

# Deep Learning

## Neural Language Model

Lionel Fillatre

2025-2026

# Outline

- Introduction
- Language Model
- N-gram
- Word Embedding
- Word2Vec
- Neural Language Model
- RNN
- Sequence to Sequence
- Conclusion

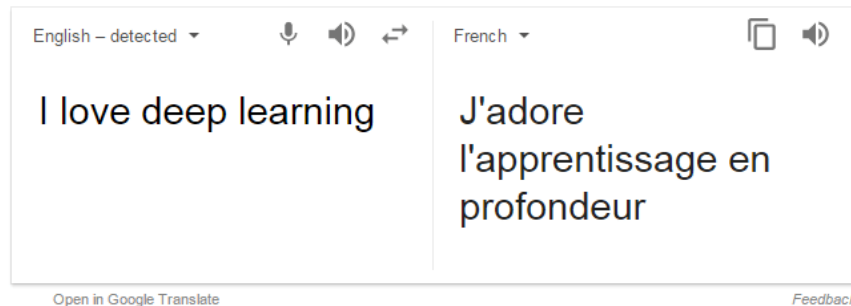
# Introduction

# Natural Language Processing (NLP) Applications

- Sentiment Analysis
- Email Filters
- Voice Recognition
- Information Extraction
- Translation
- Large language model-based chatbot
- ...



	Sentence	Class	index
0	So there is no way for me to plug it in here i...	0	0
1	Good case, Excellent value.	1	1
2	Great for the jawbone.	1	2
3	Tied to charger for conversations lasting more...	0	3
4	The mic is great.	1	4
5	I have to jiggle the plug to get it to line up...	0	5
6	If you have several dozen or several hundred c...	0	6
7	If you are Razr owner...you must have this!	1	7
8	Needless to say, I wasted my money.	0	8
9	What a waste of money and time!	0	9
10	And the sound quality is great.	1	10



# Language Model

# Uses of Language Models

- Speech recognition
  - “I ate a cherry” is a more likely sentence than “Eye eight uh Jerry”
- OCR & Handwriting recognition
  - More probable sentences are more likely correct readings.
- Machine translation
  - More likely sentences are probably better translations.
- Generation
  - More likely sentences are probably better Natural Language generations.
- Context sensitive spelling correction
  - “Their are problems wit this sentence.”

# You use Language Models every day!

Connais-tu l'Université Côte d'Azur ?



Oui, je connais l'Université Côte d'Azur (souvent abrégée "UCA" ou "UniCA"). Voici un aperçu de ce qu'elle est, de ses caractéristiques, forces et contexte — si tu veux, je peux t'en dire plus dans un domaine précis (recherche, formations, vie étudiante, etc.).

## Présentation générale

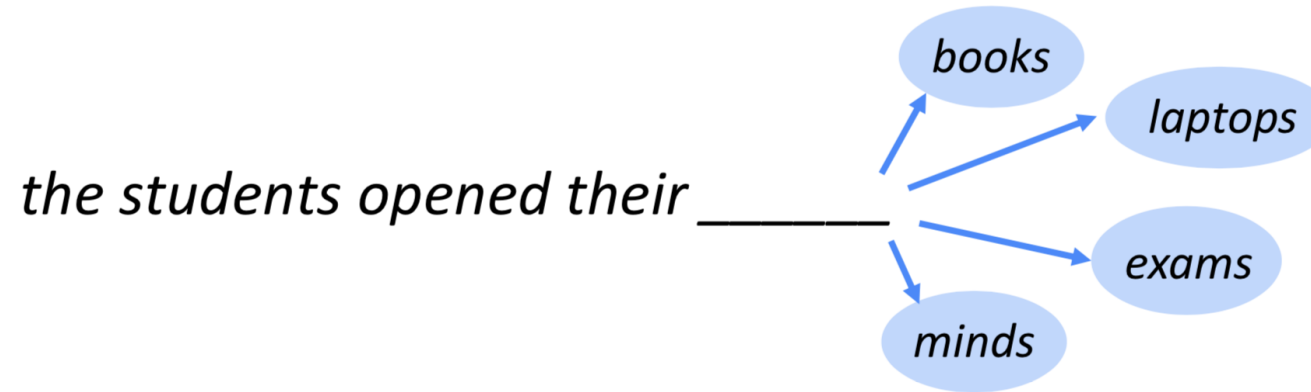
- L'Université Côte d'Azur est une **université publique de recherche** située à Nice, avec des campus répartis dans la région Provence-Alpes-Côte d'Azur : Nice, Cannes, Grasse, Menton, Sophia Antipolis... [Wikipédia +2](#)
- Elle a été officiellement créée sous sa forme actuelle en **2019**, succédant à l'Université de Nice Sophia-Antipolis. [Wikipédia +2](#)
- Elle accueille environ **30 000-35 000 étudiant·e·s**, dont une proportion non négligeable d'étudiants internationaux (~20 %) [Université Côte d'Azur +2](#)

## Statut, structuration, distinctions

- Elle dispose d'un **statut d'établissement public scientifique, culturel et professionnel expérimental** (EPSCP expérimental) depuis sa refonte. [Wikipédia +1](#)
- En juillet 2024, elle a obtenu le statut de **Grand établissement**. [Université Côte d'Azur +2](#)
- Elle a obtenu le label national **Initiative d'Excellence (IDEX)**, ce qui la place parmi les universités françaises "intensives en recherche" avec un fort rayonnement. [FUN MOOC +3](#)

# Language Modeling

- Language Modeling is the task of predicting what word comes next



- Given a sequence of words  $w_1, w_2, \dots, w_t$ , compute the probability distribution of the next word  $w_{t+1}$

$$\Pr(w_{t+1} | w_t, w_{t-1}, \dots, w_1)$$

where  $w_{t+1}$  can be any word in the vocabulary  $V = \{v_1, v_2, \dots, v_{|V|}\}$

- A system that does this is called a **Language Model**.



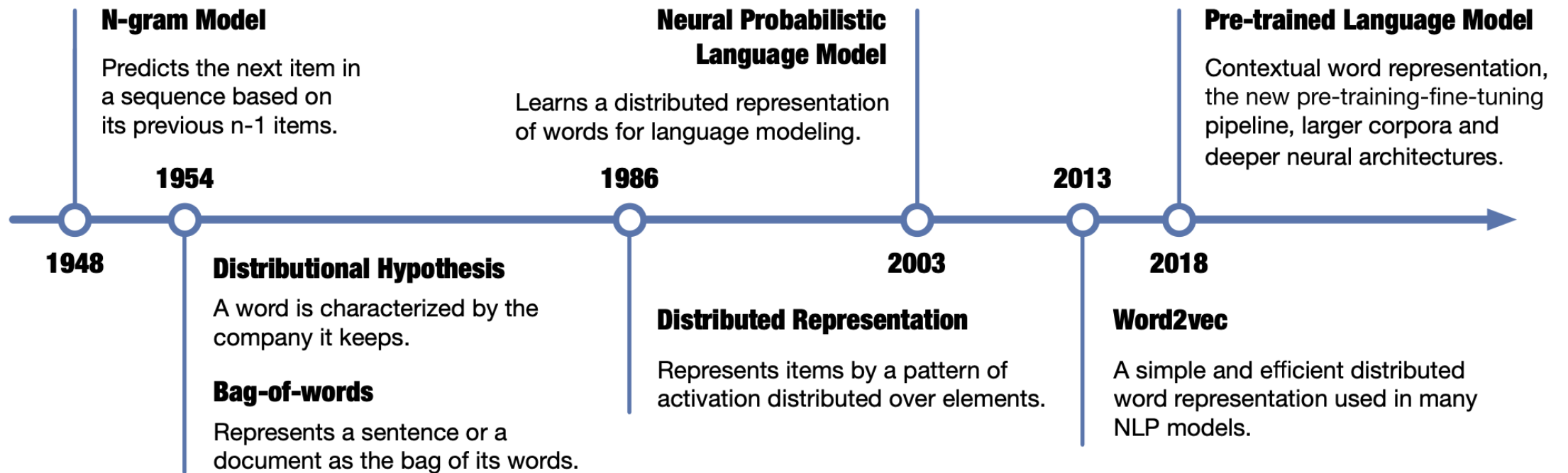
# Language Modeling

- You can also think of a Language Model as a system that assigns probability to a piece of text.
- For example, if we have some text  $w_1, w_2, \dots, w_T$ , then the probability of this text (according to the Language Model) is:

$$\Pr(w_1, w_2, \dots, w_T) = \prod_{t=1}^T \Pr(w_t | w_{t-1}, \dots, w_1)$$

- Example:  $\Pr(\text{its water was so transparent}) = \Pr(\text{its}) * \Pr(\text{water} | \text{its}) * \Pr(\text{was} | \text{its water}) * \Pr(\text{so} | \text{its water was}) * \Pr(\text{transparent} | \text{its water was so})$
- Language Modeling provides  $\Pr(w_t | w_{t-1}, \dots, w_1)$

# Timeline for the representation learning in NLP



# N-Gram

# N-gram Language Models

- Question: How to learn a Language Model?
- Answer (pre-Deep Learning): Learn a  $n$ -gram Language Model!
- Definition: A  $n$ -gram is a chunk of  $n$  consecutive words.
  - unigrams: “the”, “students”, “opened”, “their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - 4-grams: “the students opened their”
- Idea: Collect statistics about how frequent different  $n$ -grams are, and use these to predict next word.

# N-Gram Models

- Estimate probability of each word given prior context.
  - $P(\text{phone} \mid \text{Please turn off your cell})$
- Number of parameters required grows exponentially with the number of words of prior context.
- An  $N$ -gram model uses only  $N-1$  words of prior context.
  - Unigram:  $P(\text{phone})$
  - Bigram:  $P(\text{phone} \mid \text{cell})$
  - Trigram:  $P(\text{phone} \mid \text{your cell})$
- Markov model
  - The Markov assumption is the presumption that the future behavior of a dynamical system only depends on its recent history.
  - In particular, in a  $k$ th-order Markov model, the next state only depends on the  $k$  most recent states, therefore an  $N$ -gram model is a  $(N-1)$ -order Markov model.

