midterm review

CS 585, Fall 2018

Introduction to Natural Language Processing

http://people.cs.umass.edu/~miyyer/cs585/

Mohit Iyyer

College of Information and Computer Sciences

University of Massachusetts Amherst

questions from last time…

- don't make the HWs harder! please make the HWs harder!
- can you go over HMMs / Viterbi? x5
- what's the purpose of the end symbol in language models and neural MT?
- do we need to do the optional reading?
- cheat sheet????

midterm details

- 8.5 x 11 cheat sheet allowed, both sides, *hand-written* only. bring calculator!
- breakdown:
  - 20% text classification (NB, LR, NN)
  - 20% language modeling
  - 20% POS tagging / HMMs
  - 20% word embeddings
  - 20% machine translation

text classification

f can be hand-designed rules

- if “won $10,000,000” in **x**, **y** = **spam**
- if “CS585 Fall 2018” in **x**, **y** = **not spam**

what are the drawbacks of this method?

naive Bayes

- represents input text as a bag of words
- what's the independence assumption???
- given labeled data, we can use naive Bayes to estimate probabilities for unlabeled data
- **goal:** infer probability distribution that generated the labeled data for each label

class conditional probabilities

Bayes rule (ex: *x* = sentence, *y* = label in {pos, neg})

$$\underbrace{p(y \mid x)}_{\text{posterior}} = \frac{\overbrace{p(y)}^{\text{prior}} \cdot \overbrace{P(x \mid y)}^{\text{likelihood}}}{p(x)}$$

our predicted label is the one with the highest posterior probability, i.e.,

$$\hat{y} = \arg\max_{y \in Y} p(y) \cdot P(x \mid y)$$

remember the independence assumption!

maximum a posteriori (MAP) class

$$\hat{y} = \arg\max_{y \in Y} p(y) \cdot P(x \mid y)$$

$$= \arg\max_{y \in Y} p(y) \cdot \prod_{w \in x} P(w \mid y)$$

$$= \arg\max_{y \in Y} \log p(y) + \sum_{w \in x} \log P(w \mid y)$$

computing the prior…

- i hate the movie
- i love the movie
- i hate the actor
- the movie i love
- i love love love love love the movie
- hate movie
- i hate the actor i love the movie

$p(y)$ lets us encode inductive bias about the labels

we can estimate it from the data by simply counting…

label y	count	p(Y=y)	log(p(Y=y))
positive	3	0.43	-0.84
negative	4	0.57	-0.56

computing the likelihood…

p(X | y=positive)

word	count	p(w l y)
i	3	0.19
hate	0	0.00
love	7	0.44
the	3	0.19
movie	3	0.19
actor	0	0.00
total	**16**	

p(X | y=negative)

word	count	p(w l y)
i	4	0.22
hate	4	0.22
love	1	0.06
the	4	0.22
movie	3	0.17
actor	2	0.11
total	**18**	

posterior probs for X_{new}

$$p(y \mid x) \propto \arg\max_{y \in Y} p(y) \cdot P(X_{\text{new}} \mid y)$$

$$\log p(\text{positive} \mid X_{\text{new}}) \propto \log P(\text{positive}) + \log p(X_{\text{new}} \mid \text{positive})$$
$$= -0.84 - 4.96 = -5.80$$

$$\log p(\text{negative} \mid X_{\text{new}}) \propto -0.56 - 8.91 = -9.47$$

Naive Bayes predicts a positive label!

Laplace (add-1) smoothing for Naïve Bayes

$$\hat{P}(w_i \mid c) = \frac{count(w_i, c)}{\sum_{w \in V} \left(count(w, c) \right)}$$

$$= \frac{count(w_i, c) + 1}{\left(\sum_{w \in V} count(w, c) \right) + |V|}$$

what happens if we do
add-*n* smoothing as *n* increases?

Features

- Input document **d** (a string...)
- Engineer a <u>feature function</u>, f(d), to generate <u>feature vector</u> ***x***

$$f(d) \longrightarrow x$$

$$f(d) = \begin{pmatrix} \text{Count of "happy",} \\ \text{(Count of "happy") / (Length of doc),} \\ \text{log(1 + count of "happy"),} \\ \text{Count of "not happy",} \\ \text{Count of words in my pre-specified} \\ \text{word list, "positive words according} \\ \text{to my favorite psychological theory",} \\ \text{Count of "of the",} \\ \text{Length of document,} \\ \ldots \end{pmatrix}$$

Typically these use <u>feature templates</u>:
Generate many features at once

for each word w:
- ${w}_count
- ${w}_log_1_plus_count
- ${w}_with_NOT_before_it_count
- ....

- Not just word counts. Anything that might be useful!
- **Feature engineering**: when you spend a lot of time trying and testing new features. Very important!!! This is a place to put linguistics in.

step 1: featurization

1. Given an input text **X**, compute feature vector **x**

$$\mathbf{x} = \langle \textit{count}(\text{nigerian}), \textit{count}(\text{prince}), \textit{count}(\text{nigerian prince}) \rangle$$

step 2: dot product w/ weights

1. Given an input text **X**, compute feature vector **x**

x = < *count*(nigerian), *count*(prince), *count*(nigerian prince) >

2. Take dot product of **x** with weights **β** to get **z**

β = < -1, -1, 4>

$$z = \sum_{i=0}^{|X|} \beta_i x_i$$

step 3: compute class probability

1. Given an input text **X**, compute feature vector **x**

x = < *count*(nigerian), *count*(prince), *count*(nigerian prince) >

2. Take dot product of **x** with weights **β** to get **z**

β = < -1, -1, 4>

$$z = \sum_{i=0}^{|X|} \beta_i x_i$$

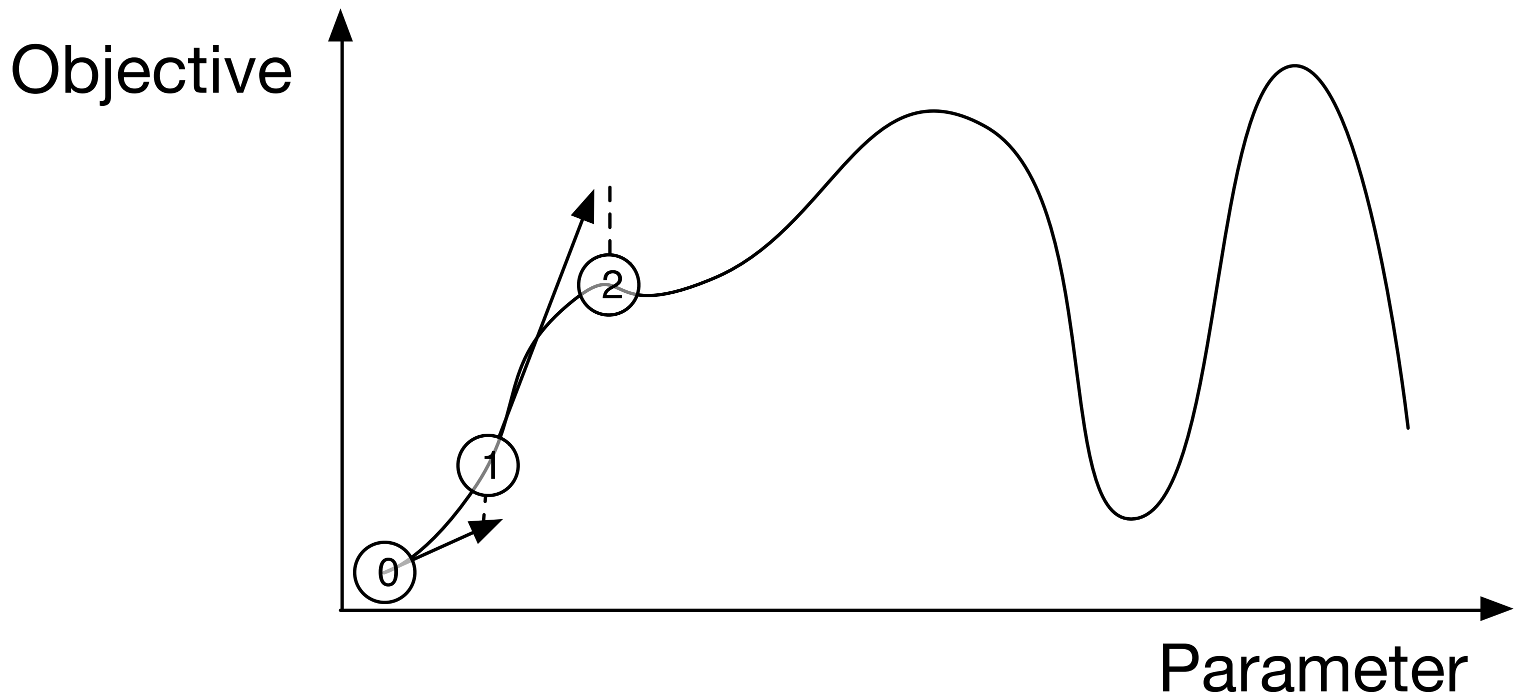
3. Apply logistic function to **z**

$$P(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}}$$

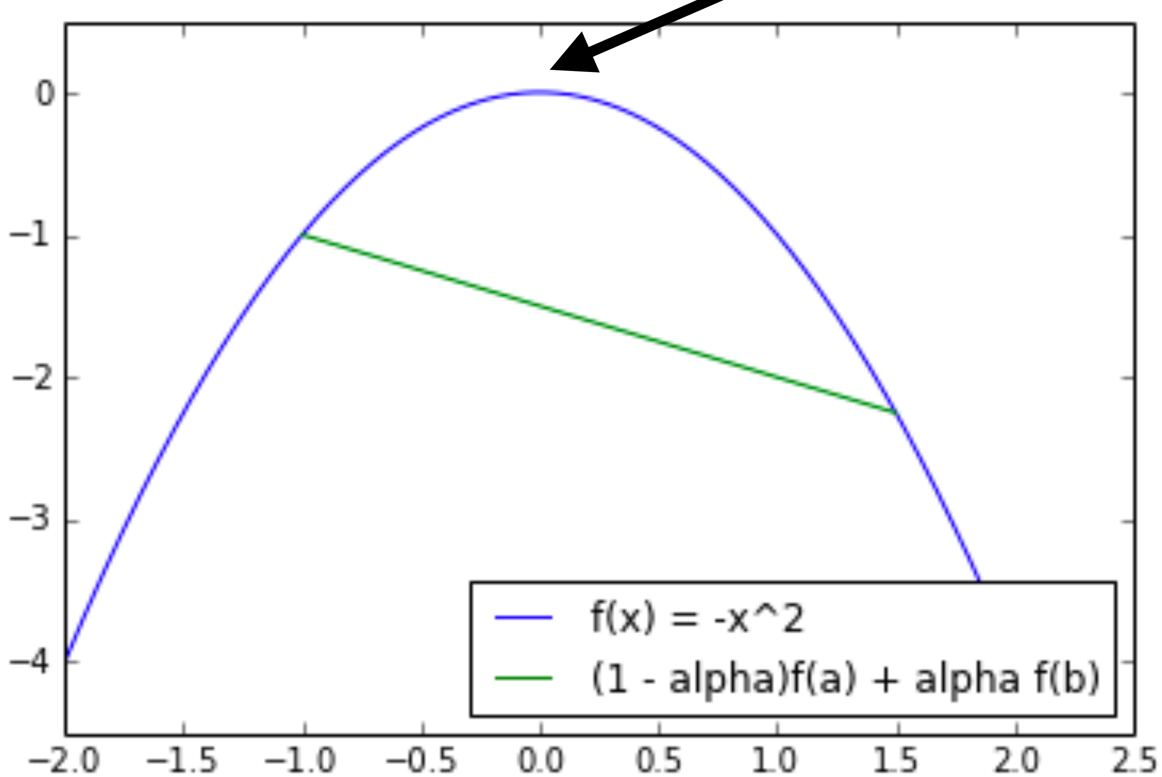
gradient ascent (non-convex)

Gradient Descent (non-convex)

