the **value of C** .
 - **How much information** (in bits, with an ideal encoding) we need to receive **to be certain** about the value of C .
 - What is the **expected number of bits** (with an ideal encoding) that we need to receive **to be certain** about the value of C .

Expected value

$$H(C) = - \sum_{c_i} P(C = c_i) \cdot \log_2 P(C = c_i)$$

Bits used by the
ideal encoding
for each value.

- If C has only two possible values:

$$H(C) = -P(C = 1) \cdot \log_2 P(C = 1) - P(C = 0) \cdot \log_2 P(C = 0)$$

Probabilities estimated from training data.

Example

- Collection of **800 training** e-mail messages.
 - Messages received in the past, manually classified.
 - **200 spam**, **600 ham** (non-spam).
- Estimate the **entropy of C** using the training messages.
 - **$C = 1$ (spam)** ᅇ **$C = 0$ (ham)**.
 - $\log_2 3 = 1.585$
- Repeat when **all the training messages** are in **one category** (**all spam**, or **all ham**).
- Repeat when we have an **equal number** of training messages **per category** (**400 spam**, **400 ham**).

$$H(C) = -P(C = 1) \cdot \log_2 P(C = 1) - P(C = 0) \cdot \log_2 P(C = 0)$$

Cross-entropy

- The **entropy** of a random variable C shows **how uncertain** we are about its value.

$$H(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P(C = c_i)$$

- **How many bits** (expected value) we need to **transmit** (or receive) with an **ideal encoding** to transmit (receive) its value.
- If we use an **encoding** based on **inaccurate probability estimates** P_m instead of the correct probabilities P :

$$H_{P_m}(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P_m(C = c_i) \geq H(C)$$

- We now need to **transmit more bits**, because we don't use an ideal encoding (which uses $-\log_2 P(c_i)$ bits per value).

Cross-entropy – continued

- If we have **two models** $P_{m1}(C)$, $P_{m2}(C)$ both trying to **estimate the correct probabilities** $P(C)$, which one is the **best**?

$$H_{P_{m1}}(C) = -\sum P(C = c_i) \cdot \log_2 P_{m1}(C = c_i) \geq H(C)$$

$$H_{P_{m2}}(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P_{m2}(C = c_i) \geq H(C)$$

- The one with the **smallest cross-entropy**.
- It allows **transmitting** the values of C using **fewer bits**.
- Its **encoding is based on more accurate probability estimates**.
- **Kullback–Leibler divergence** (relative entropy):

$$D_{KL}(P||P_m) = H_{P_m}(C) - H(C) = \sum_{c_i} P(C = c_i) \log_2 \frac{P(C = c_i)}{P_m(C = c_i)}$$

Language entropy

- We now receive **n -tuples of words** of a language **L** . How **uncertain** are we about the next **n -tuple** we'll receive?

$$H(W_1^n | L) = H(W_1, \dots, W_n | L) = - \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **Per-word entropy** (entropy rate):

$$\frac{1}{n} \cdot H(W_1^n | L) = - \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **Entropy of language L :**

$$\begin{aligned} H(L) &= \lim_{n \rightarrow +\infty} \frac{1}{n} H(W_1^n | L) = \\ &= - \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n) \end{aligned}$$

Language entropy – continued

- **Entropy of language L :**

$$H(L) = \lim_{n \rightarrow +\infty} \frac{1}{n} H(W_1^n | L) =$$
$$- \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **It can be proven** (Shannon-McMillan-Breiman) **that:**

$$H(L) = - \lim_{n \rightarrow +\infty} \frac{1}{n} \log_2 P(W_1^n = w_1^n) \simeq - \frac{1}{N} \log_2 P(w_1^N)$$

- In other words, we can use **only one, very long sequence of N words** (a corpus) of the language L .
- More precisely, assuming that the language is not only **stationary** but also **ergodic**.

Language cross-entropy

- If we have **two language models** P_{m1} , P_{m2} for a language L , which one is the **best**?

$$H_{P_{m1}}(L) = - \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P_{m1}(W_1^n = w_1^n)$$

$$= - \lim_{n \rightarrow +\infty} \frac{1}{n} \log_2 P_{m1}(w_1^n) \simeq - \frac{1}{N} \log_2 P_{m1}(w_1^N)$$

$$H_{P_{m2}}(L) = - \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P_{m2}(W_1^n = w_1^n)$$

$$= - \lim_{n \rightarrow +\infty} \frac{1}{n} \log_2 P_{m2}(w_1^n) \simeq - \frac{1}{N} \log_2 P_{m2}(w_1^N)$$

- The model with the **smallest language cross-entropy**. In effect, the model that assigns the **largest probability** to the **test corpus**.