# N-gram Language Models

- First we make a simplifying assumption: preceding  $n - 1$  words

$$\Pr(w_{t+1} | w_t, \dots, w_1) = \Pr(w_{t+1} | w_t, \dots, w_{t-n+2})$$

- Conditional probabilities:

$$\Pr(w_{t+1} | w_t, \dots, w_{t-n+2}) = \frac{\Pr(w_{t+1}, w_t, \dots, w_{t-n+2})}{\Pr(w_t, \dots, w_{t-n+2})}$$

- **Question:** How do we get these  $n$ -gram and  $(n - 1)$ -gram probabilities?
- **Answer:** By counting them in some large corpus of text!
  - Statistical approximation:

$$\Pr(w_{t+1} | w_t, \dots, w_{t-n+2}) \approx \frac{\text{count}(w_{t+1}, w_t, \dots, w_{t-n+2})}{\text{count}(w_t, \dots, w_{t-n+2})}$$

# Word Embedding

# Representing words as discrete symbols

- In traditional NLP, we regard words as discrete symbols:
  - hotel, conference, motel – a localist representation

- Words can be represented by one-hot vectors:

- motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

- hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

- One-hot encoding

- Vector dimension = number of words in vocabulary  $V$  (e.g., 500,000)
  - The number of words in the vocabulary is  $|V|$
  - The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word.

Vocabulary:  
Man, woman, boy,  
girl, prince,  
princess, queen,  
king, monarch



	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

Each word gets  
a 1x9 vector  
representation



# Problem with words as discrete symbols

- **Example:** in web search, if user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”.
- But:
  - motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
  - hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
  - These two vectors are orthogonal.
- There is no natural notion of **similarity** for one-hot vectors!
- **Solution:**
  - Learn to encode similarity in the vectors themselves.

# Word vectors

- We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts

Try to build a lower dimensional embedding

Vocabulary:  
Man, woman, boy,  
girl, prince,  
princess, queen,  
king, monarch



	Femininity	Youth	Royalty
Man	0	0	0
Woman	1	0	0
Boy	0	1	0
Girl	1	1	0
Prince	0	1	1
Princess	1	1	1
Queen	1	0	1
King	0	0	1
Monarch	0.5	0.5	1

[@shane\\_a\\_lynn](#) | [@TeamEdgeTier](#)

Each word gets a  
1x3 vector

Similar words...  
similar vectors

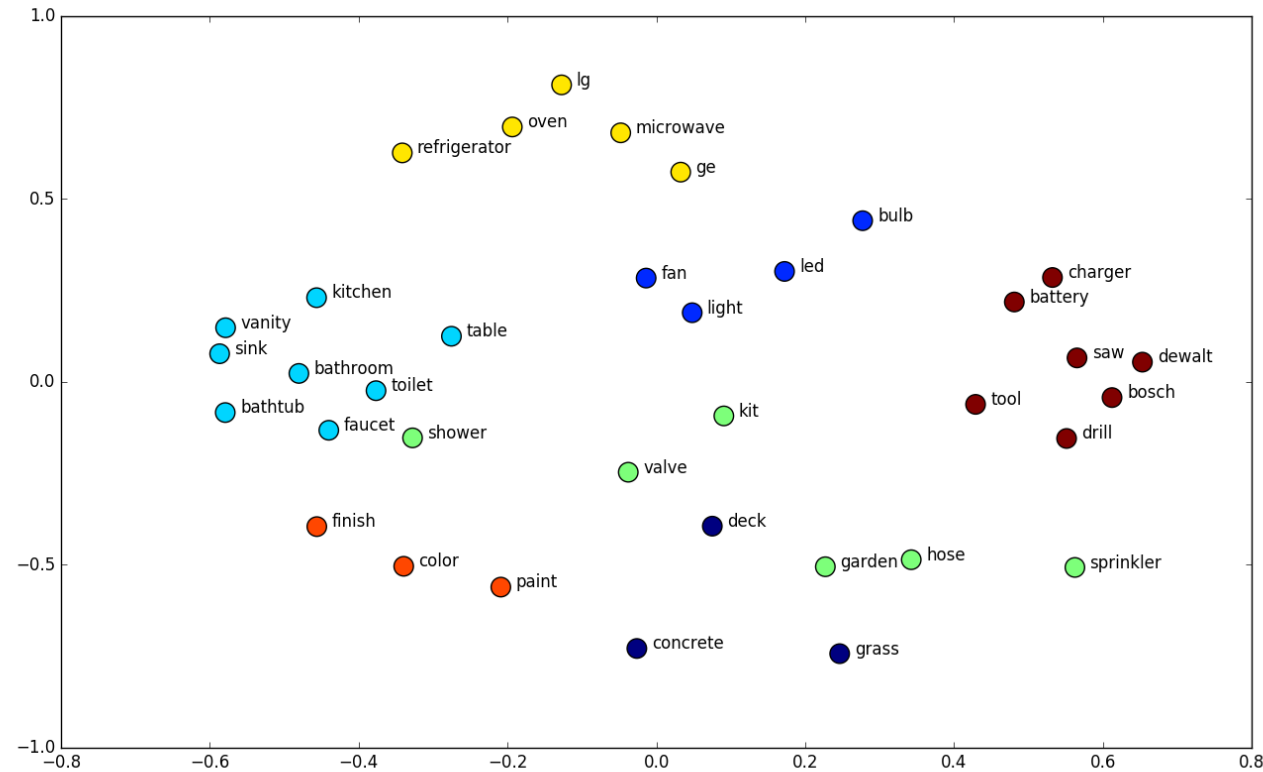
*banking* =

0.286  
0.792  
-0.177  
-0.107  
0.109  
-0.542  
0.349  
0.271

- Note: word vectors are sometimes called word embeddings or word representations. They are a distributed representation.

# Representation Space

- Example 2D word embedding space, where similar words are found in similar locations



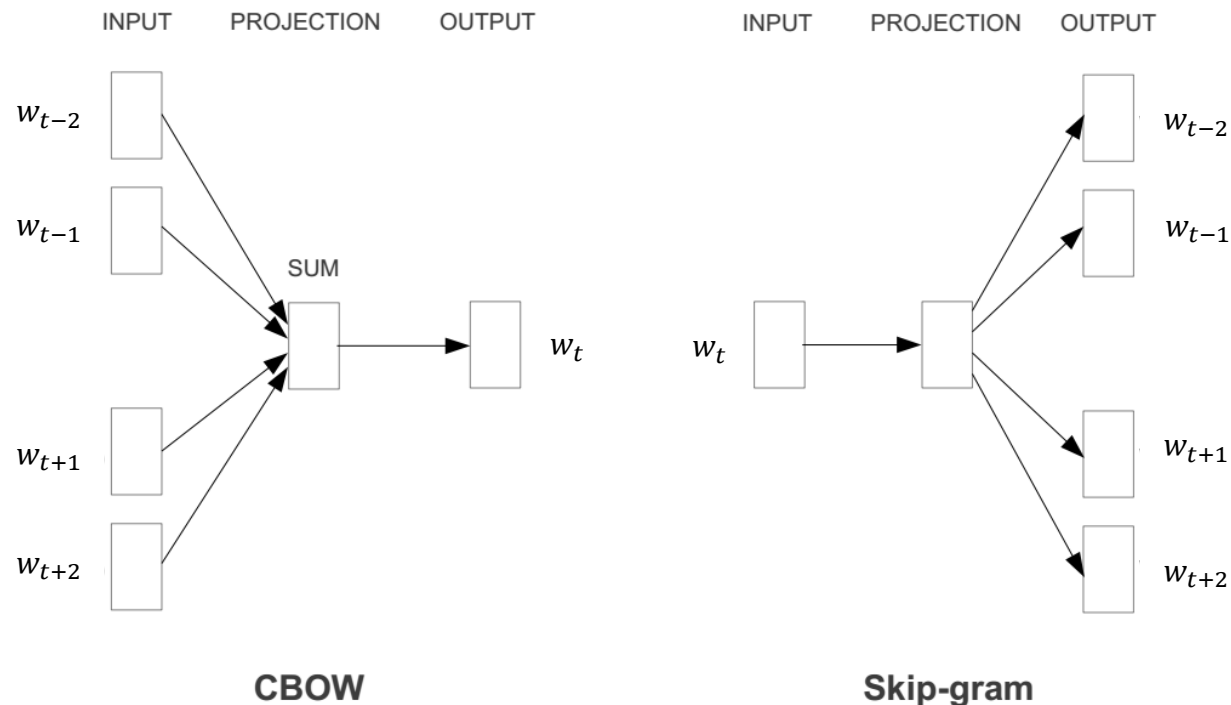
# Word2Vec

# Word2vec: Overview

- Word2vec (Mikolov et al. 2013) is a framework for learning word vectors
- **Principle:**
  - « A word is characterized by the company it keeps ». (J. R. Firth, 1957)
  - Words that are used and occur in the same contexts tend to purport similar meanings.
- **How to do it:**
  - We have a large corpus of text
  - Every word in a fixed vocabulary is represented by a vector
  - Go through each position  $t$  in the text, which has a center word  $c$  and context (“outside”) words  $o$
  - Use the similarity of the word vectors for  $c$  and  $o$  to calculate the probability of  $o$  given  $c$  (or vice versa)
  - Keep adjusting the word vectors to maximize this probability

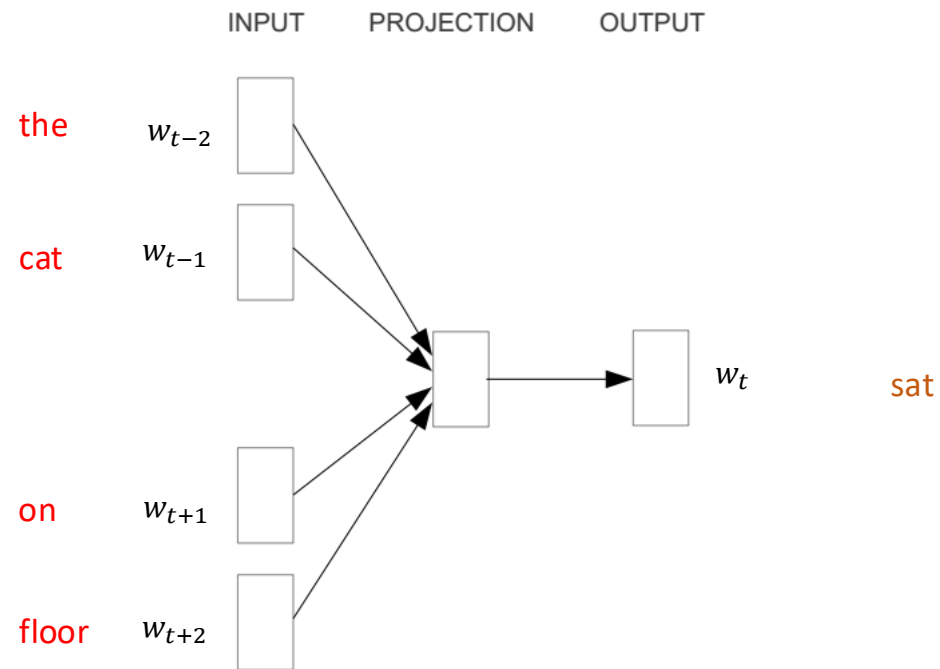
# Represent the meaning of word – word2vec

- 2 basic neural network models:
  - Continuous Bag of Word (CBOW): use a window of word to predict the middle word
  - Skip-gram (SG): use a word to predict the surrounding ones in window.