Goal
Optimize log likelihood with respect to variables β

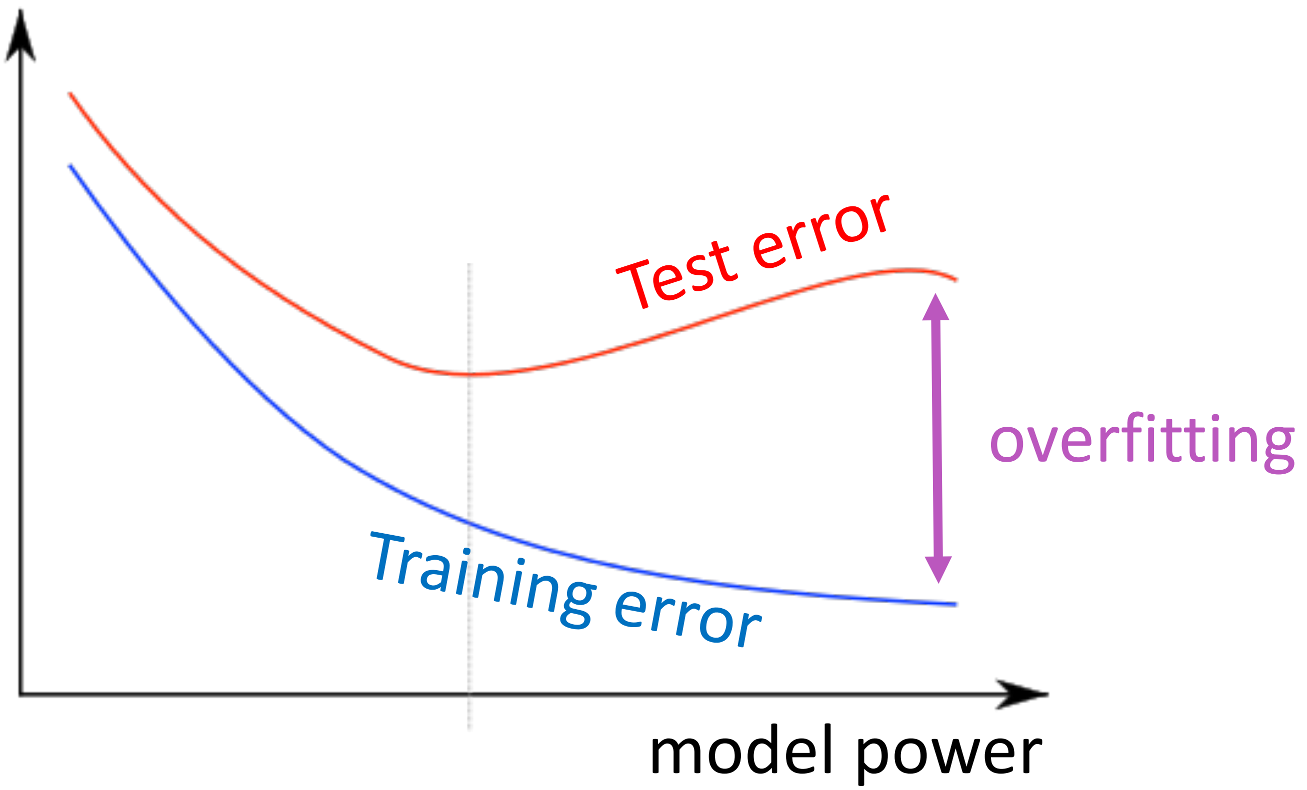


good news! the log-likelihood in LR is *concave*, which means that it has just one local (and global) maximum



Regularization

- Regularization prevents overfitting when we have a lot of features (or later a very powerful/deep model,++)



L2 regularization

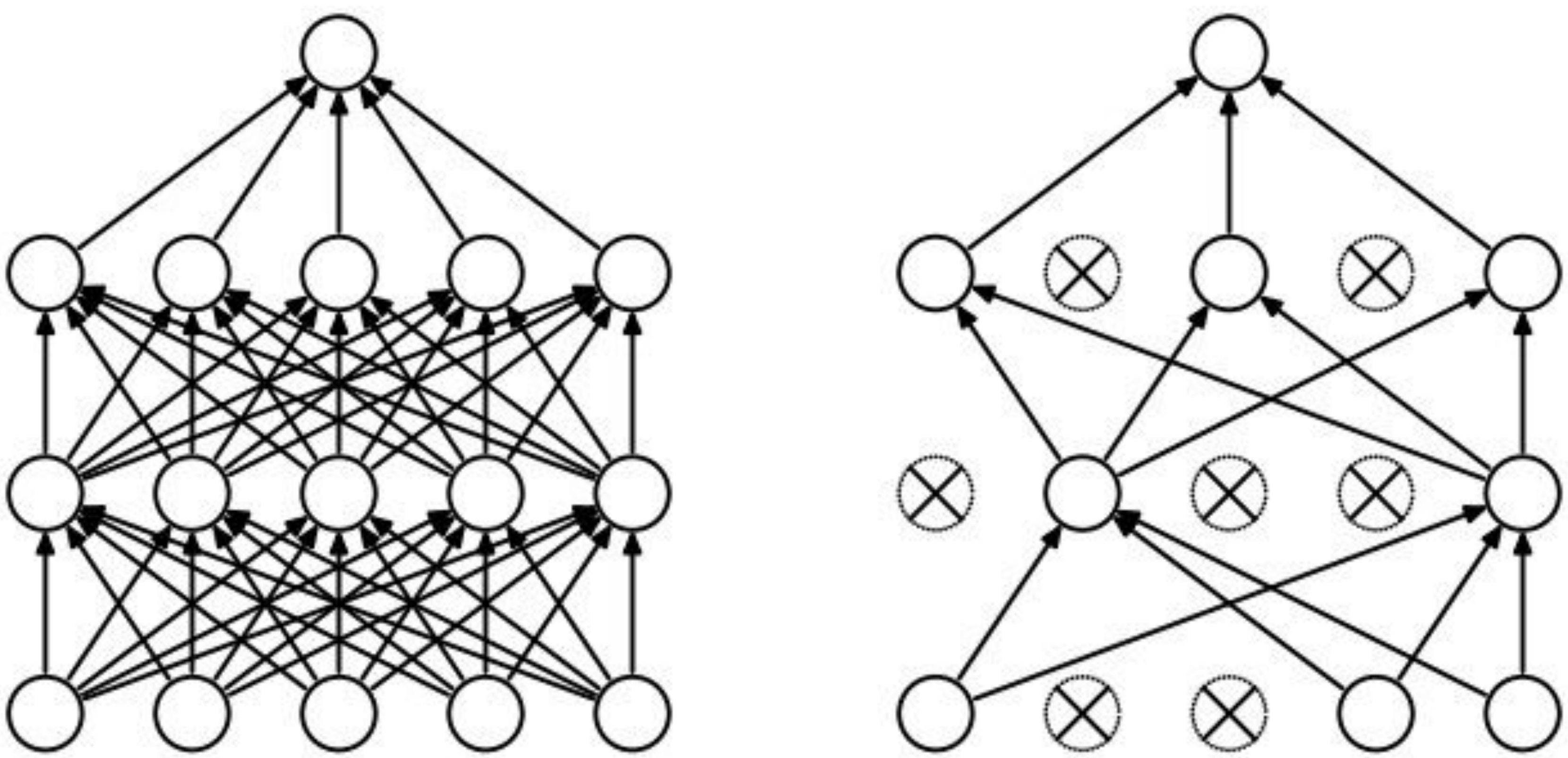
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) \boxed{+ \lambda \sum_k \theta_k^2}$$

θ represents all of the model's parameters!

penalizing their norm leads to smaller weights >
we are constraining the parameter space >
we are putting a prior on our model

dropout (for neural networks)

randomly set *p*% of neurons to 0 in the forward pass

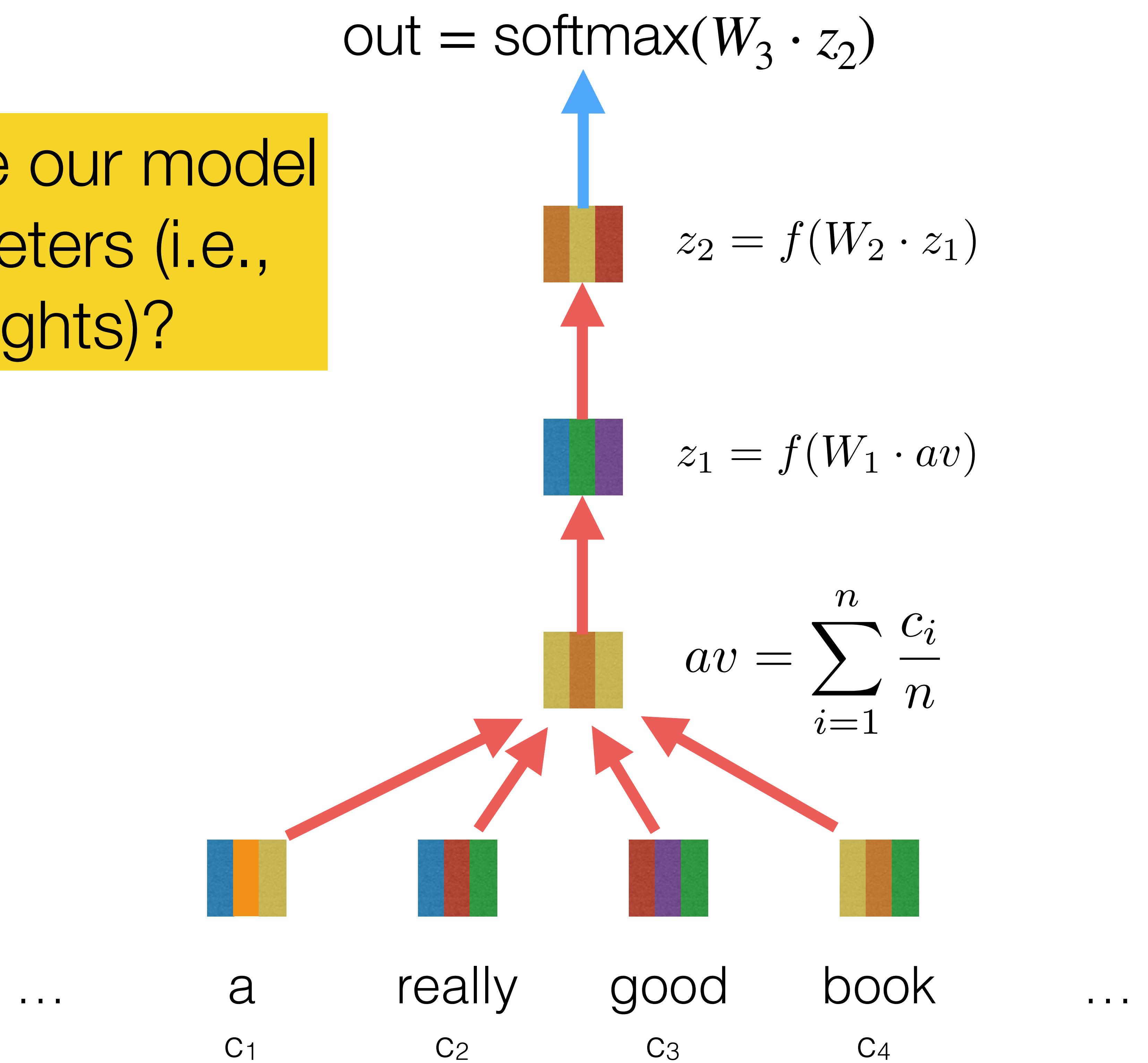


(a) Standard Neural Net (b) After applying dropout.

[Srivastava et al., 2014]

deep averaging networks

what are our model parameters (i.e., weights)?



backpropagation

- use the chain rule to compute partial derivatives w/ respect to each parameter
- trick: re-use derivatives computed for higher layers to compute derivatives for lower layers!

$$\frac{\partial L}{\partial c_i} = \boxed{\frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2}} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial \text{av}} \frac{\partial \text{av}}{\partial c_i}$$

$$\frac{\partial L}{\partial W_2} = \boxed{\frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2}} \frac{\partial z_2}{\partial W_2}$$

Rumelhart et al., 1986

language models

back to reality…

Probabilistic Language Modeling

- Goal: compute the probability of a sentence or sequence of words:

  $P(W) = P(w_1,w_2,w_3,w_4,w_5 \ldots w_n)$

- Related task: probability of an upcoming word:

  $P(w_5 | w_1,w_2,w_3,w_4)$

- A model that computes either of these:

  $P(W)$ or $P(w_n | w_1,w_2 \ldots w_{n-1})$ is called a **language model** or **LM**

we have already seen one way to do this… where?