Cross-Entropy and Perplexity

- For example, if a **bigram language model** is used on a test corpus of N tokens:

$$\begin{aligned} -\frac{1}{N} \log_2 P_m(w_1^N) &= -\frac{1}{N} \log_2 [P_m(w_1 | start) P_m(w_2 | w_1) \dots] \\ &= -\frac{1}{N} [\log_2 P_m(w_1 | start) + \log_2 P_m(w_2 | w_1) + \dots] \end{aligned}$$

- Usually **perplexity** scores are published:

$$\text{Perplexity} = 2^{H_{P_m}(L)} \cong 2^{-\frac{1}{N} \log_2 P_m(w_1^N)}$$

- The **lower** the perplexity, the **better** the model.
 - Alternative interpretation: a LM with **perplexity** r has the **same perplexity** (and cross-entropy) as a LM that always returns (when predicting the next word) a **uniform probability distribution over a vocabulary of r words** (see self-study exercise for details and proof).

Comparison 1-4-Gram

$$-\log P(w_i|w_{i-1})$$

$$-\log P(w_i|w_{i-2}, w_{i-1})$$

word	unigram	bigram	trigram	4-gram
i	6.684	3.197	3.197	3.197
would	8.342	2.884	2.791	2.791
like	9.129	2.026	1.031	1.290
to	5.081	0.402	0.144	0.113
commend	15.487	12.335	8.794	8.633
the	3.885	1.402	1.084	0.880
rapporteur	10.840	7.319	2.763	2.350
on	6.765	4.140	4.150	1.862
his	10.678	7.316	2.367	1.978
work	9.993	4.816	3.498	2.394
.	4.896	3.020	1.785	1.510
</s>	4.828	0.005	0.000	0.000
average	8.051	4.072	2.634	2.251
perplexity	265.136	16.817	6.206	4.758

$$-\frac{1}{N} \sum_{i=1}^N \log P(w_i|w_{i-2}, w_{i-1})$$

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

From P. Blunsom's presentation "From Language Modelling to Machine Translation"

http://videolectures.net/deeplearning2015_blunsom_machine_translation/

Training, development, test data

- **Training data:**
 - Used to **estimate** (learn) the **probabilities of n -grams**.
 - More generally, we **train our model** on these data.
- **Development data:**
 - Used to **select models** (e.g., 2-gram or 3-gram LM), **tune hyper-parameters** (e.g., λ of interpolated LMs), **select best training epochs** (in neural networks) etc.
 - **If we make these choices by evaluating on test data, we indirectly train our model on the test dataset!**
- **Test data:**
 - Used for the **final evaluation of our model**, to see how well it performs on **unseen data**.

Training, development, test data

n-gram probability
estimates (more generally,
learning algorithm applied
to this subset)

training data

perplexity (or
other score,
for hyper-
parameter
tuning etc.)

development data

final perplexity
(or other score, to
check the
performance on
unseen data)

test data

- In competitions, the **test data may not be publicly available**.
- We may have to use the **development data as test data**.
- A **small subset** of the **training data** may have to be “**held out**” (e.g., for hyper-parameter tuning).
 - Often called “**validation data**”.
 - This **reduces** the size of the **training set**.
 - And a **small held-out set, may not be representative**.

Cross-validation

- **Instead of holding out data** from the training data:
 - Divide the training data into **n parts** (e.g., 5), often **preserving class ratios** (e.g., positives/negatives) in all parts (“stratified”).
 - Perform **n iterations** (folds) to obtain a **score** (e.g., accuracy) for a particular **combination of hyper-parameter values**.
 - In each iteration, use a **different part as validation data** (to obtain a score) and the **other $n - 1$ parts as training data**.
 - **Average** the score (e.g., accuracy) **over iterations** to obtain a score for the particular combination of hyper-parameter values.