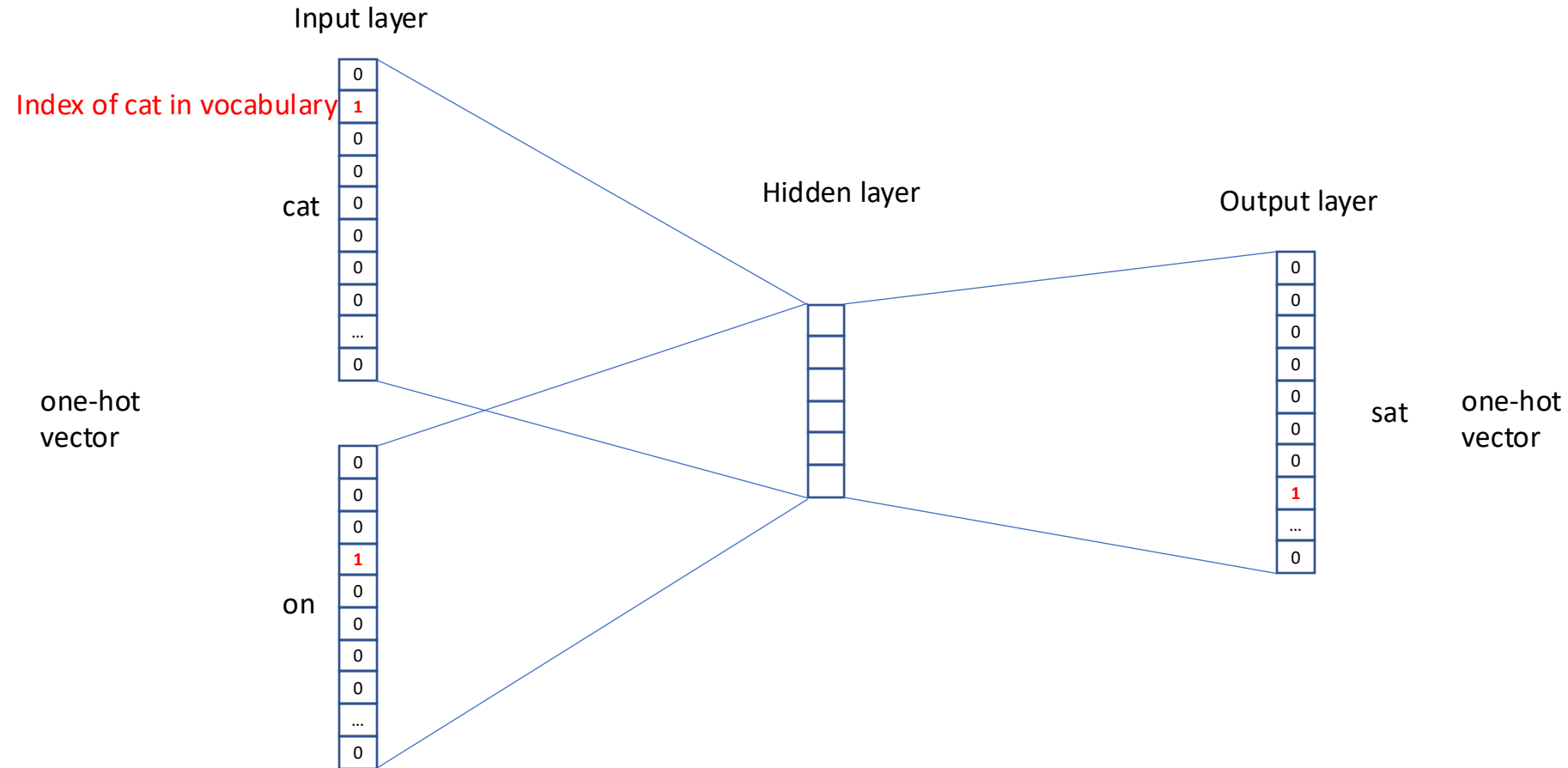


# Word2vec – Continuous Bag of Word

- E.g. “The cat sat on floor”
  - Window size = 2

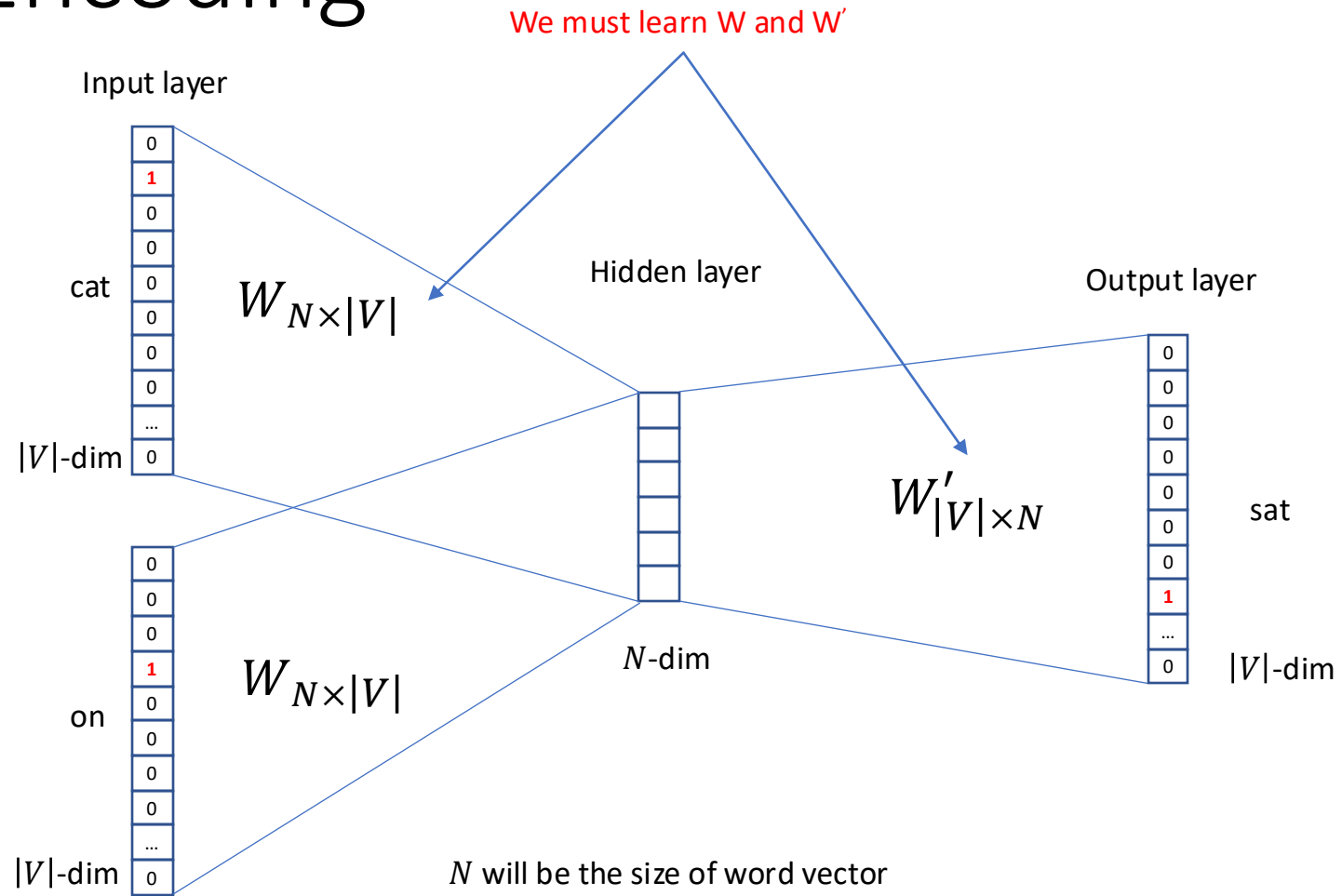


# Input Encoding (window size = 1)

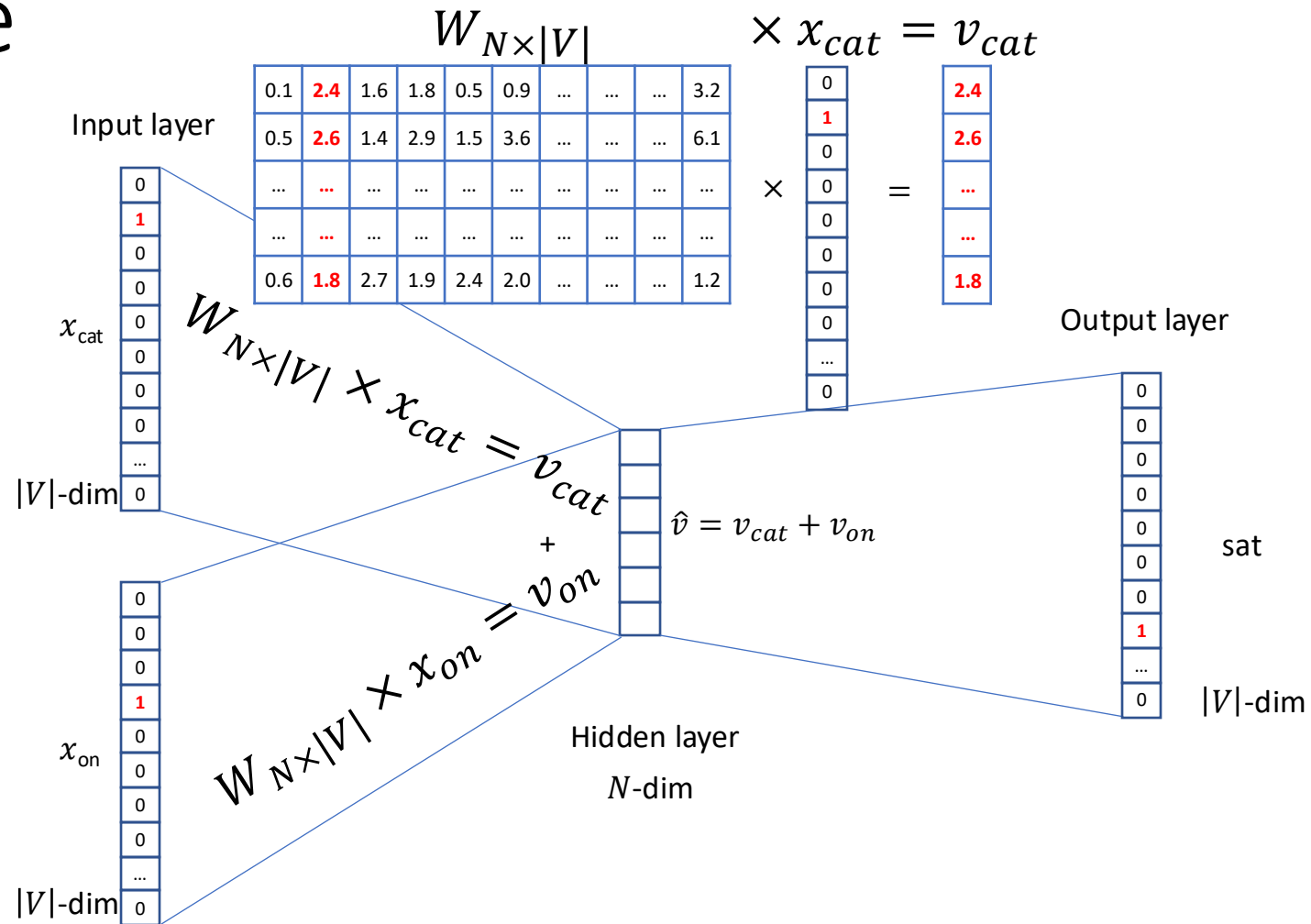