How to compute P(W)

- How to compute this joint probability:
    - P(its, water, is, so, transparent, that)
- Intuition: let's rely on the Chain Rule of Probability

The Chain Rule applied to compute joint probability of words in sentence

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

P("its water is so transparent") =
P(its) × P(water|its) × P(is|its water)
× P(so|its water is) × P(transparent|its water is so)

How to estimate these probabilities

- Could we just count and divide?

$$P(\text{the} \mid \text{its water is so transparent that}) = \frac{Count(\text{its water is so transparent that the})}{Count(\text{its water is so transparent that})}$$

- No! Too many possible sentences!
- We'll never see enough data for estimating these

Markov Assumption

Andrei Markov (1856~1922)

- Simplifying assumption:

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{that})$$

- Or maybe

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{transparent that})$$

Estimating bigram probabilities

- The Maximum Likelihood Estimate (MLE)
  - relative frequency based on the empirical counts on a training set

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

c — count

Perplexity

The best language model is one that best predicts an unseen test set

- Gives the highest P(sentence)

Perplexity is the inverse probability of the test set, normalized by the number of words:

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Chain rule:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

For bigrams:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

Minimizing perplexity is the same as maximizing probability

Lower perplexity = better model

- Training 38 million words, test 1.5 million words, Wall Street Journal

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

Add-one estimation (again!)

- Also called Laplace smoothing
- Pretend we saw each word one more time than we did
- Just add one to all the counts!
- MLE estimate: $$P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$
- Add-1 estimate: $$P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

Compare with raw bigram counts

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Backoff and Interpolation

- Sometimes it helps to use **less** context
  - Condition on less context for contexts you haven't learned much about
- **Backoff:**
  - use trigram if you have good evidence,
  - otherwise bigram, otherwise unigram
- **Interpolation:**
  - mix unigram, bigram, trigram

- Interpolation works better

Absolute Discounting Interpolation

- Save ourselves some time and just subtract 0.75 (or some d)!

$$P_{\text{AbsoluteDiscounting}}(w_i \mid w_{i-1}) = \underbrace{\frac{c(w_{i-1}, w_i) - d}{c(w_{i-1})}}_{\text{discounted bigram}} + \underbrace{\lambda(w_{i-1})}_{\text{Interpolation weight}} \underbrace{P(w)}_{\text{unigram}}$$

- (Maybe keeping a couple extra values of d for counts 1 and 2)
- But should we really just use the regular unigram P(w)?

Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their* $\boldsymbol{w}_j$*"* never occurred in data? Then $\boldsymbol{w}_j$ has probability 0!

(Partial) Solution: Add small δ to count for every $\boldsymbol{w}_j \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}_j|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w}_j)}}{\boxed{\text{count}(\text{students opened their})}}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any* $\boldsymbol{w}_j$!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Problems with n-gram Language Models

Storage: Need to store count for all possible *n*-grams. So model size is O(exp(*n*)).

$$P(\boldsymbol{w}_j|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w}_j)}}{\text{count}(\text{students opened their})}$$

Increasing *n* makes model size huge!

A RNN Language Model

output distribution

$$\hat{y} = \text{softmax}(W_2 h^{(t)} + b_2)$$

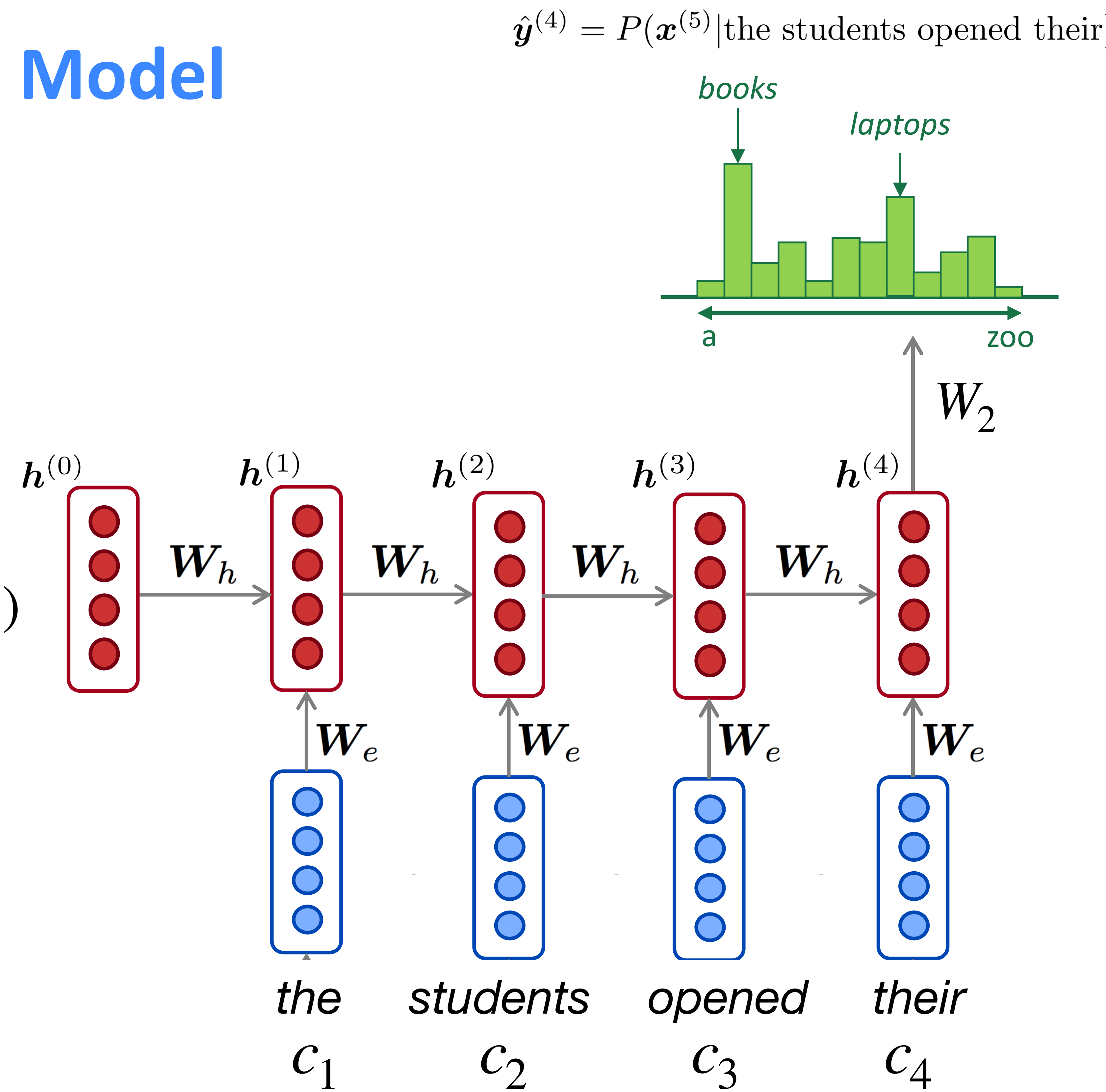
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t + b_1)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



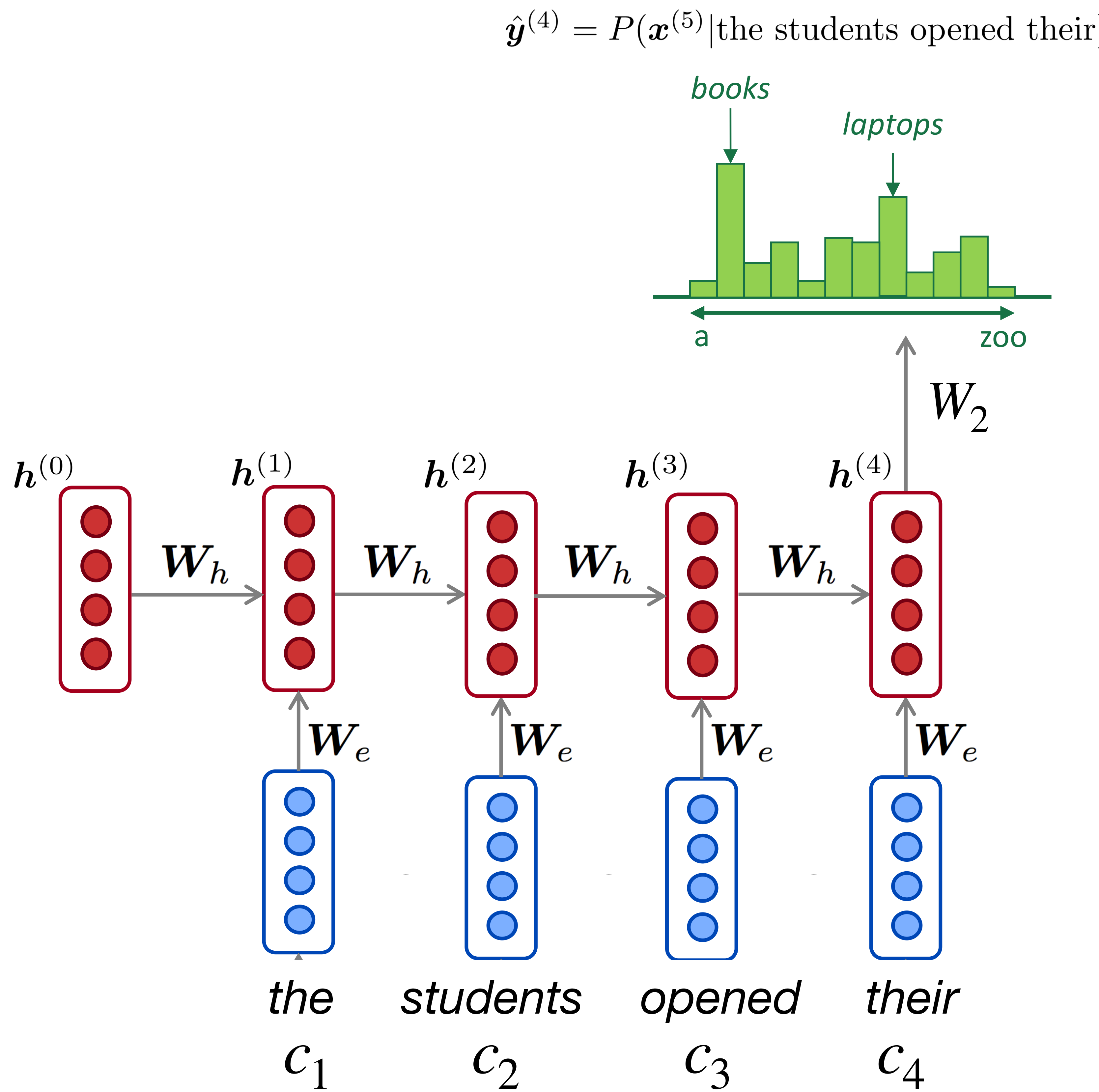
why is this good?

RNN **Advantages**:

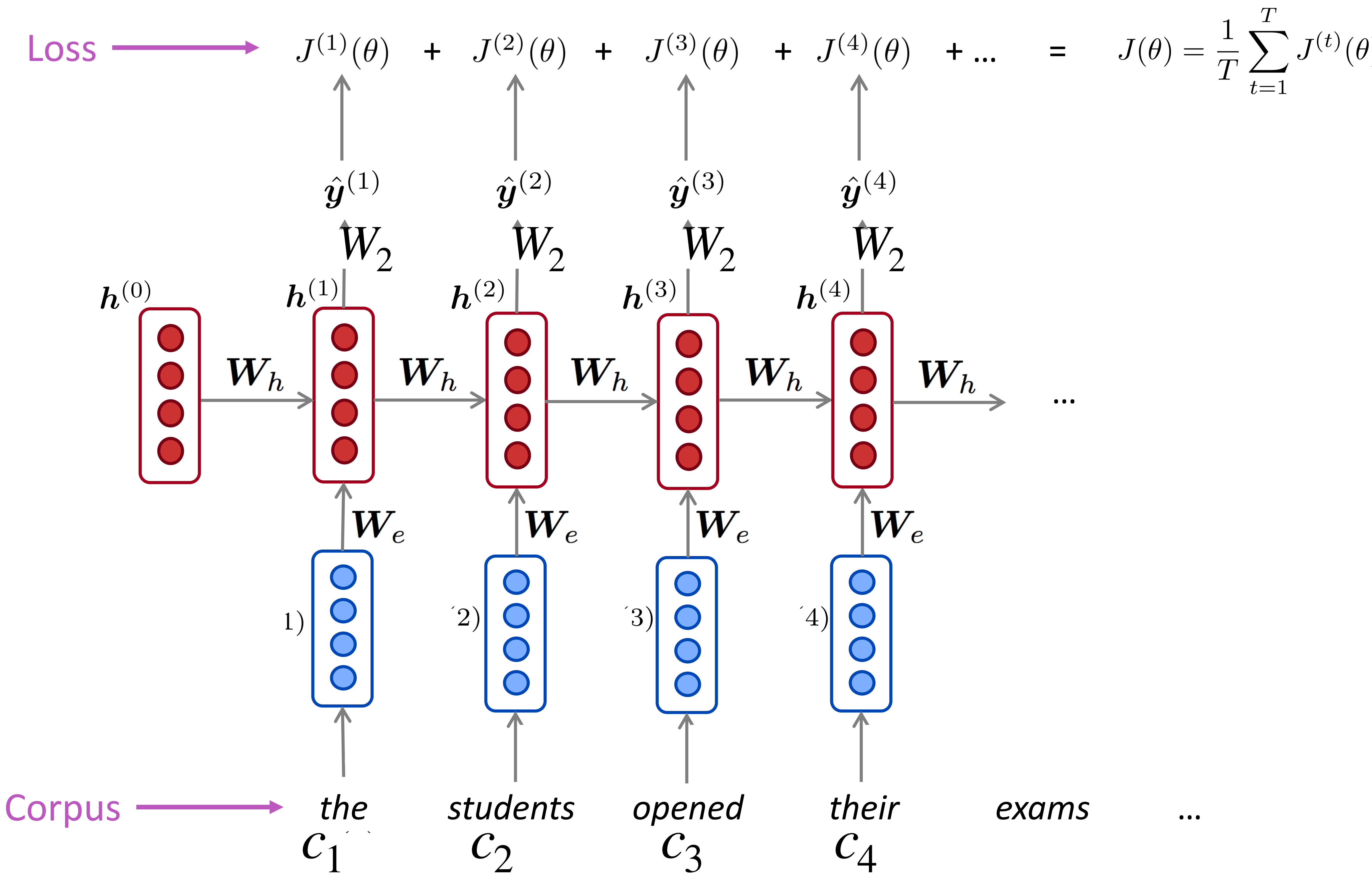
- Can process any length input
- Model size doesn't increase for longer input
- Computation for step *t* can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared

RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back

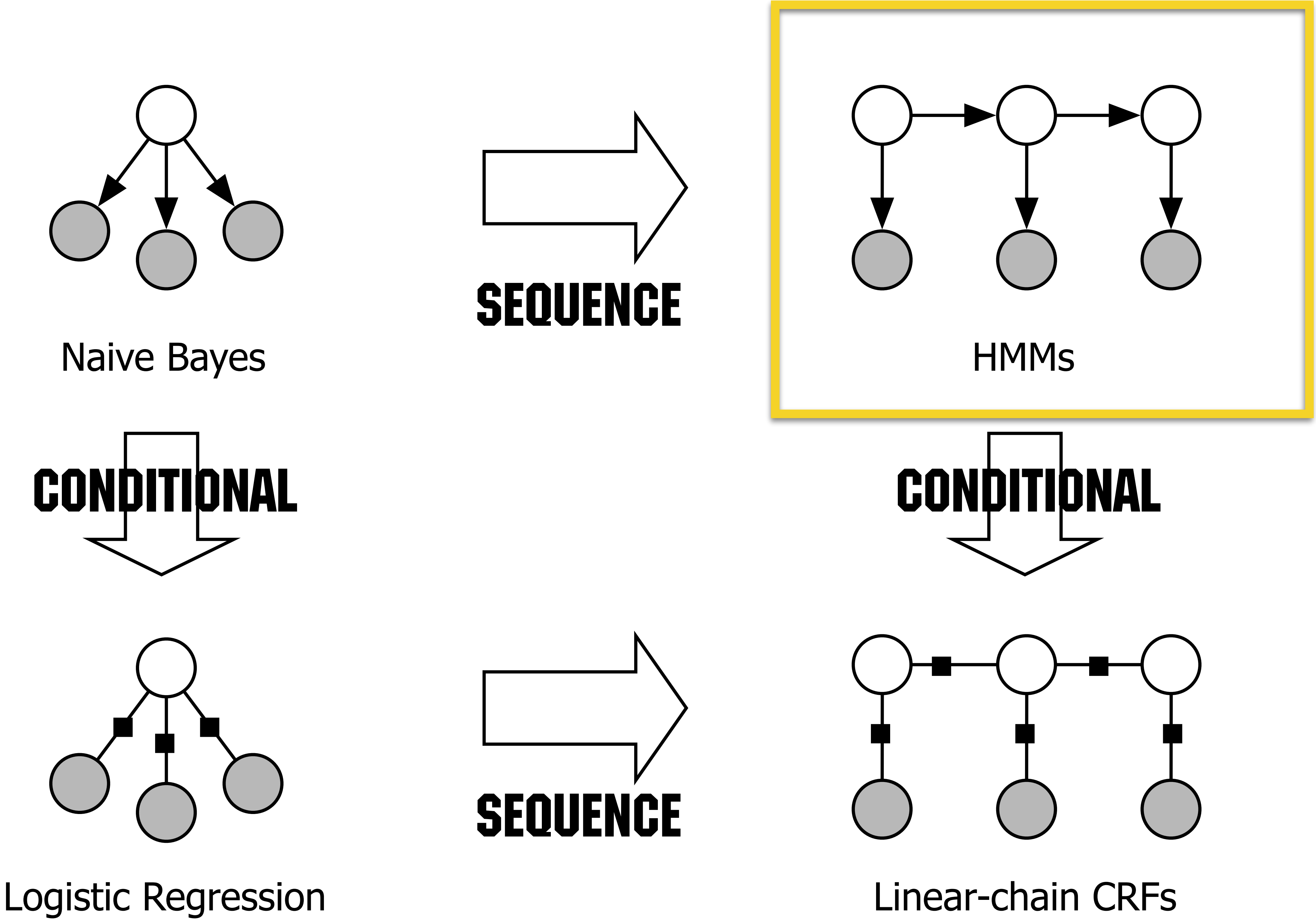


Training a RNN Language Model



POS tagging / HMMs

These are all **log-linear** models



are neural networks log-linear models?

Tagging (Sequence Labeling)

- Given a sequence (in NLP, words), assign appropriate labels to each word.
- Many NLP problems can be viewed as sequence labeling:
  - POS Tagging
  - Chunking
  - Named Entity Tagging
- Labels of tokens are dependent on the labels of other tokens in the sequence, particularly their neighbors

Plays	well	with	others.
VBZ	RB	IN	NNS

Open class (lexical) words

Nouns
- Proper: *IBM*, *Italy*
- Common: *cat / cats*, *snow*

Verbs
- Main: *see*, *registered*
- Modals: *can*, *had*

Adjectives
old older oldest

Adverbs
slowly

Numbers
122,312, *one*

… more

Closed class (functional)

- Determiners *the some*
- Conjunctions *and or*
- Pronouns *he its*
- Prepositions *to with*
- Particles *off up*
- Interjections *Ow Eh*

… more

Two Types of Constraints

Influential/JJ members/NNS of/IN the/DT House/NNP Ways/NNP and/CC Means/NNP Committee/NNP introduced/VBD legislation/NN that/WDT would/MD restrict/VB how/WRB the/DT new/JJ savings-and-loan/NN bailout/NN agency/NN can/MD raise/VB capital/NN ./.

- "Local": e.g., *can* is more likely to be a modal verb MD rather than a noun NN
- "Contextual": e.g., a noun is much more likely than a verb to follow a determiner
- Sometimes these preferences are in conflict:

  The trash can is in the garage

Hidden Markov Models

- We have an input sentence $x = x_1, x_2, \ldots, x_n$ (x_i is the i'th word in the sentence)
- We have a tag sequence $y = y_1, y_2, \ldots, y_n$ (y_i is the i'th tag in the sentence)
- We'll use an HMM to define

$$p(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$$

for any sentence $x_1 \ldots x_n$ and tag sequence $y_1 \ldots y_n$ of the same length.

- Then the most likely tag sequence for x is

$$\arg\max_{y_1 \ldots y_n} p(x_1 \ldots x_n, y_1, y_2, \ldots, y_n)$$

are HMMs generative or discriminative models?

HMM Definition

Assume K parts of speech, a lexicon size of V, a series of observations $\{x_1, \dots, x_N\}$, and a series of unobserved states $\{z_1, \dots, z_N\}$.

- π A distribution over start states (vector of length K): $\pi_i = p(z_1 = i)$
- θ Transition matrix (matrix of size K by K): $\theta_{i,j} = p(z_n = j | z_{n-1} = i)$ Markov assumption!
- β An emission matrix (matrix of size K by V): $\beta_{j,w} = p(x_n = w | z_n = j)$

VBZ	CONJ	VBZ	PRO
come	and	get	it

joint prob $p(x_1, x_2, x_3, x_4, z_1, z_2, z_3, z_4) = ???$

VBZ CONJ VBZ PRO

come and get it

joint prob $p(x_1, x_2, x_3, x_4, z_1, z_2, z_3, z_4) = ???$

$$\overbrace{}^{\pi_{\text{VBZ}}} \quad \overbrace{}^{\beta_{\text{VBZ, come}}} \quad \overbrace{}^{\theta_{\text{VBZ, CONJ}}}$$

$$= p(\text{VBZ})p(\text{come} \mid \text{VBZ})p(\text{CONJ} \mid \text{VBZ})$$
$$p(\text{and} \mid \text{CONJ})p(\text{VBZ} \mid \text{CONJ})p(\text{get} \mid \text{VBZ})$$
$$p(\text{PRO} \mid \text{VBZ})p(\text{it} \mid \text{PRO})$$

Training Sentences

x = tokens
z = POS tags

x	here	come	old	flattop
z	MOD	V	MOD	N

a	crowd	of	people	stopped	and	stared
DET	N	PREP	N	V	CONJ	V

gotta	get	you	into	my	life
V	V	PRO	PREP	PRO	V

and	I	love	her
CONJ	PRO	V	PRO

Initial Probability π

POS	Frequency	Probability
MOD	1.1	0.234
DET	1.1	0.234
CONJ	1.1	0.234
N	0.1	0.021
PREP	0.1	0.021
PRO	0.1	0.021
V	1.1	0.234

let's use add-alpha smoothing with alpha = 0.1

Transition Probability θ

- We can ignore the words; just look at the parts of speech. Let's compute one row, the row for verbs.
- We see the following transitions: V $\rightarrow$ MOD, V $\rightarrow$ CONJ, V $\rightarrow$ V, V $\rightarrow$ PRO, and V $\rightarrow$ PRO

POS	Frequency	Probability
MOD	1.1	0.193
DET	0.1	0.018
CONJ	1.1	0.193
N	0.1	0.018
PREP	0.1	0.018
PRO	2.1	0.368
V	1.1	0.193

how many transition probability distributions do we have?

Emission Probability β

Let's look at verbs . . .

Word	a	and	come	crowd	flattop
Frequency	0.1	0.1	1.1	0.1	0.1
Probability	0.0125	0.0125	0.1375	0.0125	0.0125

Word	get	gotta	her	here	i
Frequency	1.1	1.1	0.1	0.1	0.1
Probability	0.1375	0.1375	0.0125	0.0125	0.0125

Word	into	it	life	love	my
Frequency	0.1	0.1	0.1	1.1	0.1
Probability	0.0125	0.0125	0.0125	0.1375	0.0125

Word	of	old	people	stared	stopped
Frequency	0.1	0.1	0.1	1.1	1.1
Probability	0.0125	0.0125	0.0125	0.1375	0.1375

how many emission probability distributions do we have?