Sub-word tokenizers

- We have a **fixed-size vocabulary** containing the most **common words and sub-words**, including **characters**.
 - **No unknown words**. Any word not in the vocabulary, is **broken into sub-words** of the vocabulary. E.g., “decentralize” may be broken into “de”, “cent”, “r”, “al”, “ize”. **Sub-words do not necessarily correspond to morphemes or syllables**.
- We’ll consider **Byte-Pair Encoding (BPE)**.
 - **Several other** widely used **sub-word tokenizers**.
 - E.g., **Unigram, SentencePiece, WordPiece**.
 - <https://huggingface.co/learn/llm-course/chapter6/1>

BPE training example (from J&M)

set_new_new_renew_reset_renew

corpus

2 _ n e w

2 _ r e n e w

1 s e t

1 _ r e s e t

vocabulary

_ , e , n , r , s , t , w

corpus

2 _ n e w

2 _ r e n e w

1 s e t

1 _ r e s e t

vocabulary

_ , e , n , r , s , t , w , n e

BPE training example (from J&M)

corpus

2 _ new
2 _ r e new
1 s e t
1 _ r e s e t

vocabulary

_ , e , n , r , s , t , w , ne , new

corpus

2 _ new
2 _re new
1 s e t
1 _re s e t

vocabulary

_ , e , n , r , s , t , w , ne , new , _r , _re

merge

current vocabulary

(_, new)	_ , e , n , r , s , t , w , ne , new , _r , _re , _new
(_re, new)	_ , e , n , r , s , t , w , ne , new , _r , _re , _new , _renew
(s, e)	_ , e , n , r , s , t , w , ne , new , _r , _re , _new , _renew , se
(se, t)	_ , e , n , r , s , t , w , ne , new , _r , _re , _new , _renew , se , set

From D. Jurafsky and J.H. Martin, “Speech and Language Processing”, 3rd Ed.
(draft August 2026), <https://web.stanford.edu/~jurafsky/slp3/>.

BPE *training* stage

- Given a **training corpus** (C) and a target **vocabulary size**, it **learns the vocabulary** (V).
- **Initial vocabulary**: all the characters (e.g., $V = \{a, b, c, d\}$).
- Find the **most frequent pair of characters** in C (e.g., $[b, c]$), add it to V as new sub-word ($V = \{a, b, c, d, bc\}$), replace all occurrences of the pair in C by the new sub-word.
- Find the **most frequent pair of vocabulary items** in C (e.g., $[d, bc]$), add it to the vocabulary ($V = \{a, b, c, d, bc, dbc\}$), replace all occurrences of the pair in C by the new sub-word.
- Repeat **until we reach** the target **vocabulary size**.
- The **training corpus** is usually **pre-separated** in words (e.g., at white spaces, punctuation) and we look for frequent pairs inside the pre-separated words only (**we don't cross word boundaries**).

BPE *encoding* stage

- Input: a **learned vocabulary** (V), the **order of the learned merges** of the training phase ([“n”+“e”, “ne”+“w”, “_”+ “r”, “_r”+“e”, ...]), **input sequence of characters to tokenize**.
- **Pre-separates the input**, as in the training stage (e.g., at white spaces, punctuation).
- **Greedy applies the learned merges**, from **most to least frequent** (by frequency order of the **training set**), to each pre-separated string of the input.
- E.g., “_reset” may become “_re”, “set”.

BPE is often applied to **Unicode bytes**.

In **multilingual settings**, V is often **dominated by English words and sub-words**. Words of **other languages** may end up being segmented to **very small sub-words or characters**.

Further issues with n-gram LMs

- What about similar words?

- she *bought* a bicycle
- she *purchased* a bicycle



cannot share strength
among similar words

- Long-distance dependencies?

- for *programming* she yesterday purchased her own brand new *laptop*
- for *running* she yesterday purchased her brand new *sportswatch*



cannot handle long-distance
dependencies

Adapted from Graham Neubig

Additional optional study slides

Kneser-Ney smoothing

- E.g., for bigrams w_{k-1}, w_k :

✓ green, apple

✓ green, paper

✓ green, book

✓ ...

✓ ...

× green, mouse

× green, cyclotron

× green, York

× ...

Encountered in the corpus, i.e., $c(w_{k-1}, w_k) > 0$. Steal probability mass from each estimate $\frac{c(w_{k-1}, w_k)}{c(w_{k-1})}$, i.e., use $\frac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}$, where D is constant.

Not encountered in the corpus, i.e., $c(w_{k-1}, w_k) = 0$. Distribute to them the probability mass stolen from all the encountered w_{k-1}, w_k that had the same w_{k-1} (“green”). Distribute proportionally to $P(w_k)$ (e.g., “mouse” is more frequent than “cyclotron”).