# Matrix Encoding

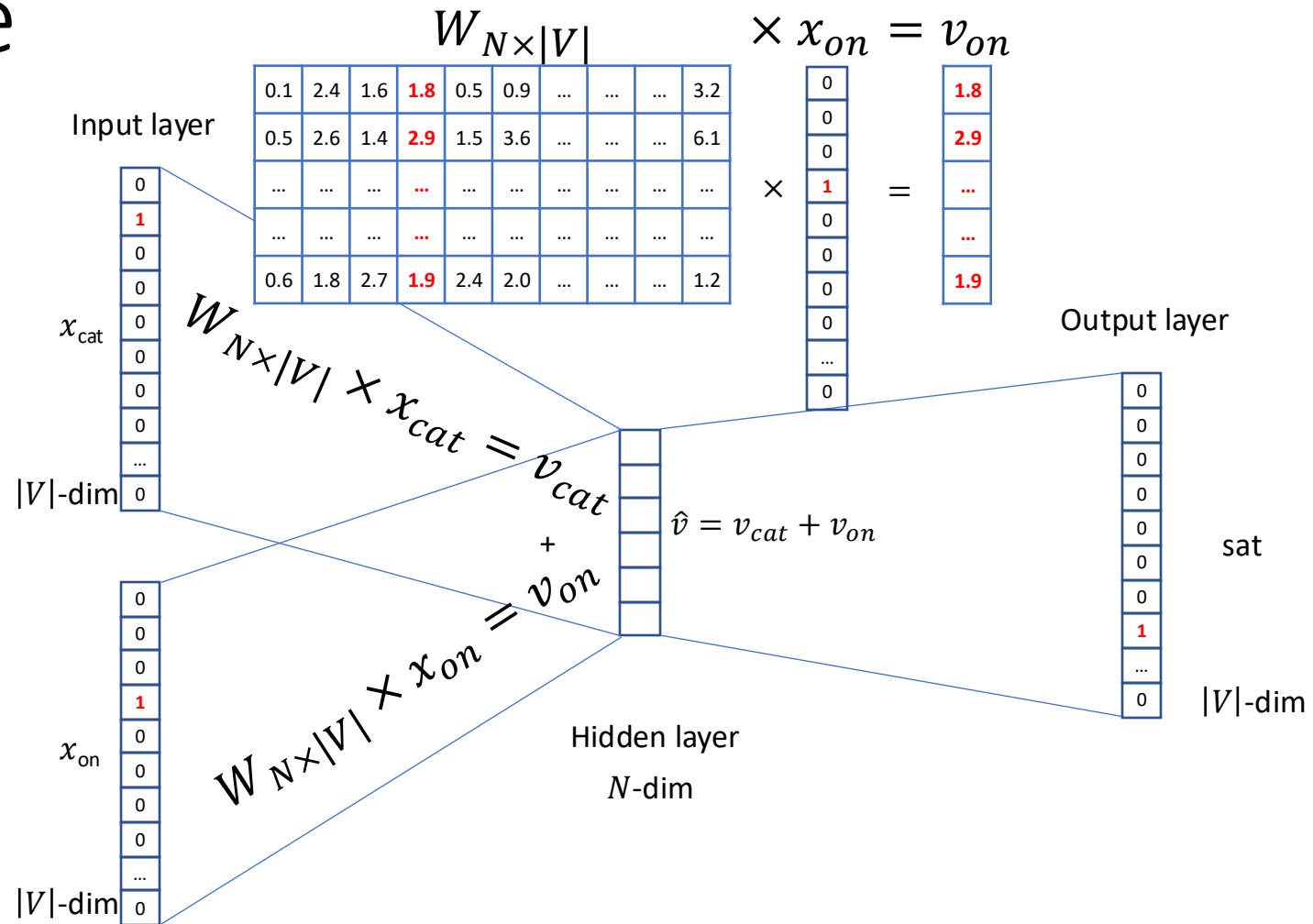


# Example

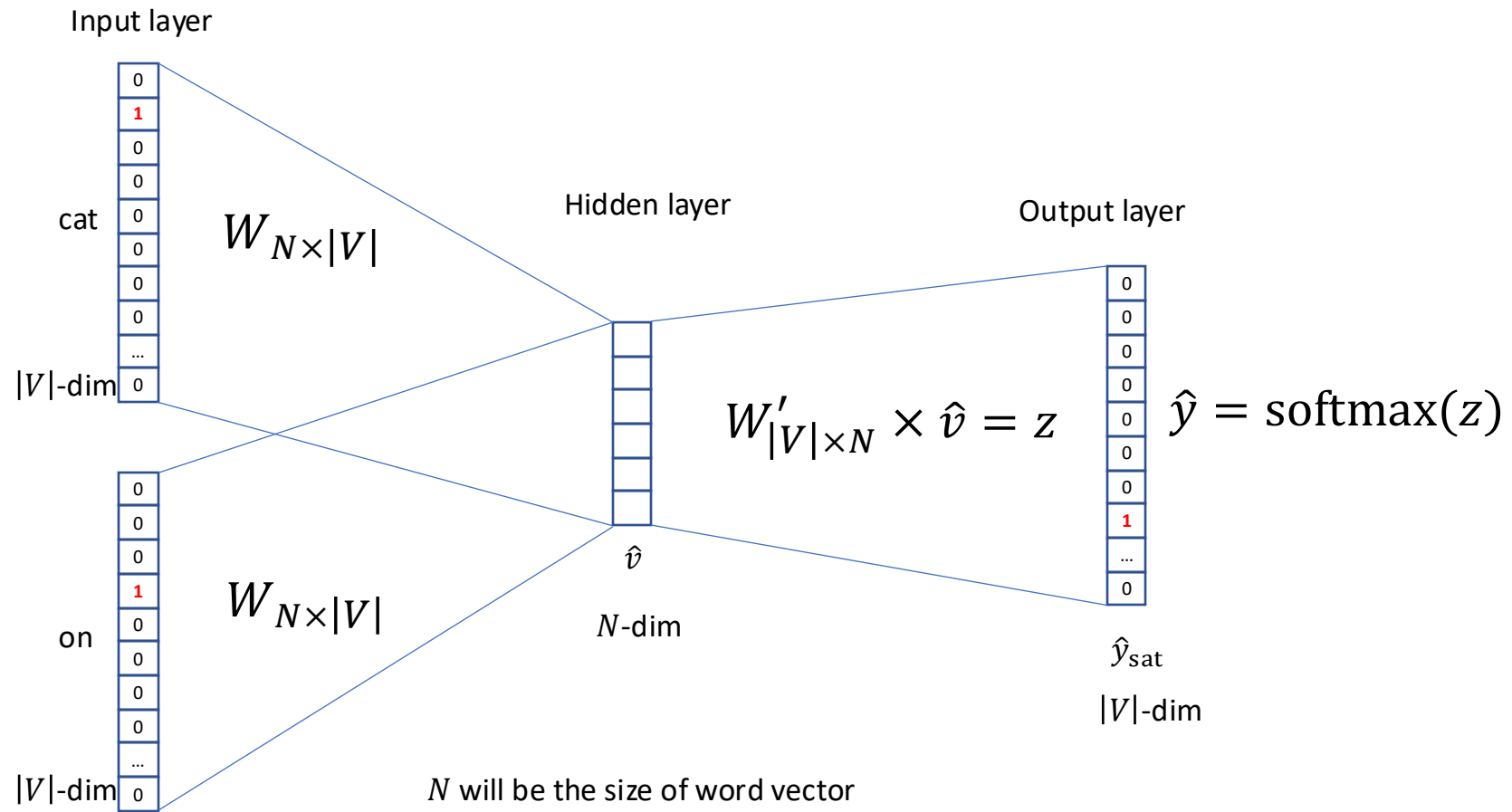


Note that we can take  $\hat{v} = \frac{v_{cat} + v_{on}}{2}$  instead of  $\hat{v} = v_{cat} + v_{on}$