Viterbi Algorithm

- Given an unobserved sequence of length L, $\{x_1, \ldots, x_L\}$, we want to find a sequence $\{z_1 \ldots z_L\}$ with the highest probability.
- It's impossible to compute K^L possibilities.
- So, we use dynamic programming to compute most likely tags for each token subsequence from 0 to t that ends in state k.
- Memoization: fill a table of solutions of sub-problems
- Solve larger problems by composing sub-solutions
- Base case:

$$\delta_1(k) = \pi_k \beta_{k,x_i} \tag{1}$$

- Recursion:

$$\delta_n(k) = \max_j \left(\delta_{n-1}(j)\theta_{j,k}\right)\beta_{k,x_n} \tag{2}$$

Viterbi Algorithm

- Given an unobserved sequence of length L, $\{x_1, \ldots, x_L\}$, we want to find a sequence $\{z_1 \ldots z_L\}$ with the highest probability.
- It's impossible to compute K^L possibilities.
- So, we use dynamic programming to compute most likely tags for each token subsequence from 0 to t that ends in state k.
- Memoization: fill a table of solution
- Solve larger problems by composing

for first time step:
p_1(tag) = initial prob(tag) *
emission prob (word$_1$ | tag)

- Base case:

$$\delta_1(k) = \pi_k \beta_{k,x_i} \tag{1}$$

- Recursion:

$$\delta_n(k) = \max_j \left(\delta_{n-1}(j)\theta_{j,k}\right)\beta_{k,x_n} \tag{2}$$

Viterbi Algorithm

- Given an unobserved sequence of length L, $\{x_1, \ldots, x_L\}$, we want to find a sequence $\{z_1 \ldots z_L\}$ with the highest probability.
- It's impossible to compute K^L possibilities.
- So, we use dynamic programming to compute most likely tags for each token subsequence from 0 to t that ends in state k.
- Memoization: fill a table of solution

composing

for first time step:
p_1(tag) = initial prob(tag) * emission prob (word$_1$ | tag)

for all other time steps:
p_n(tag) = max over prev_tag (p_{n-1}(prev_tag) * transition prob(tag|prev_tag)) * emission prob(word | tag)

$$\delta_1(k) = \pi_k \beta_{k,x_i} \tag{1}$$

$$\delta_n(k) = \max_j \left(\delta_{n-1}(j)\theta_{j,k}\right)\beta_{k,x_n} \tag{2}$$

POS	π_k	β_{k,x_1}	$\log \delta_1(k) = \log(\pi_k \beta_{k,x_1})$
MOD	0.234	0.024	-5.18
DET	0.234	0.032	-4.89
CONJ	0.234	0.024	-5.18
N	0.021	0.016	-7.99
PREP	0.021	0.024	-7.59
PRO	0.021	0.016	-7.99
V	0.234	0.121	-3.56

come and get it

Why logarithms?

1. More interpretable than a float with lots of zeros.
2. Underflow is less of an issue
3. Addition is cheaper than multiplication

$$log(ab) = log(a) + log(b) \tag{4}$$

POS	π_k	β_{k,x_1}	$\log \delta_1(k)$
MOD	0.234	0.024	-5.18
DET	0.234	0.032	-4.89
CONJ	0.234	0.024	-5.18
N	0.021	0.016	-7.99
PREP	0.021	0.024	-7.59
PRO	0.021	0.016	-7.99
V	0.234	0.121	-3.56

$= \log(\pi_k \beta_{k,x_1})$

come and

for first time step:
$p_1(\text{tag})$ = initial prob(tag) *
emission prob (word_1 | tag)

Why logarithms?

1. More interpretable than a float with lots of zeros.
2. Underflow is less of an issue
3. Addition is cheaper than multiplication

$$log(ab) = log(a) + log(b) \tag{4}$$

POS	$\log \delta_1(j)$	$\log \delta_1(j)\theta_{j,\text{CONJ}}$	$\log \delta_2(\text{CONJ})$
MOD	-5.18		
DET	-4.89		
CONJ	-5.18		???
N	-7.99		
PREP	-7.59		
PRO	-7.99		
V	-3.56	???	

come **and** get it

$$\log\left(\delta_0(\text{V})\theta_{\text{V, CONJ}}\right) = \log \delta_0(k) + \log \theta_{\text{V, CONJ}} = -3.56 + -1.65$$

for all other time steps:
p_n(tag) = max over prev_tag
(p_{n-1}(prev_tag) * transition prob(tag|prev_tag))
* emission prob(word | tag)

POS	$\log \delta_1(j)$	$\log \delta_1(j)\theta_{j,\text{CONJ}}$	$\log \delta_2(\text{CONJ})$
MOD	-5.18		
DET	-4.89		
CONJ	-5.18		???
N	-7.99		
PREP	-7.59		
PRO	-7.99		
V	-3.56	???	

come **and** get it

$$\log\left(\delta_0(\text{V})\theta_{\text{V, CONJ}}\right) = \log \delta_0(k) + \log \theta_{\text{V, CONJ}} = -3.56 + -1.65$$

this computation is inside the **max:**

p_{n-1}(V) * transition prob(CONJ|V))

for all other time steps:

p_n(tag) = max over prev_tag (p_{n-1}(prev_tag) * transition prob(tag|prev_tag)) * emission prob(word | tag)

POS	$\log \delta_1(j)$	$\log \delta_1(j)\theta_{j,\text{CONJ}}$	$\log \delta_2(\text{CONJ})$
MOD	-5.18	-8.48	
DET	-4.89	-7.72	
CONJ	-5.18	-8.47	???
N	-7.99	≤ -7.99	
PREP	-7.59	≤ -7.59	
PRO	-7.99	≤ -7.99	
V	-3.56	-5.21	

come **and** get it

do the computation for all possible prev tags:
p_{n-1}(prev_tag) *
transition prob(CONJ|prev_tag))
and then take the max, which happens to be **V** here

POS	$\log \delta_1(j)$	$\log \delta_1(j)\theta_{j,\text{CONJ}}$	$\log \delta_2(\text{CONJ})$
MOD	-5.18	-8.48	
DET	-4.89	-7.72	
CONJ	-5.18	-8.47	
N	-7.99	≤ -7.99	
PREP	-7.59	≤ -7.59	
PRO	-7.99	≤ -7.99	
V	-3.56	-5.21	

come **and** get it

now just multiply by the emission probability p(word$_2$|CONJ) to get the final p$_2$(CONJ)

$$\log \delta_1(k) = -5.21 + \log \beta_{\text{CONJ, and}} = -5.21 - 0.64$$

backpointer!

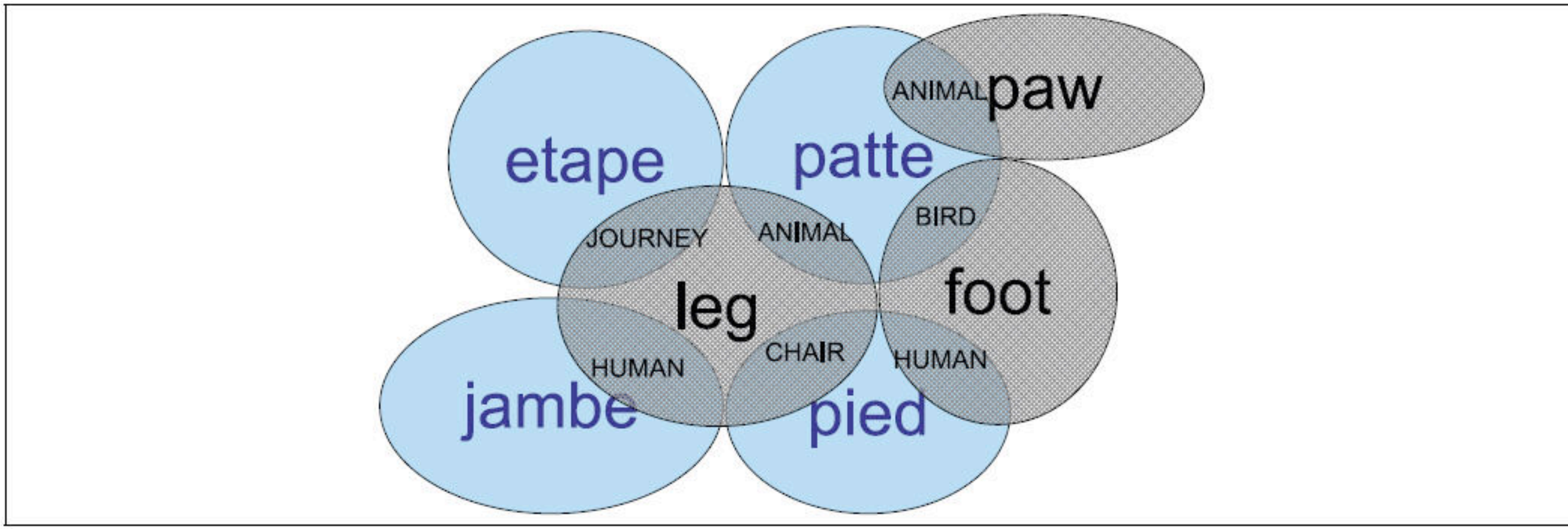
POS	$\delta_1(k)$	$\delta_2(k)$	b_2	$\delta_3(k)$	b_3	$\delta_4(k)$	b_4
MOD	-5.18						
DET	-4.89						
CONJ	-5.18	-6.02	V				
N	-7.99						
PREP	-7.59						
PRO	-7.99						
V	-3.56						
WORD	come	and		get		it	

to find p_2(CONJ), we had to compute a max over k previous states.
the same is true for p_2(N), p_2(PREP), etc.
for one time step, complexity is k^2!

machine translation

MT is hard

- Word meaning:
  many-to-many and context dependent



- *Translation* itself is hard: metaphors, cultural references, etc.

Recap: The Noisy Channel Model

- Goal: translation system from French to English
- Have a model $p(e \mid f)$ which estimates conditional probability of any English sentence e given the French sentence f. Use the training corpus to set the parameters.
- A Noisy Channel Model has two components:

$$p(e) \quad \textbf{the language model}$$

$$p(f \mid e) \quad \textbf{the translation model}$$

- Giving:

$$p(e \mid f) = \frac{p(e, f)}{p(f)} = \frac{p(e)p(f \mid e)}{\sum_e p(e)p(f \mid e)}$$

and

$$\text{argmax}_e p(e \mid f) = \text{argmax}_e p(e)p(f \mid e)$$

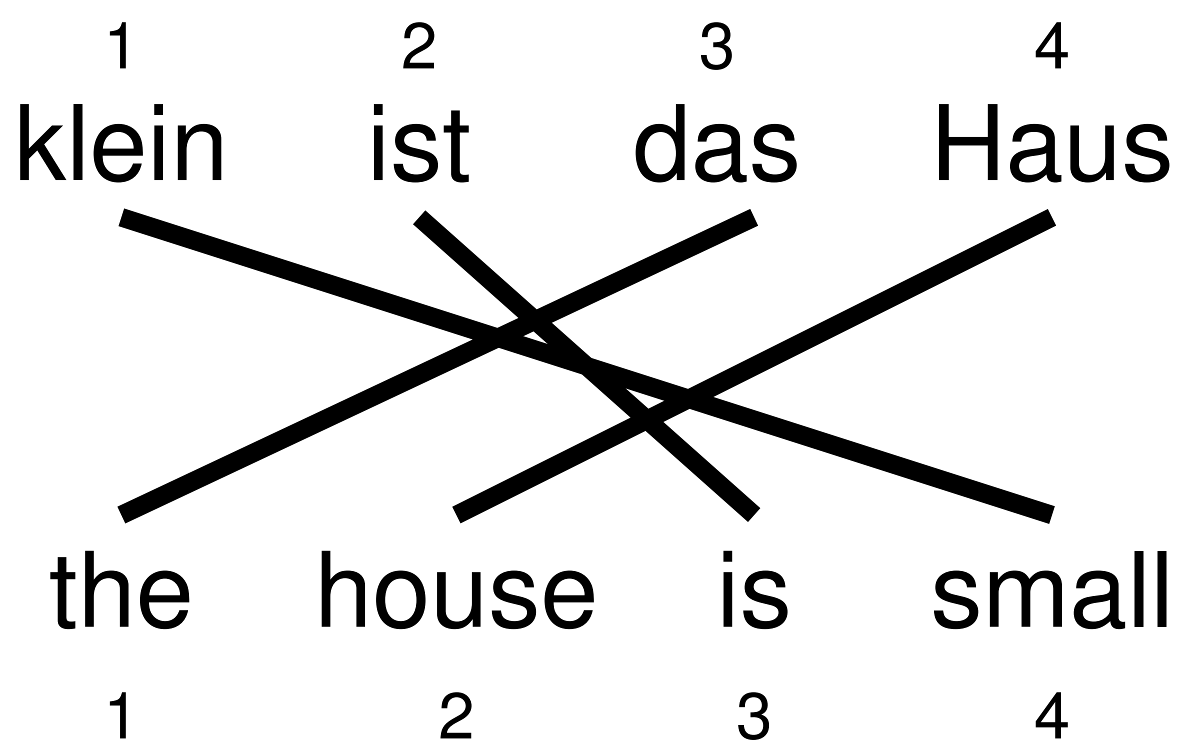
Alignment Function

- Formalizing alignment with an alignment function
- Mapping an English target word at position i to a German source word at position j with a function $a : i \to j$
- Example