Kneser-Ney smoothing

- Formula for ideas of previous slide (D is constant):

$$P_{KN}(w_k \mid w_{k-1}) = \begin{cases} \frac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & \text{if } c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot P(w_k), & \text{else} \end{cases}$$

- α values needed to ensure that probabilities sum up to 1.

Improved K-N smoothing

- Instead of $P(w_k)$, distribute the stolen probability mass proportionally to:

$$Prev(w_k) = \frac{prev(w_k)}{\sum_{v \in V : c(w_{k-1}, v) = 0} prev(v)}$$

The denominator ensures that the $Prev(w_k)$ scores of all the words w_k that need to receive stolen probability mass **sum to 1**.

where:

$$prev(w_k) = |\{w \in V : c(w, w_k) > 0\}|$$

How many vocabulary (distinct) words occur immediately before w_k in the corpus. E.g., “York” may occur almost always after “New”; hence “green York” should not be given much of the probability mass stolen from the encountered bigrams that start with “green”.

Improved K-N smoothing

$$P_{KN}(w_k \mid w_{k-1}) = \begin{cases} \frac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & \text{if } c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot \text{Prev}(w_k), & \text{else} \end{cases}$$

$$a(w_{k-1}) = \frac{D}{c(w_{k-1})} \cdot |\{w \in V : c(w_{k-1}, w) > 0\}|$$

Total probability mass stolen from bigrams that start with w_{k-1}
(w_{k-1} = “green” in our example).

Katz backoff

- Consult an n -gram model with a **smaller n , whenever necessary**. For example, when using a trigram model:

$$P_{Katz}(w_k \mid w_{k-2}, w_{k-1}) = \begin{cases} P(w_k \mid w_{k-2}, w_{k-1}), & \text{if } c(w_{k-2}^k) > 0 \\ \alpha(w_{k-2}, w_{k-1}) \cdot P_{Katz}(w_k \mid w_{k-1}), & \text{else} \end{cases}$$

$$P_{Katz}(w_k \mid w_{k-1}) = \begin{cases} P(w_k \mid w_{k-1}), & \text{if } c(w_{k-1}, w_k) > 0 \\ \alpha(w_{k-1}) \cdot P(w_k), & \text{else} \end{cases}$$

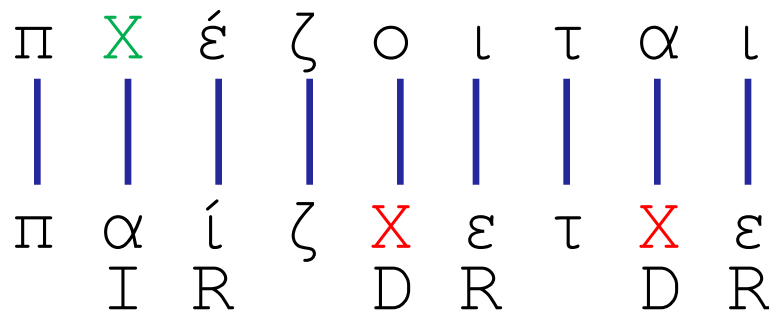
- The α values are needed to ensure that:

$$\sum_{w_k \in V} P_{Katz}(w_k \mid w_{k-2}, w_{k-1}) = 1 \quad \sum_{w_k \in V} P_{Katz}(w_k \mid w_{k-1}) = 1$$

- Consult the book of Jurafsky & Martin for formulae to compute the α values and (many) other smoothing methods.

Edit distance

- Input: **two strings** (e.g., words from **tweet** and **dictionary**).
- What is the **total minimum cost** to **convert one input string to the other**, using particular operators?
- **Levenshtein** distance (one possible edit distance):
 - Operators: **insert** (I, cost **1**), **delete** (D, cost **1**), **replace** (R, cost **2**). Other work may set the cost of R to 1.
- When converting from **Greeklish to Greek**, we may want to set, for example, $R(e, \varepsilon) < R(e, \alpha)$.



We also get an **alignment** of characters. Similarly, we can compute the edit distance and alignment of the **words of two sentences**, by applying I, D, R to words instead of characters.

Computing Levenshtein distance

- How can we **convert**:

$\pi\acute{\epsilon}\zeta\omicron\iota$ to $\pi\alpha\acute{\iota}\zeta\omega$

based on shorter (by one final letter) forms of
 $\pi\acute{\epsilon}\zeta\omicron\iota$ and/or $\pi\alpha\acute{\iota}\zeta\omega$?