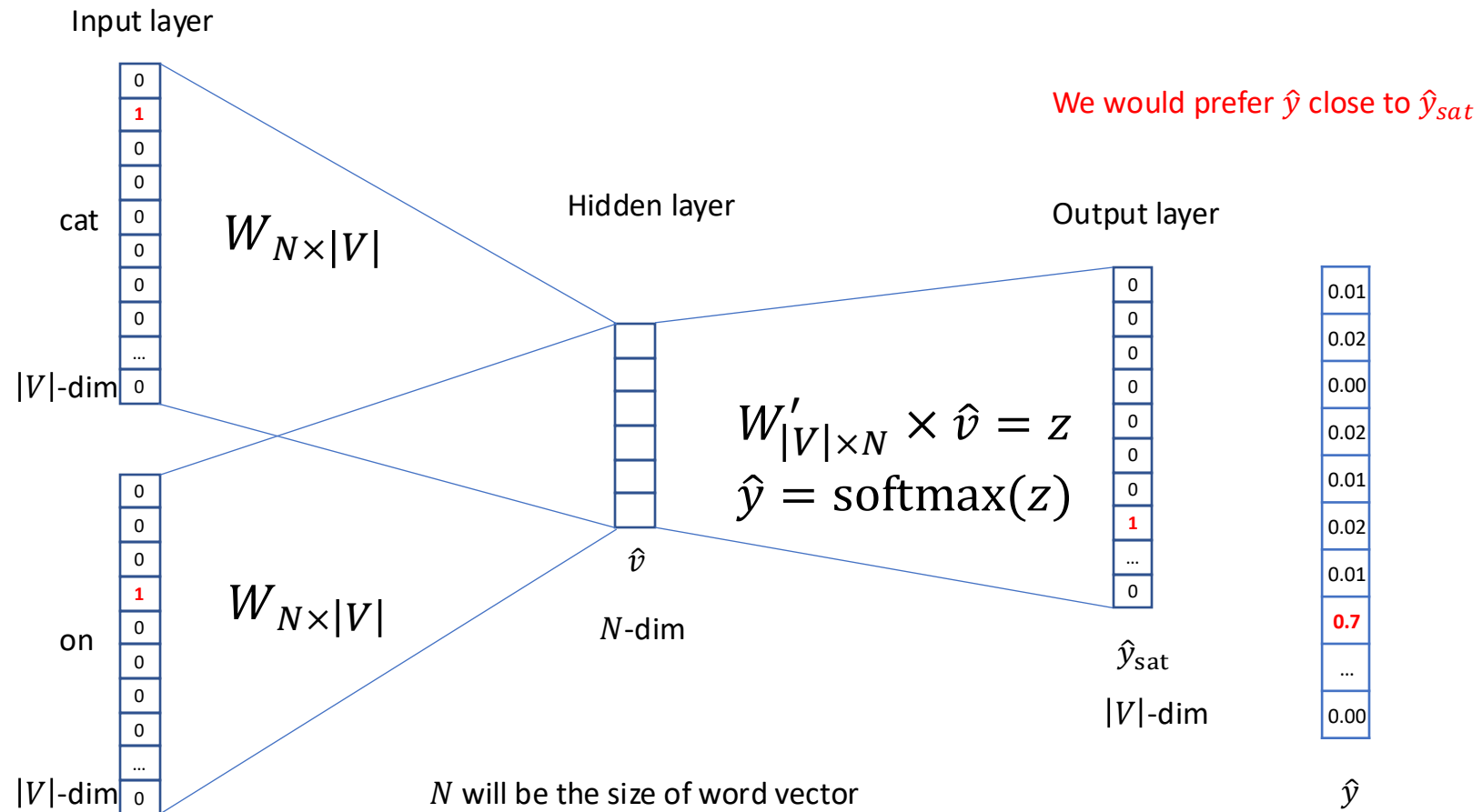
# Example



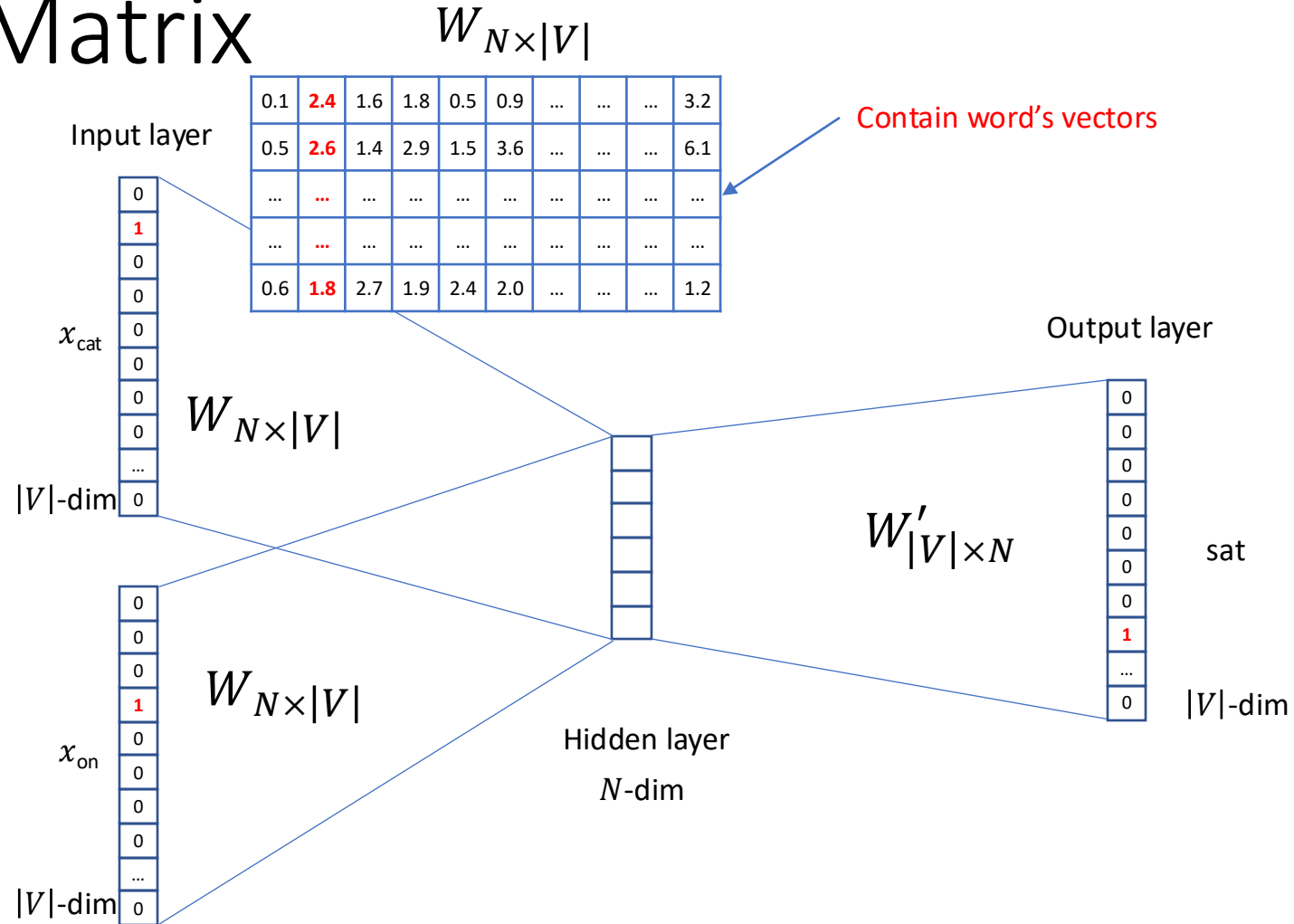
# Output Layer



# Output



# Word's Matrix



We can consider either  $W$  or  $W'$  as the word's representation, or even take the average.

# Output: Similarity Computation

- **Assumption:** Computed using the dot product of the two vectors
- **Model:** To convert a similarity to a probability, use softmax

$$\Pr(w_k | w_{k+j}, -m \leq j \leq m, j \neq 0) = \Pr(w_k | \hat{v}_k) = \frac{\exp(\hat{v}_k^T u_k)}{\sum_{j=1}^{|V|} \exp(\hat{v}_k^T u_j)}$$

- Reminder:  $\hat{v}_k$  is the vector representation (hidden vector) of the context

$$c_k = \{w_{k+j}, -m \leq j \leq m, j \neq 0\}$$

surrounding the word  $w_k$

- Reminder:  $u_k$  is the vector representation of the word  $w_k$
- In practice, use negative sampling
  - Too many words in the denominator
  - The denominator is only computed for a few words

# Word2vec: objective function

- Let  $\theta$  be the set of parameters  $W$  and  $W'$
- The objective function is the cross-entropy
- Likelihood is equivalent to cross-entropy when using one-hot encoding:

$$L(\theta) = \prod_{t=1}^T \Pr(w_t | w_{t+j}, -m \leq j \leq m, j \neq 0; \theta)$$

$$-\log L(\theta) = -\sum_{t=1}^T \log \Pr(w_t | w_{t+j}, -m \leq j \leq m, j \neq 0; \theta)$$

$$-\log L(\theta) = -\sum_{t=1}^T \sum_{x \in V} \log \Pr(w_t | w_{t+j}, -m \leq j \leq m, j \neq 0; \theta) \Pr(w_t = x)$$



# Some interesting results

## Word Analogies

Test for linear relationships, examined by Mikolov et al. (2014)

man:woman :: king:?