$$a : \{1 \to 1, 2 \to 2, 3 \to 3, 4 \to 4\}$$

Reordering

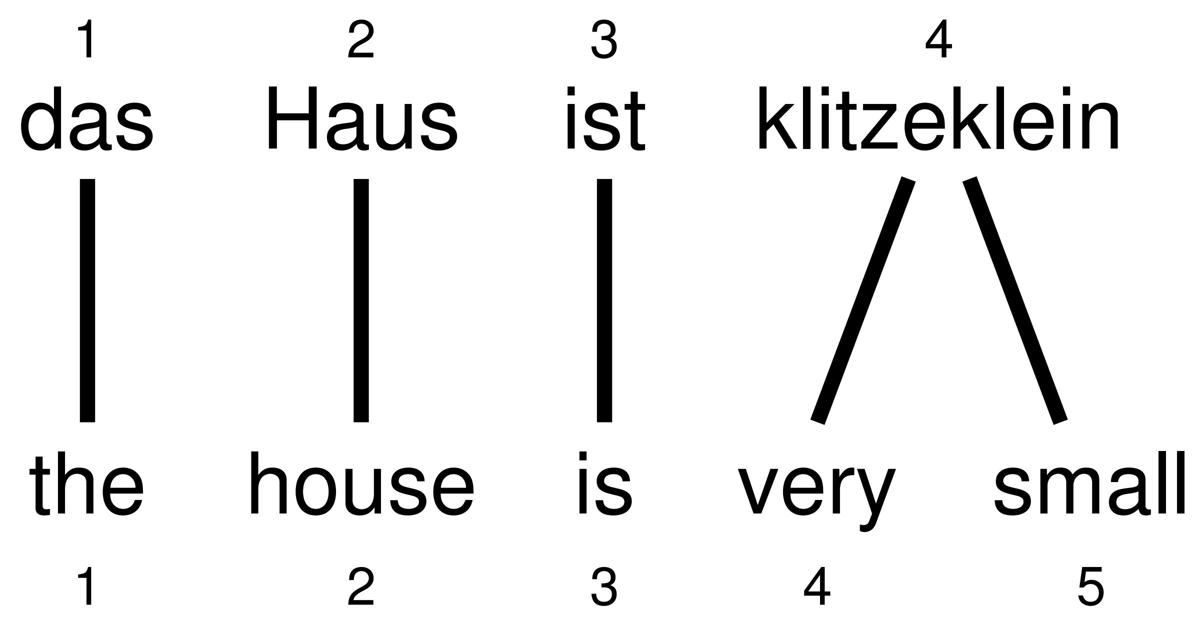
Words may be reordered during translation



$$a : \{1 \to 3, 2 \to 4, 3 \to 2, 4 \to 1\}$$

One-to-Many Translation

A source word may translate into multiple target words



$$a : \{1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 4\}$$

IBM Model 1: Alignments

- How do we model $p(f \mid e)$?

translation model in noisy channel

- English sentence e has l words $e_1 \dots e_l$,
  French sentence f has m words $f_1 \dots f_m$.

- An alignment a identifies which English word each French word originated from

- Formally, an alignment a is $\{a_1, \dots a_m\}$, where each $a_j \in \{0 \dots l\}$.

- There are $(l+1)^m$ possible alignments.

IBM Model 1: The Generative Process

To generate a French string f from an English string e:

- **Step 1:** Pick an alignment a with probability $\frac{1}{(l+1)^m}$
- **Step 2:** Pick the French words with probability

$$p(f \mid a, e, m) = \prod_{j=1}^{m} t(f_j \mid e_{a_j})$$

The final result:

$$p(f, a \mid e, m) = p(a \mid e, m) \times p(f \mid a, e, m) = \frac{1}{(l+1)^m} \prod_{j=1}^{m} t(f_j \mid e_{a_j})$$

example

- e.g., $l = 6$, $m = 7$

$$e = \text{And the program has been implemented}$$

$$f = \text{Le programme a ete mis en application}$$

- $a = \{2, 3, 4, 5, 6, 6, 6\}$

$$\begin{aligned}
p(f \mid a, e) \quad = \quad & t(Le \mid the) \times \\
& t(programme \mid program) \times \\
& t(a \mid has) \times \\
& t(ete \mid been) \times \\
& t(mis \mid implemented) \times \\
& t(en \mid implemented) \times \\
& t(application \mid implemented)
\end{aligned}$$

chicken & egg problem!

- if we had the alignments, we could estimate the parameters of our model (i.e., the lexical translation probabilities)
- if we had the parameters, we could estimate the alignments.

- we have neither! :(

Parameter Estimation if the Alignments are Observed

- First: case where alignments are observed in training data. E.g.,

$$\begin{aligned} e^{(100)} &= \text{And the program has been implemented} \\ f^{(100)} &= \text{Le programme a ete mis en application} \\ a^{(100)} &= \langle 2, 3, 4, 5, 6, 6, 6 \rangle \end{aligned}$$

- Training data is $(e^{(k)}, f^{(k)}, a^{(k)})$ for $k = 1 \ldots n$. Each $e^{(k)}$ is an English sentence, each $f^{(k)}$ is a French sentence, each $a^{(k)}$ is an alignment
- Maximum-likelihood parameter estimates in this case are trivial:

$$t_{ML}(f|e) = \frac{\text{Count}(e,\ f)}{\text{Count}(e)}$$

EM algorithm

- Expectation maximization (EM) in a nutshell:
  1. initialize model parameters (trans. probs) using some method (e.g., uniform)
  2. assign probabilities to missing data (alignments)
  3. estimate model parameters from the completed data
  4. iterate steps 2-3 until convergence

dataset:

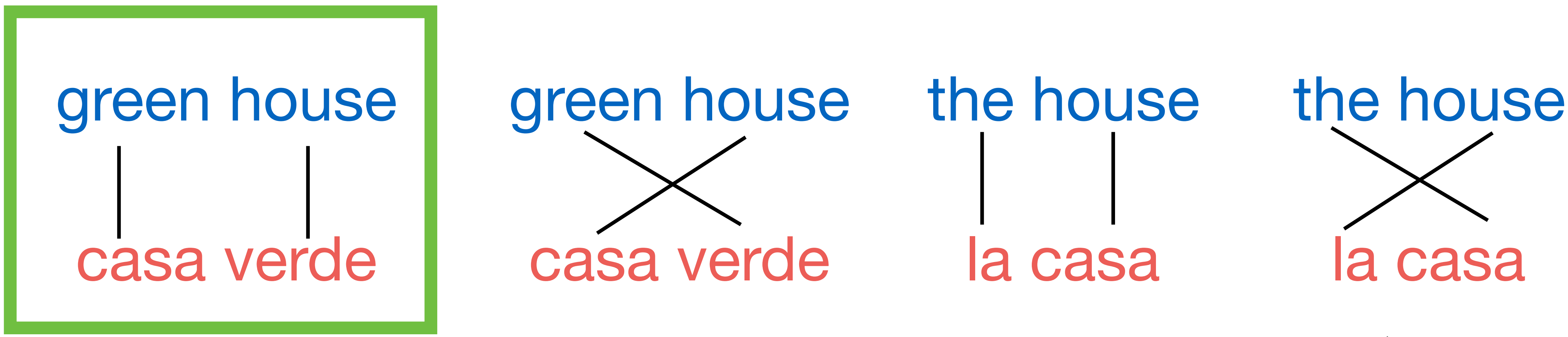
green house   the house
casa verde   la casa

initialize translation probabilities uniformly:

t(casa|green) = 1/3	t(verde|green) = 1/3	t(la|green) = 1/3
t(casa|house) = 1/3	t(verde|house) = 1/3	t(la|house) = 1/3
t(casa|the) = 1/3	t(verde|the) = 1/3	t(la|the) = 1/3

E-Step 1: compute expected counts E[*count(t(f,e)*]

first, for all alignments, let's compute $p(f, a \mid e) = \prod_{j=1}^{m} t(f_j \mid e_{a_j})$



$$p(f, a \mid e) = t(\text{casa} \mid \text{green}) \times t(\text{verde} \mid \text{house}) = \frac{1}{9}$$

E-Step 1: compute expected counts E[*count(t(f,e)*)]

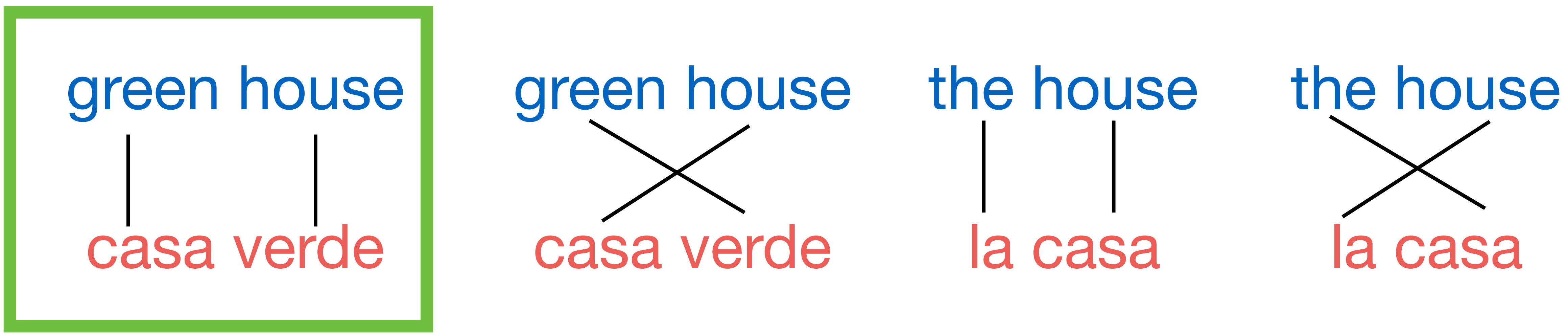
first, for all alignments, let's compute $p(f, a \mid e) = \prod_{j=1}^{m} t(f_j \mid e_{a_j})$

green house	green house	the house	the house
casa verde	casa verde	la casa	la casa
$p(f, a \mid e) = \frac{1}{9}$	$p(f, a \mid e) = \frac{1}{9}$	$p(f, a \mid e) = \frac{1}{9}$	$p(f, a \mid e) = \frac{1}{9}$

E-Step 1: compute expected counts E[*count(t(f,e)*)]

next, let's compute *alignment probabilities* by normalizing:

$$p(a \mid f, e) = \frac{p(a, f \mid e)}{\sum_a p(a, f \mid e)}$$



$$p(a \mid f, e) = \frac{\frac{1}{9}}{\frac{2}{9}} = \frac{1}{2}$$

E-Step 1: compute expected counts E[*count(t(f,e)*]

now let's finally compute expected (fractional) counts for each (f,e) pair

there is exactly one casa—green alignment with prob. 1/2

t(casa|green) = 1/2	t(verde|green) =	t(la|green) =	total(green) =
t(casa|house) =	t(verde|house) =	t(la|house) =	total(house) =
t(casa|the) =	t(verde|the) =	t(la|the) =	total(the) =

M-Step 1: compute MLE counts by normalizing

easy! just normalize each row to sum to 1

t(casa|green) = 1/2	t(verde|green) = 1/2	t(la|green) = 0
t(casa|house) = 1/2	t(verde|house) = 1/4	t(la|house) = 1/4
t(casa|the) = 1/2	t(verde|the) = 0	t(la|the) = 1/2

note that each of the correct translations have increased in probability! t(casa|house) is now 1/2 instead of 1/3