- 1st way: Delete the last letter of $\pi\acute{\epsilon}\zeta\omicron\iota$ and convert $\pi\acute{\epsilon}\zeta\omicron$ to $\pi\alpha\acute{\iota}\zeta\omega$.

$\pi\acute{\epsilon}\zeta\omicron \setminus \rightarrow \pi\alpha\acute{\iota}\zeta\omega$

$\text{Del}(\iota) + \text{cost}(\pi\acute{\epsilon}\zeta\omicron, \pi\alpha\acute{\iota}\zeta\omega)$

- 2nd way: Convert $\pi\acute{\epsilon}\zeta\omicron\iota$ to $\pi\alpha\acute{\iota}\zeta$ and add ω to the end of $\pi\alpha\acute{\iota}\zeta$.

$\pi\acute{\epsilon}\zeta\omicron\iota \rightarrow \pi\alpha\acute{\iota}\zeta \textcircled{\omega}$

$\text{cost}(\pi\acute{\epsilon}\zeta\omicron\iota, \pi\alpha\acute{\iota}\zeta) + \text{Ins}(\omega)$

Computing Levenshtein distance (II)

- 3rd way: Convert $\pi\acute{\epsilon}\zeta o$ to $\pi\alpha\acute{\iota}\zeta$ and replace ι by ω .

$$\pi\acute{\epsilon}\zeta o \textcircled{\iota} \rightarrow \pi\alpha\acute{\iota}\zeta \textcircled{\omega}$$

$$\text{cost}(\pi\acute{\epsilon}\zeta o, \pi\alpha\acute{\iota}\zeta) + \text{Rep}(\iota, \omega)$$

- **Which way is the best?**
 - The one with the **smallest cost**.
 - **At each step**, we consider all three ways and we select the **cheapest one (Ins, Del, or Rep)**.

Computing Levenshtein distance

	#	π	α	$\acute{\imath}$	ζ	ϵ	τ	ϵ
#	0	1	2	3	4	5	6	7
π	1							
$\acute{\epsilon}$	2							
ζ	3							
$\omicron$	4							
ι	5							
τ	6							
α	7							
ι	8							

↑ Del (+1)
← Ins (+1)

What is the (min) cost to convert “#” to “# $\pi\alpha\acute{\imath}$ ”?

Cost to convert “#” to “# $\pi\alpha$ ” and insert “ $\acute{\imath}$ ”.

What is the (min) cost to convert “# $\pi\acute{\epsilon}\zeta$ ” to “#”?

Cost to delete « ζ » and convert “# $\pi\acute{\epsilon}$ ” to “#”.

Computing Levenshtein distance

	#	π	α	$\acute{\imath}$	ζ	ϵ	τ	ϵ
#	0	1	2	3	4	5	6	7
π	1	0						
$\acute{\epsilon}$	2							
ζ	3							
$\omicron$	4							
ι	5							
τ	6							
α	7							
ι	8							

↑ Del (+1)

← Ins (+1)

↖ Rep (+2, or 0 for same letter)

What is the (min) cost to convert “# π ” to “# π ”?

1st way: **Delete** the “ π ” of the first (vertical) string and **convert** the remaining string “#” to “# π ”.

2nd way: **Convert** the “# π ” of the first string to “#” and **add** “ π ” to the resulting string.

3rd way: **Convert** the “#” of the first string to “#” and replace the “ π ” of the first string by “ π ” (same letter).

Computing Levenshtein distance

	#	π	α	$\acute{\imath}$	ζ	ϵ	τ	ϵ
#	0	1	2	3	4	5	6	7
π	1	0	1					
$\acute{\epsilon}$	2							
ζ	3							
$\omicron$	4							
ι	5							
τ	6							
α	7							
ι	8							

↑ Del (+1)
← Ins (+1)
↖ Rep (+2, or 0 for same letter)

2+1=3
 1+2=3
 0+1=1

What is the (min) cost to convert “# π ” to “# $\pi\alpha$ ”;

1st way: **Delete** the “ π ” of the first (vertical) string and **convert** the remaining string “#” to “# $\pi\alpha$ ”.

2nd way: **Convert** the “# π ” of the first string to “# π ” (no change) and **add** “ α ” to the resulting string.

3rd way: **Convert** the “#” of the first string to “# π ” and **replace** the “ π ” of the first string by “ α ”.