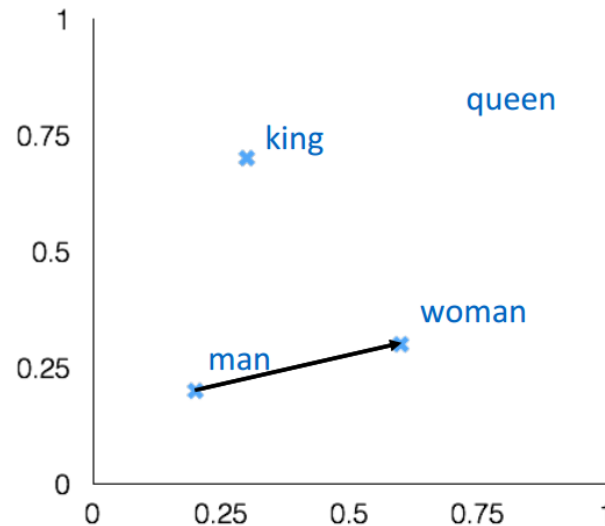
+ king [ 0.30 0.70 ]

- man [ 0.20 0.20 ]

+ woman [ 0.60 0.30 ]

---

queen [ 0.70 0.80 ]



# Neural Language Model

# How to build a neural Language Model?

- Remind the Language Modeling task:

- Input: sequence of words  $x^{(1)}, x^{(2)}, \dots, x^{(t)}$ ,
- Output: probability distribution of the next word

$$\Pr \left( x^{(t+1)} \mid x^{(t)}, x^{(t-1)}, \dots, x^{(1)} \right)$$

- We can try a window-based neural model (as for word embedding)

# A fixed-window neural Language Model (one hidden layer)

- Output distribution

$$\hat{y} = \text{softmax}(Uh + b_2)$$

- Hidden layer

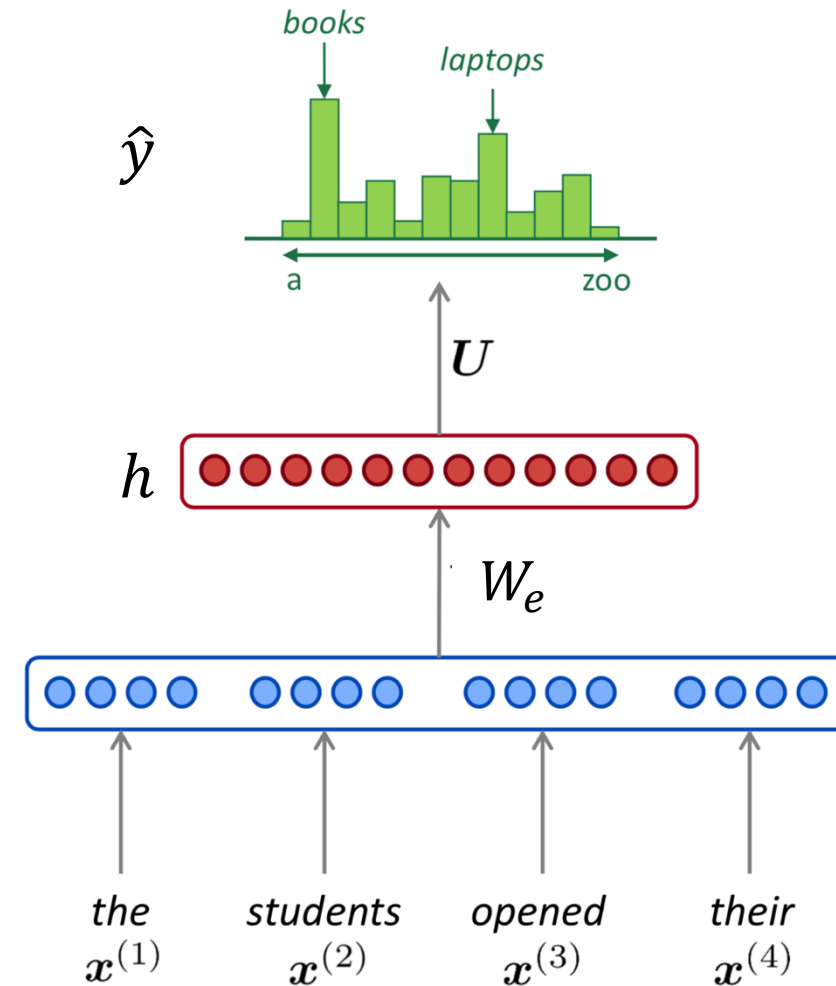
$$h = f(W_e e + b_1)$$

- Concatenated word embeddings

$$e = [e^{(1)}, e^{(2)}, e^{(3)}, e^{(4)}, e^{(5)}]$$

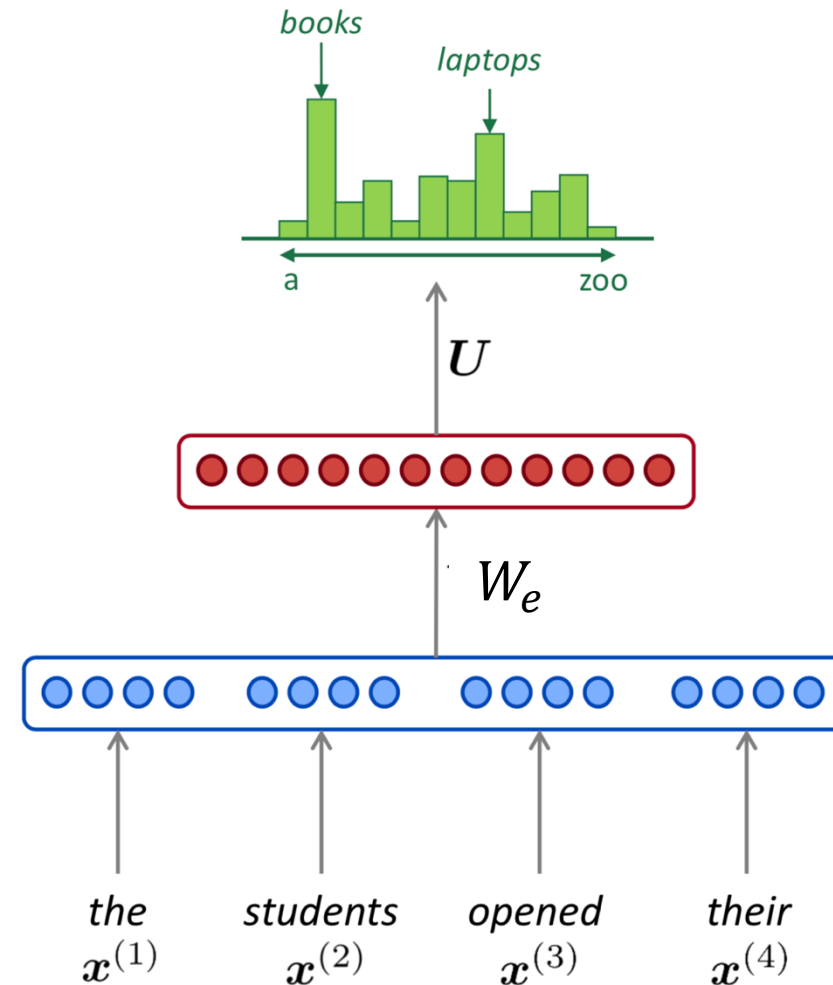
- Words / one-hot vectors

$$x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, x^{(5)}$$



# A fixed-window neural Language Model

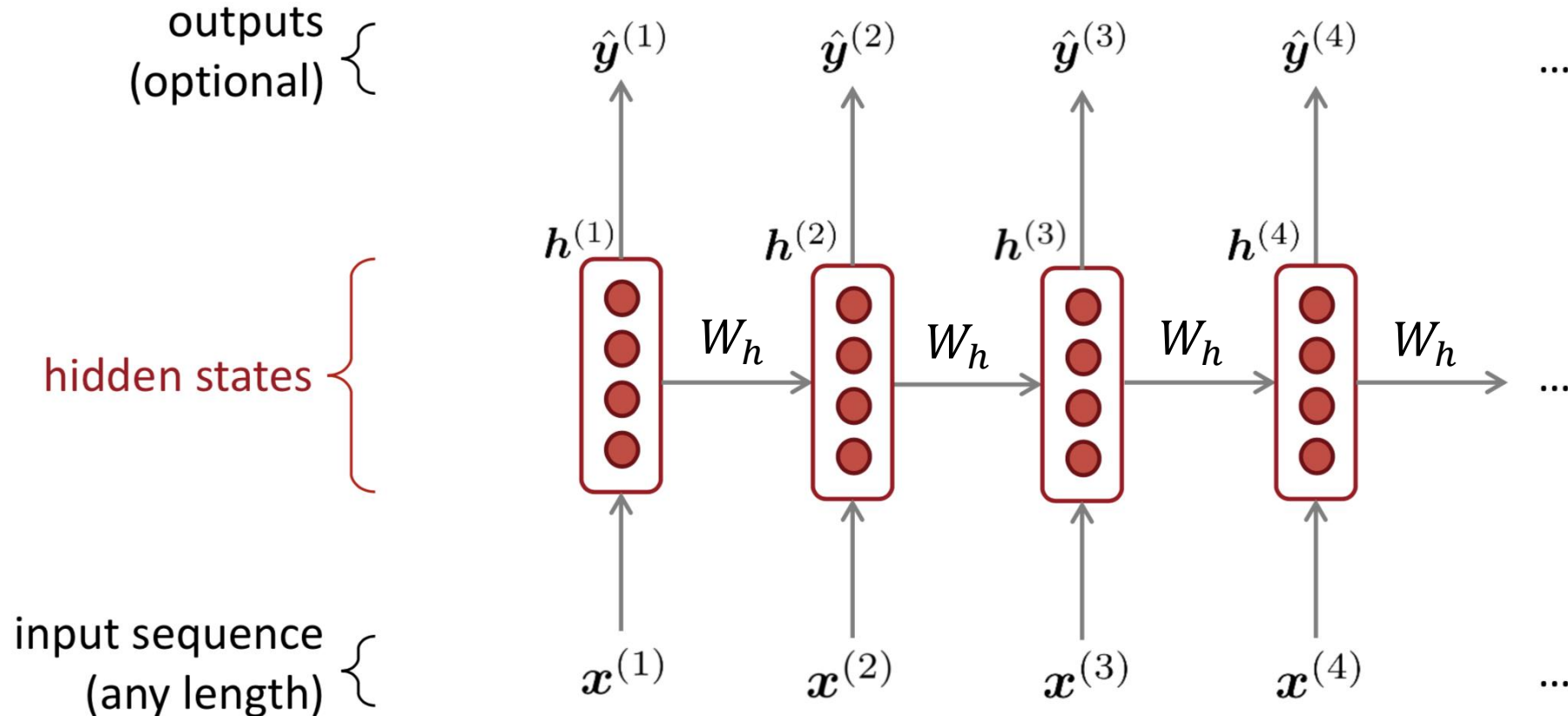
- Improvements over  $n$ -gram LM:
  - No sparsity problem
  - Don't need to store all observed  $n$ -grams
- Remaining problems:
  - Fixed window is too small
  - Enlarging window enlarges  $W_e$
  - Window can never be large enough!
  - $x^{(1)}$  and  $x^{(2)}$  are multiplied by completely different weights in  $W_e$ 
    - ⇒ No symmetry in how the inputs are processed.
  - We need a neural architecture that can process any length input