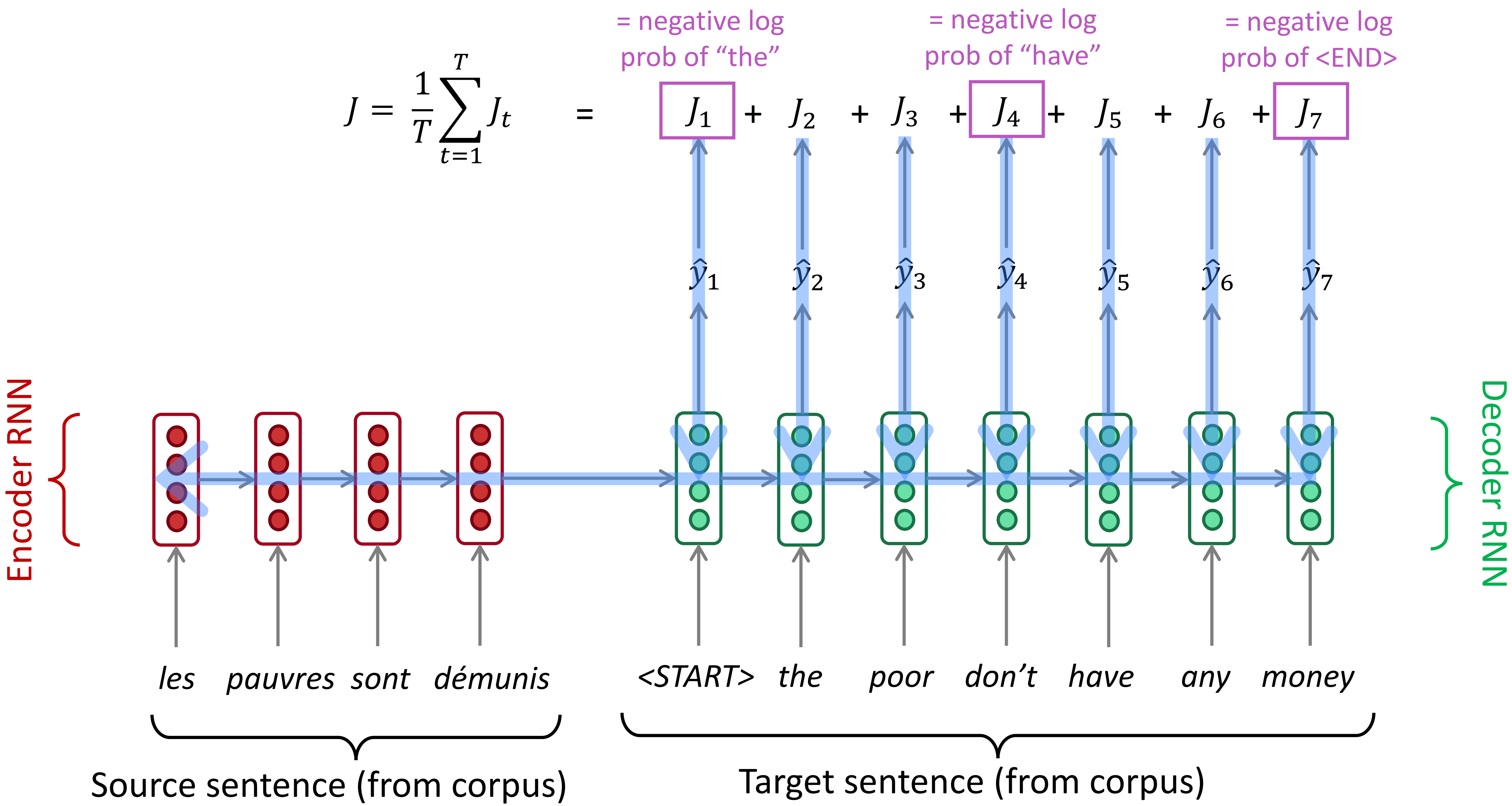
limitations of IBM models

- *discrete* alignments
- all alignments equally likely (model 1 only)
- translation of each *f* word depends only on aligned *e* word!

seq2seq models

- use two different RNNs to model $\prod_{i=1}^{L} p(e_i \,|\, e_1, \ldots, e_{i-1}, f)$
- first we have the *encoder*, which encodes the foreign sentence *f*
- then, we have the *decoder,* which produces the English sentence *e*

Training a Neural Machine Translation system



what are the parameters of this model?

$$W_h^{enc}, W_e^{enc}, C^{enc}, W_h^{dec}, W_e^{dec}, C^{dec}, W_{out}$$

C is word embedding matrix

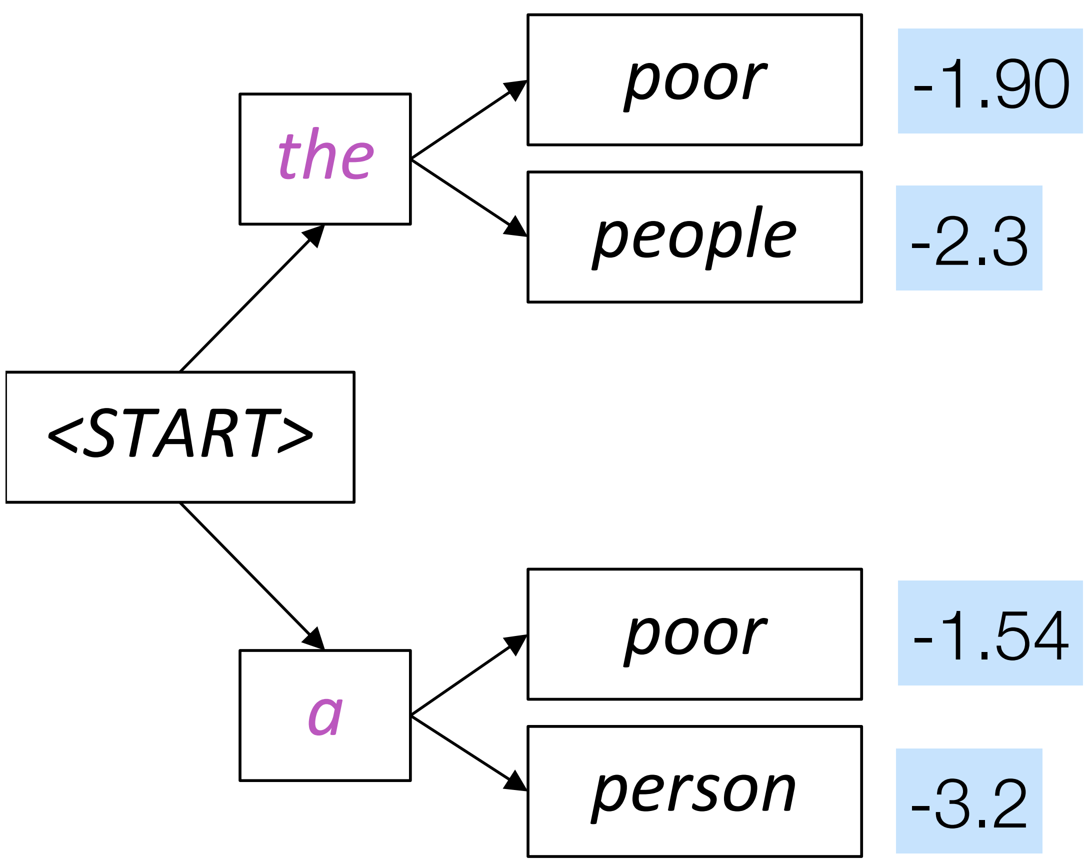
Beam search

- in greedy decoding, we cannot go back and revise previous decisions!
  - *les pauvres sont démunis (the poor don't have any money)*
  - → *the* ____
  - → *the poor* ____
  - → *the poor are* ____
- fundamental idea of beam search: explore several different hypotheses instead of just a single one
  - keep track of *k* most probable partial translations at each decoder step instead of just one!