Computing Levenshtein distance

	#	π	α	í	ζ	ε	τ	ε
#	0	1	2	3	4	5	6	7
π	1	0	1	2	3	4	5	6
é	2	1	2	3	4	3	4	5
ζ	3	2	3	4	3	4	5	6
o	4	3	4	5	4	5	6	7
ι	5	4	5	4	5	6	7	8
τ	6	5	6	5	6	7	6	7
α	7	6	5	6	7	8	7	8
ι	8	7	6	5	6	7	8	9

↑ Del (+1)

← Ins (+1)

↖ Rep (+2, or 0 for same letter)

Shaded cells show one of the possible alignments.

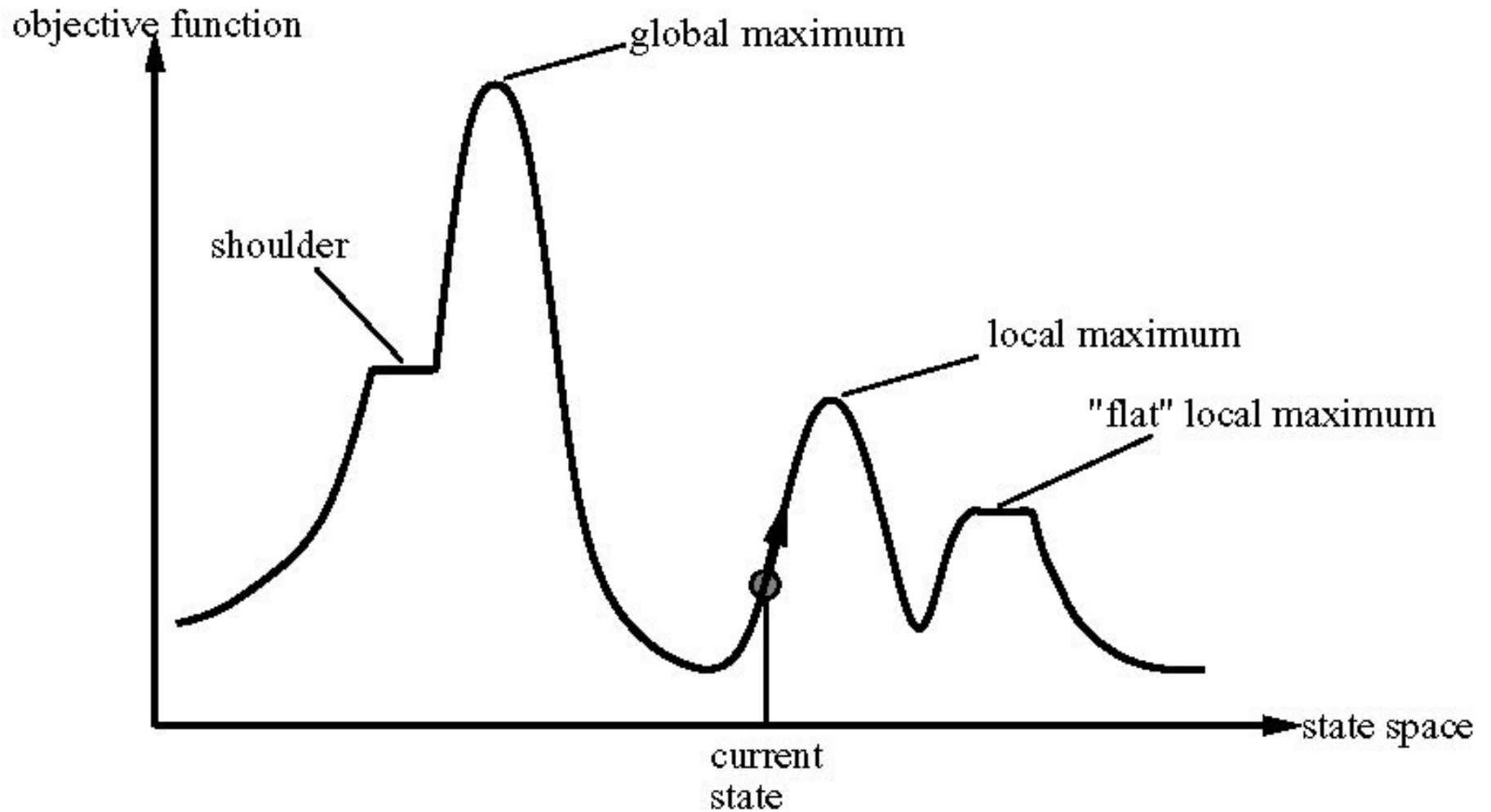
For each cell, the outgoing arrows point to the neighbor(s) the cell's (best) value is based on.