# Recurrent Neural Networks (RNN)

# Recurrent Neural Networks (RNN)

- Core idea: Apply the same weights repeatedly



$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$

# A RNN Language Model

- Output distribution

$$\hat{y} = \text{softmax}(Uh^{(t)} + b_2) \in \mathbb{R}^{|V|}$$

- Hidden states

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$$

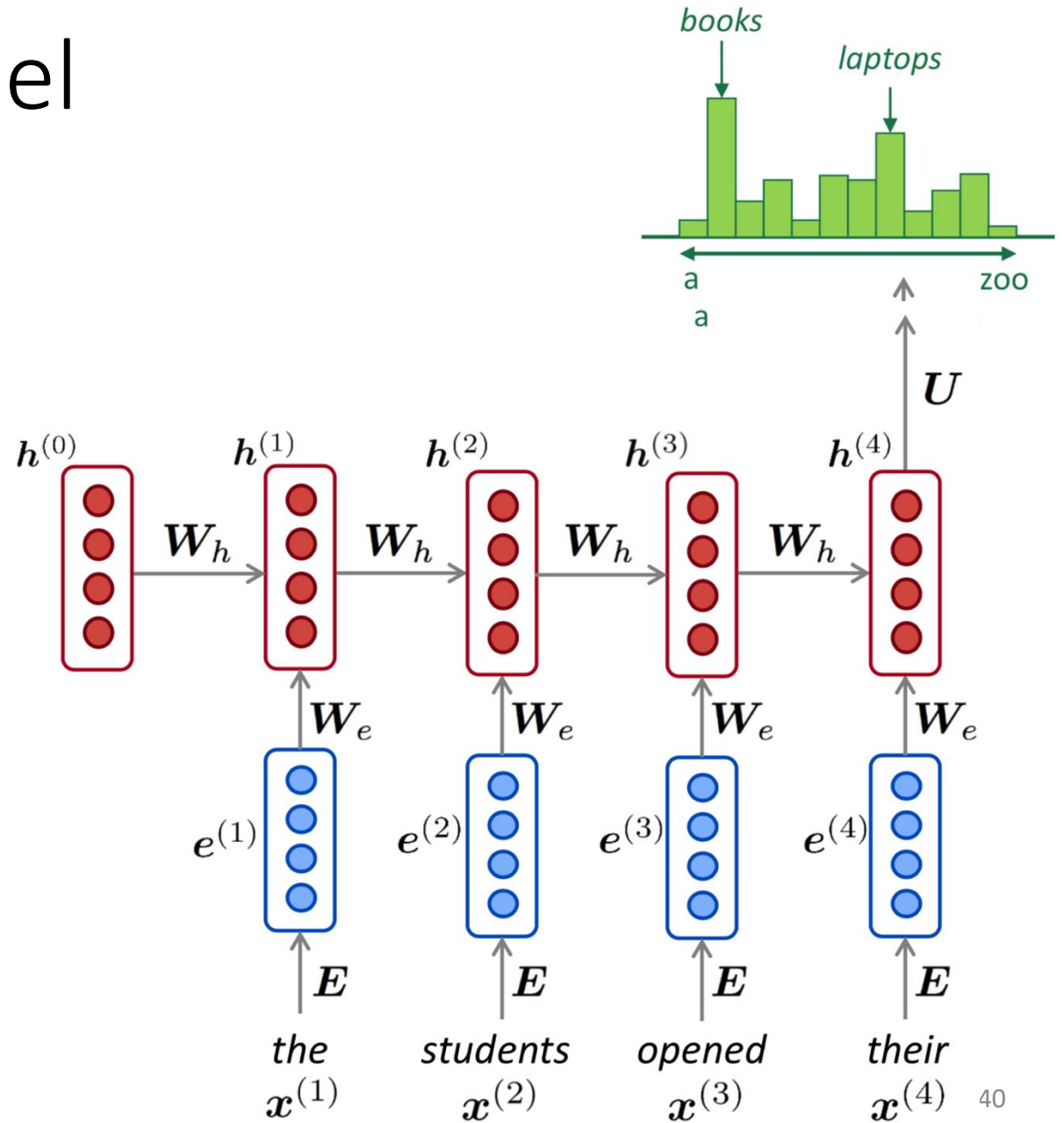
$h^{(0)}$  is the initial hidden state

- Word embeddings

$$e^{(t)} = E x^{(t)}$$

- Words as one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$





# A RNN Language Model

- RNN Advantages:
  - Can process any length input
  - Computation for step  $t$  can (in theory) use information from many steps back
  - Model size doesn't increase for longer input
  - Same weights applied on every timestep, so there is symmetry in how inputs are processed.
- RNN Disadvantages:
  - Recurrent computation is slow
  - In practice, difficult to access information from many steps back

# Training a RNN Language Model

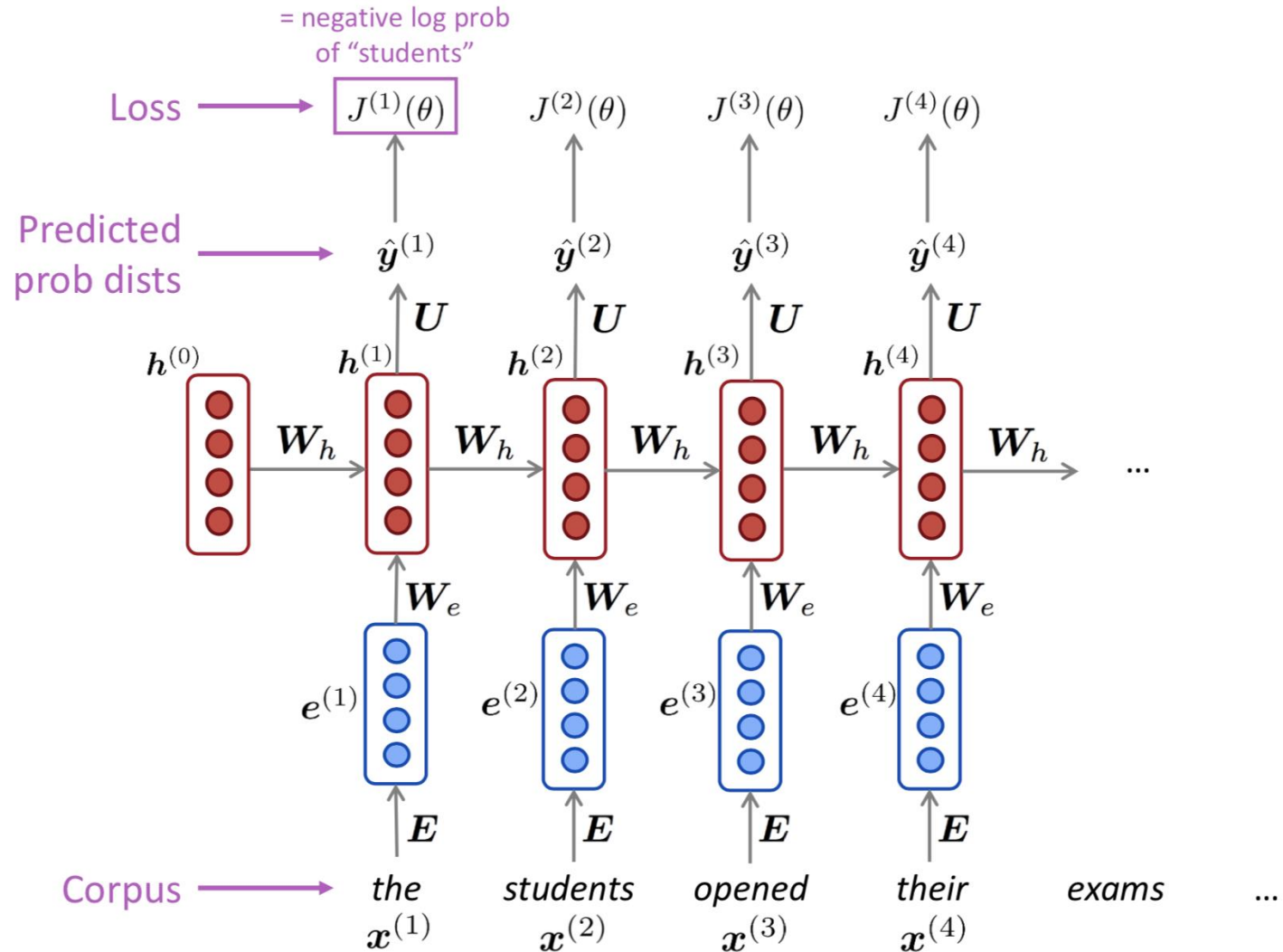
- Get a big corpus of text which is a sequence of words  $x^{(1)}, x^{(2)}, \dots, x^{(T)}$
- Feed into RNN-LM; compute output distribution  $\hat{y}^{(t)}$  for every step  $t$ .
  - i.e. predict probability dist of every word, given words so far
- Loss function on step  $t$  is cross-entropy between predicted probability distribution  $\hat{y}^{(t)}$ , and the true next word  $y^{(t)}$  (one-hot encoding for  $x^{(t+1)}$ ):

$$J^{(t)}(\theta) = \text{crossentropy}(y^{(t)}, \hat{y}^{(t)}) = - \sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = - \log \hat{y}_{x_{t+1}}^{(t)}$$