    the beam size *k* is usually 5-10

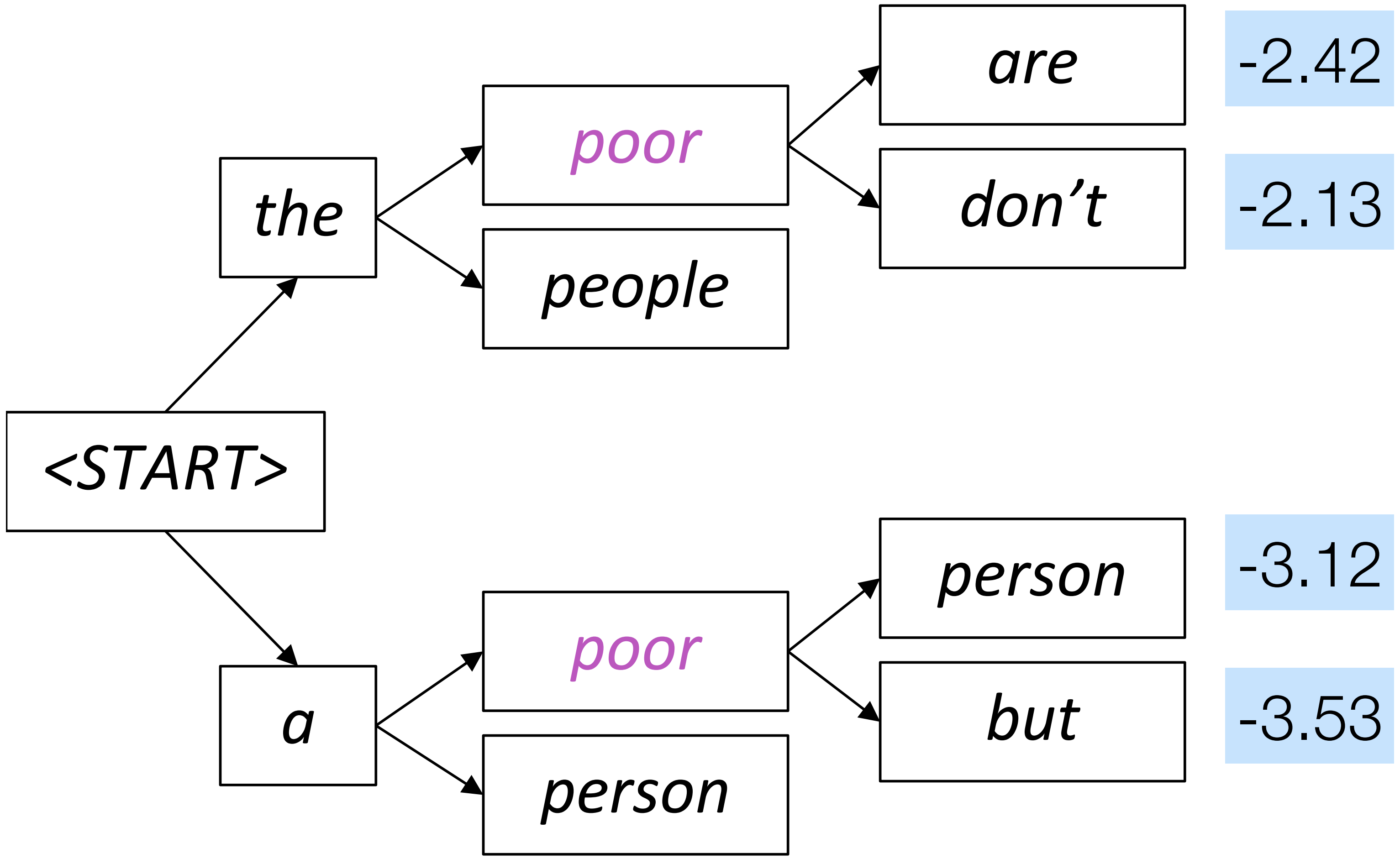
Beam search decoding: example

Beam size = 2



Beam search decoding: example

Beam size = 2

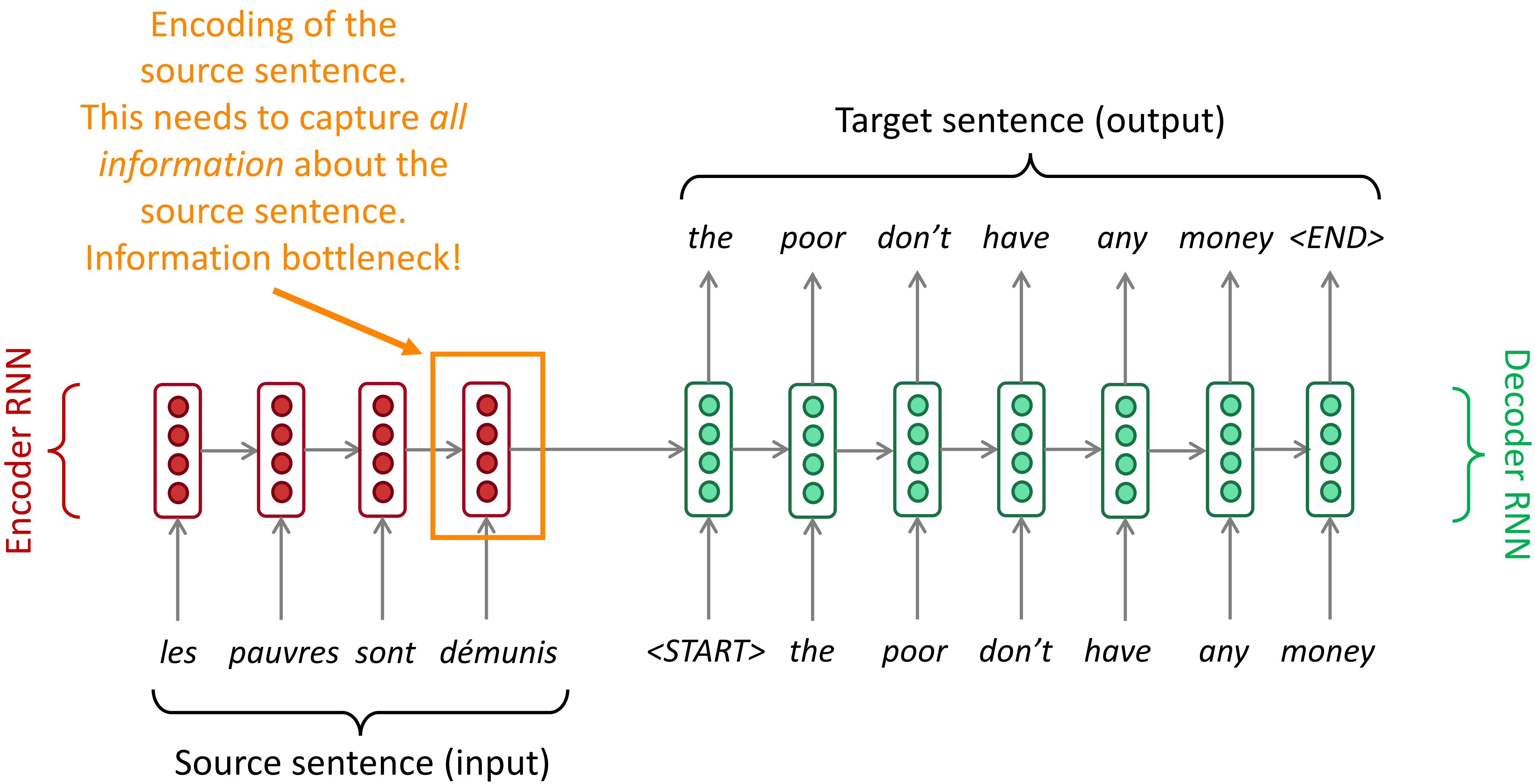


does beam search always produce the *best* translation (i.e., does it always find the argmax?)

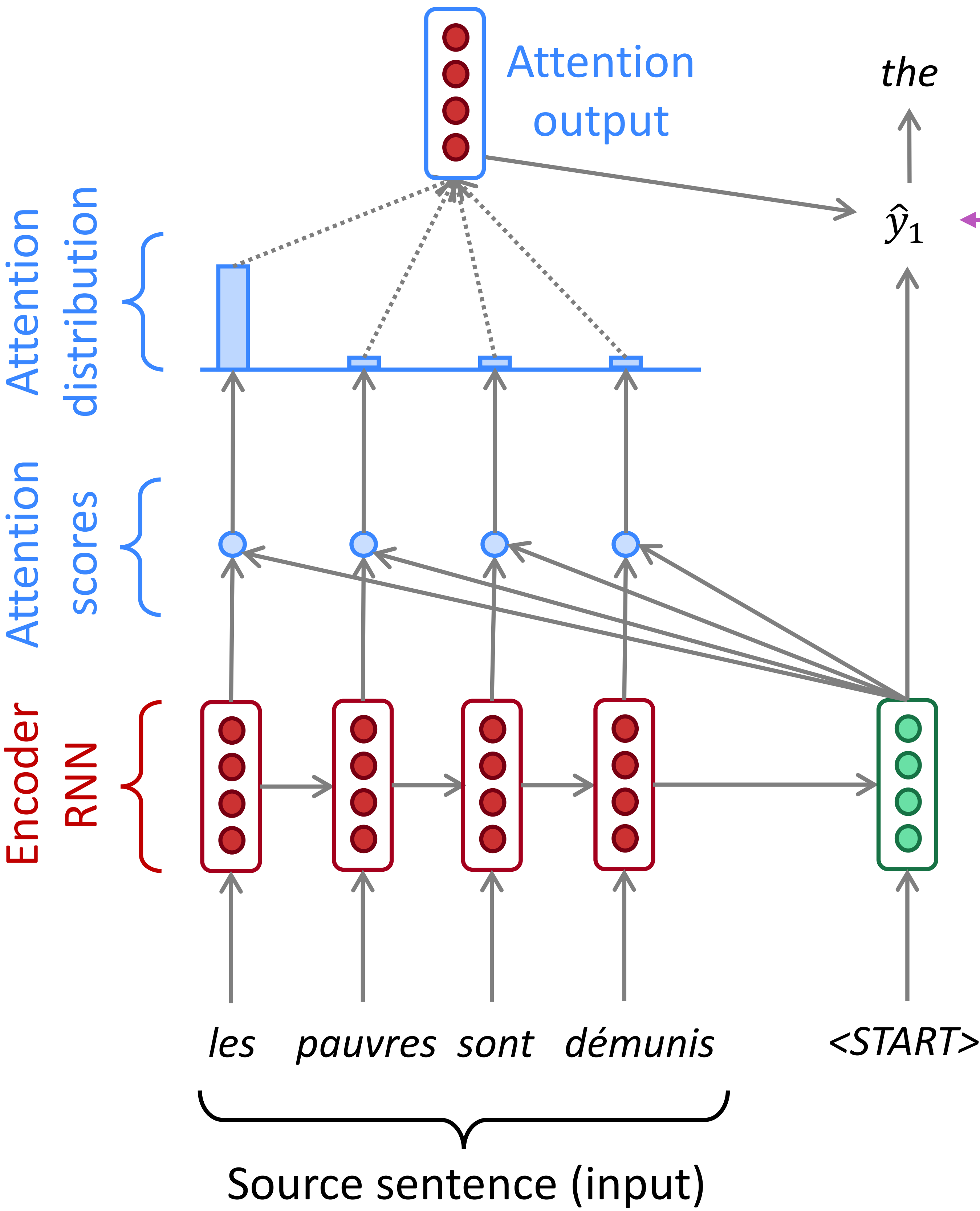
how many probabilities do we need to evaluate at each time step with a beam size of k?

what are the termination conditions for beam search?

Sequence-to-sequence: the bottleneck problem

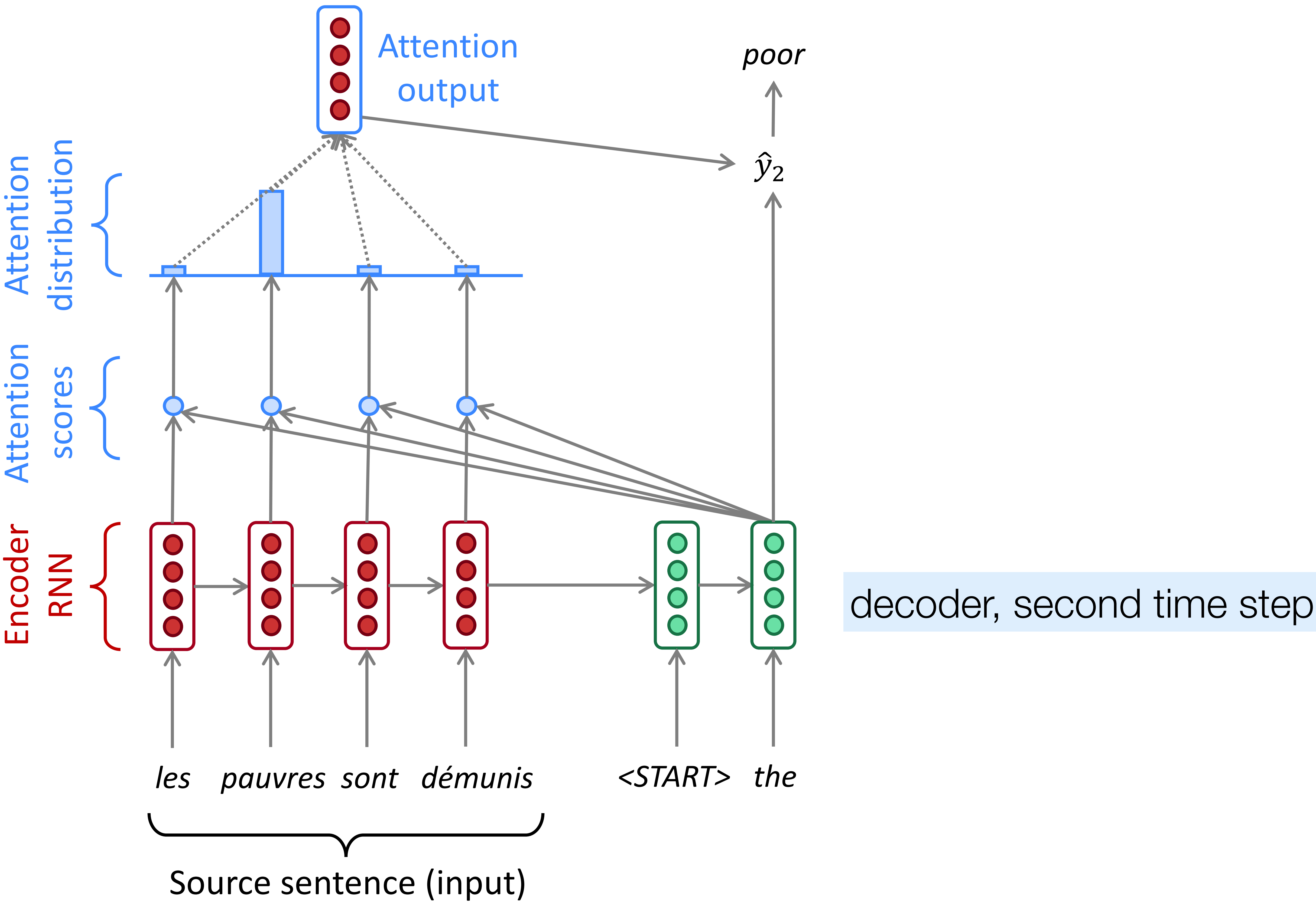


Sequence-to-sequence with attention



Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

Sequence-to-sequence with attention



BLEU
Bilingual Evaluation Understudy

N-gram overlap between machine translation output and reference translation

Compute precision for n-grams of size 1 to 4

Add brevity penalty (for too short translations)

$$\text{BLEU} = \min\left(1, \frac{\textit{output-length}}{\textit{reference-length}}\right) \left(\prod_{i=1}^{4} \textit{precision}_i\right)^{\frac{1}{4}}$$

Typically computed over the entire corpus, not single sentences

word representations

why use vectors to encode meaning?

- computing the similarity between two words (or phrases, or documents) is *extremely* useful for many NLP tasks
- Q: how **tall** is Mount Everest?

  A: The official **height** of Mount Everest is 29029 ft

all words are equally (dis)similar!

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

dot product is zero!

these vectors are **orthogonal**

how can we compute a vector representation such that the dot product correlates with word similarity?

dense word vectors

- model the meaning of a word as an **embedding** in a vector space
  - this vector space is commonly low dimensional (e.g., 100-500d).
  - what is the dimensionality of a one-hot word representation?

- embeddings are real-valued vectors (not binary or counts)

Word2vec

- Instead of **counting** how often each word *w* occurs near "*apricot*"
- Train a classifier on a binary **prediction** task:
  - Is *w* likely to show up near "*apricot*"?

- We don’t actually care about this task
  - But we'll take the learned classifier weights as the word embeddings

Setup

Let's represent words as vectors of some length (say 300), randomly initialized.

So we start with 300 * V random parameters

Over the entire training set, we'd like to adjust those word vectors such that we

- Maximize the similarity of the target word, context word pairs (t,c) drawn from the positive data
- Minimize the similarity of the (t,c) pairs drawn from the negative data.

Objective Criteria

We want to maximize…

$$\sum_{(t,c)\in +} logP(+|t,c) + \sum_{(t,c)\in -} logP(-|t,c)$$

Maximize the + label for the pairs from the positive training data, and the – label for the pairs sample from the negative data.

Focusing on one target word t:

n_i is the vector for the negative sample

$$
\begin{aligned}
L(\theta) &= \log P(+|t,c) + \sum_{i=1} \log P(-|t,n_i) \\
&= \log \sigma(c \cdot t) + \sum_{i=1}^{k} \log \sigma(-n_i \cdot t) \\
&= \log \frac{1}{1+e^{-c \cdot t}} + \sum_{i=1}^{k} \log \frac{1}{1+e^{n_i \cdot t}}
\end{aligned}
$$

you should be able to take derivatives of this as in HW2!