Hill climbing search (HC)

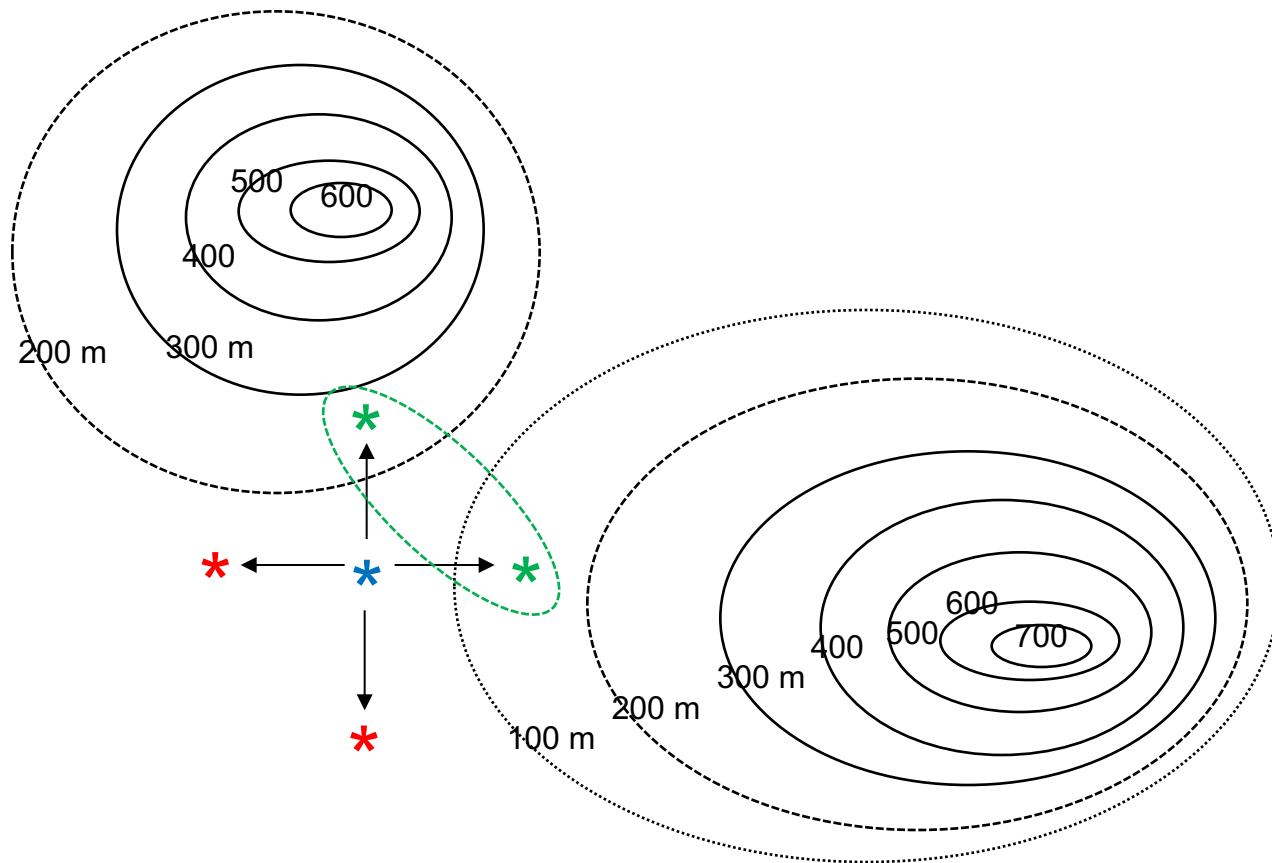
1. Make the **initial state** the **current state**.
2. **Generate** and **assess** the **children-states** of the current state.
3. If **no child-state is better** than the current state, **return the current state**.
4. Make the **best child-state** the **current state**.
5. Go to **step 2**.

Spoiler alert: Most **neural networks** are also trained using a **kind of HC** (**SGD**, stochastic gradient descent), where the **state contains the weights** of the network.

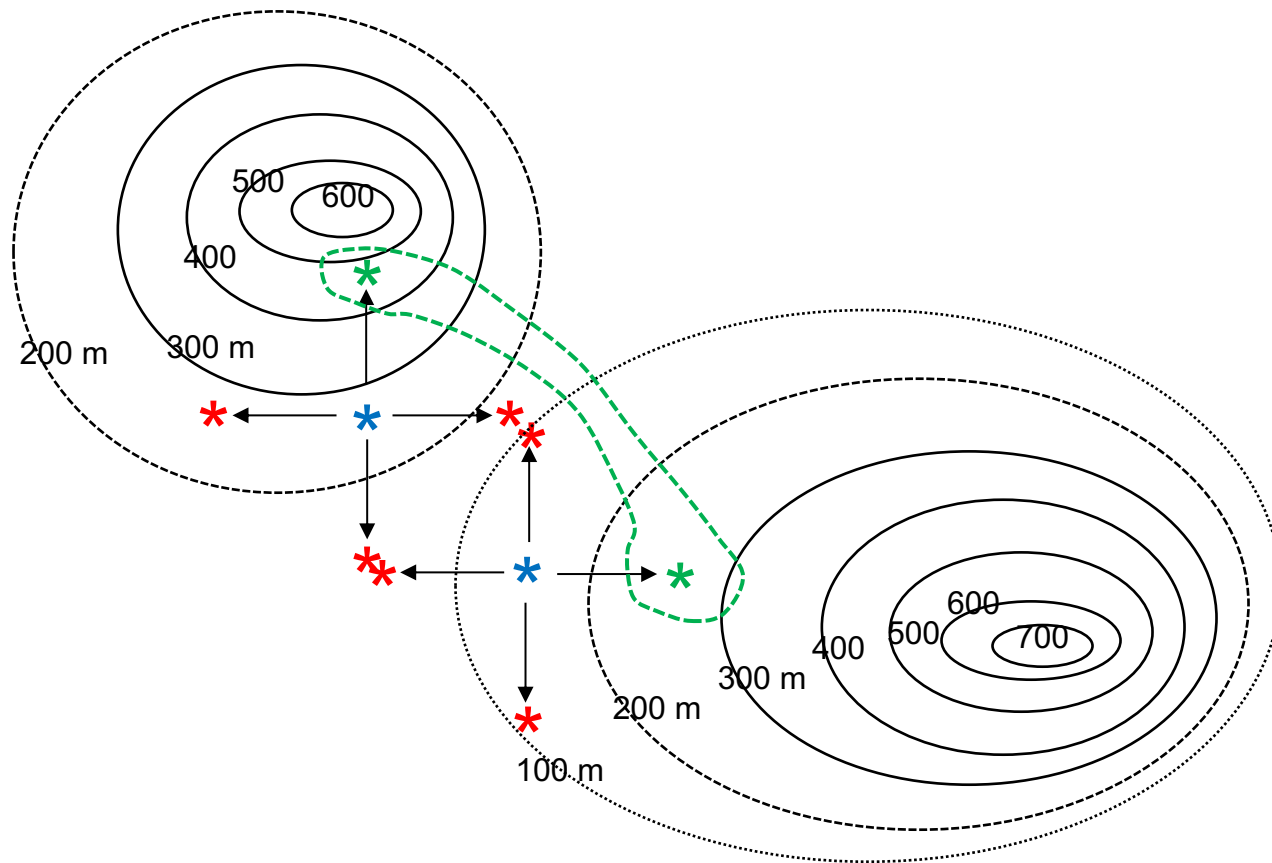
Hill climbing



(Local) Beam search



(Local) Beam search



(Local) Beam search

- Keep **k states** in the **search frontier**.
 - Initially k random states.
- At each step, **produce** and **assess** the **children-states of the k states** in the frontier.
 - If a final state criterion exists and we reach a final state, stop.
- **Keep the k best of the children-states** and repeat.
 - Until we exceed a maximum number of iterations.
- **Random restarts:** we often **repeat the search** several times, starting from **different initial k states**
 - In spelling correction decoding, there is only one initial state.

Recommended reading

- Jurafsky & Martin (2nd ed.): chapter 4 (not sections 4.5.2, 4.5.3, 4.7.1, 4.9.2), sections 3.10, 3.11, 5.9.
 - Available at AUEB's library.
 - See also the free draft of the 3rd edition:
<http://web.stanford.edu/~jurafsky/slp3/>
- For more information, consult chapters 2 and 6 of Manning & Schütze's book *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
 - Available at AUEB's library.
 - Chapter 2 introduces basic concepts of probability theory, entropy, the noisy channel etc.
 - Chapter 6 covers n -gram language models.