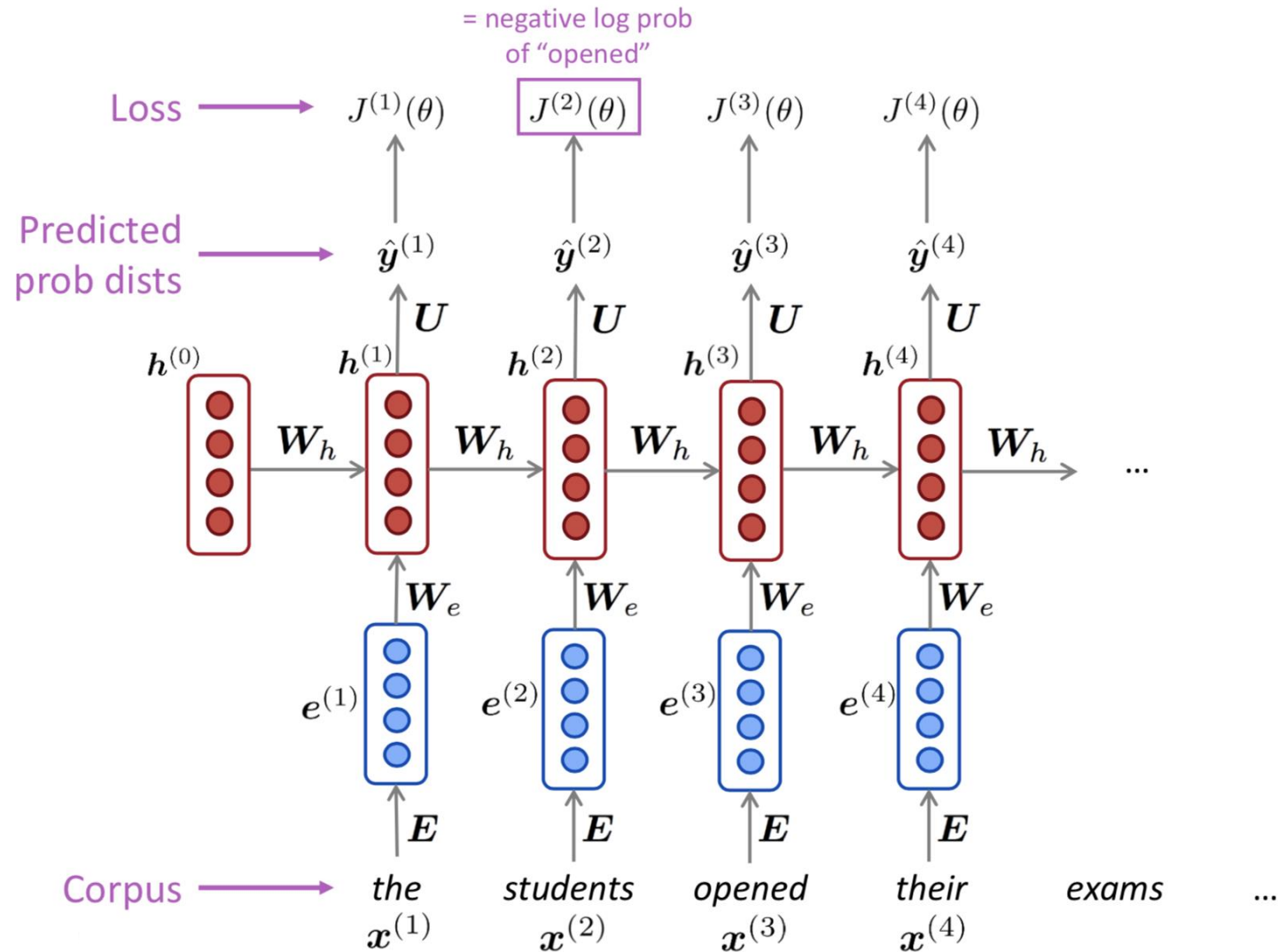
- Average this to get overall loss for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^T - \log \hat{y}_{x_{t+1}}^{(t)}$$

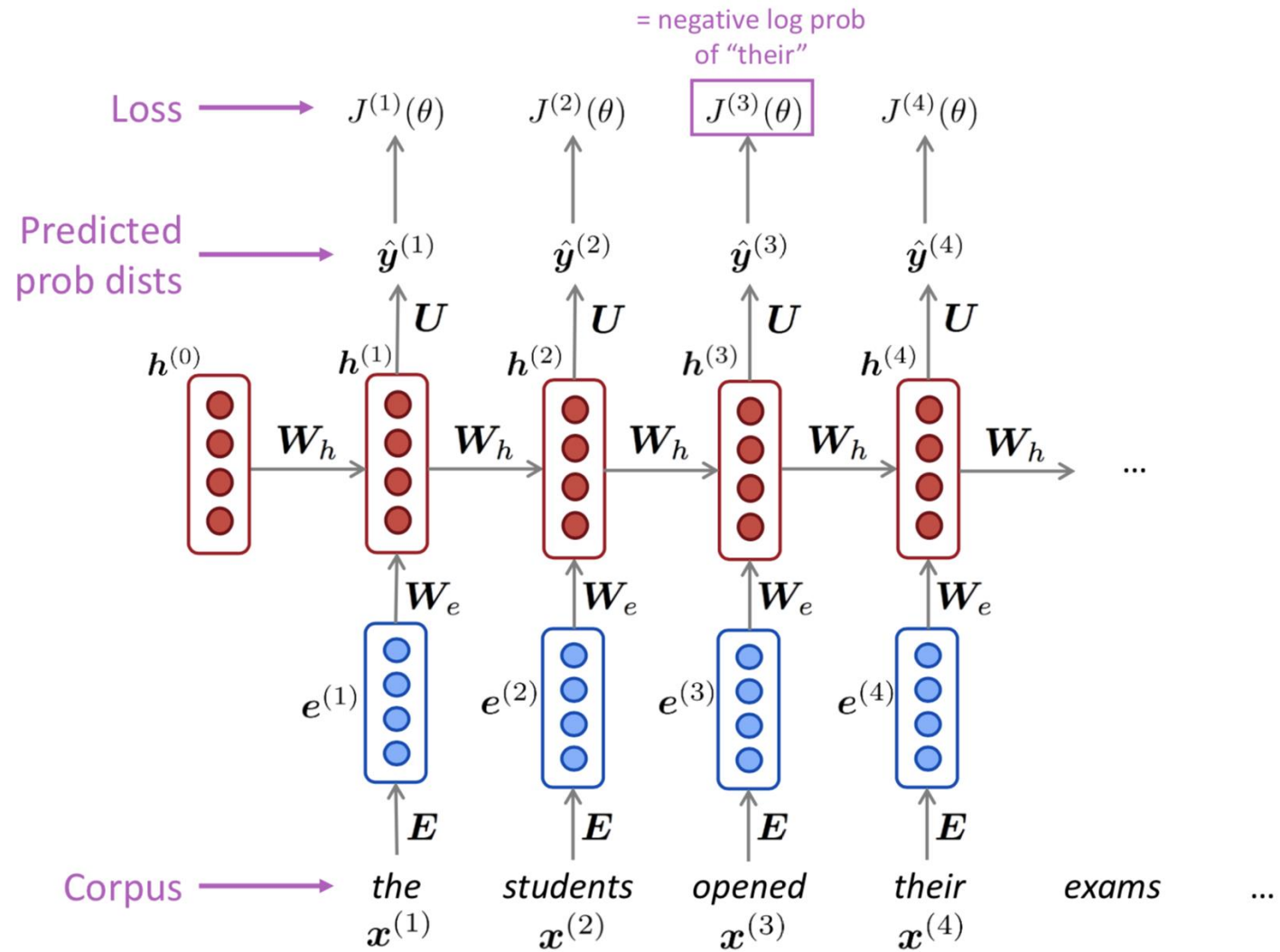
# Train a RNN Language Model



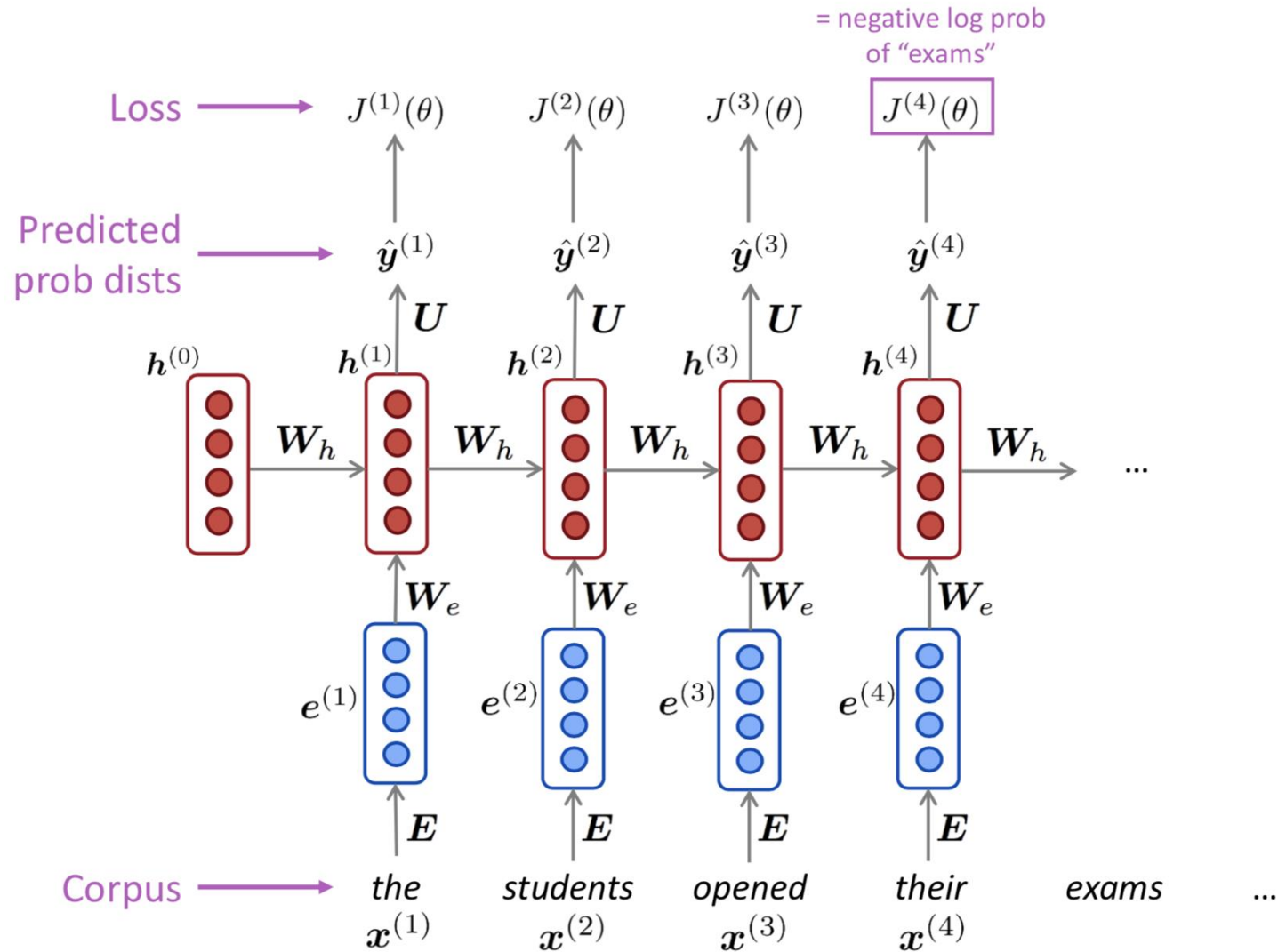
# Train a RNN Language Model



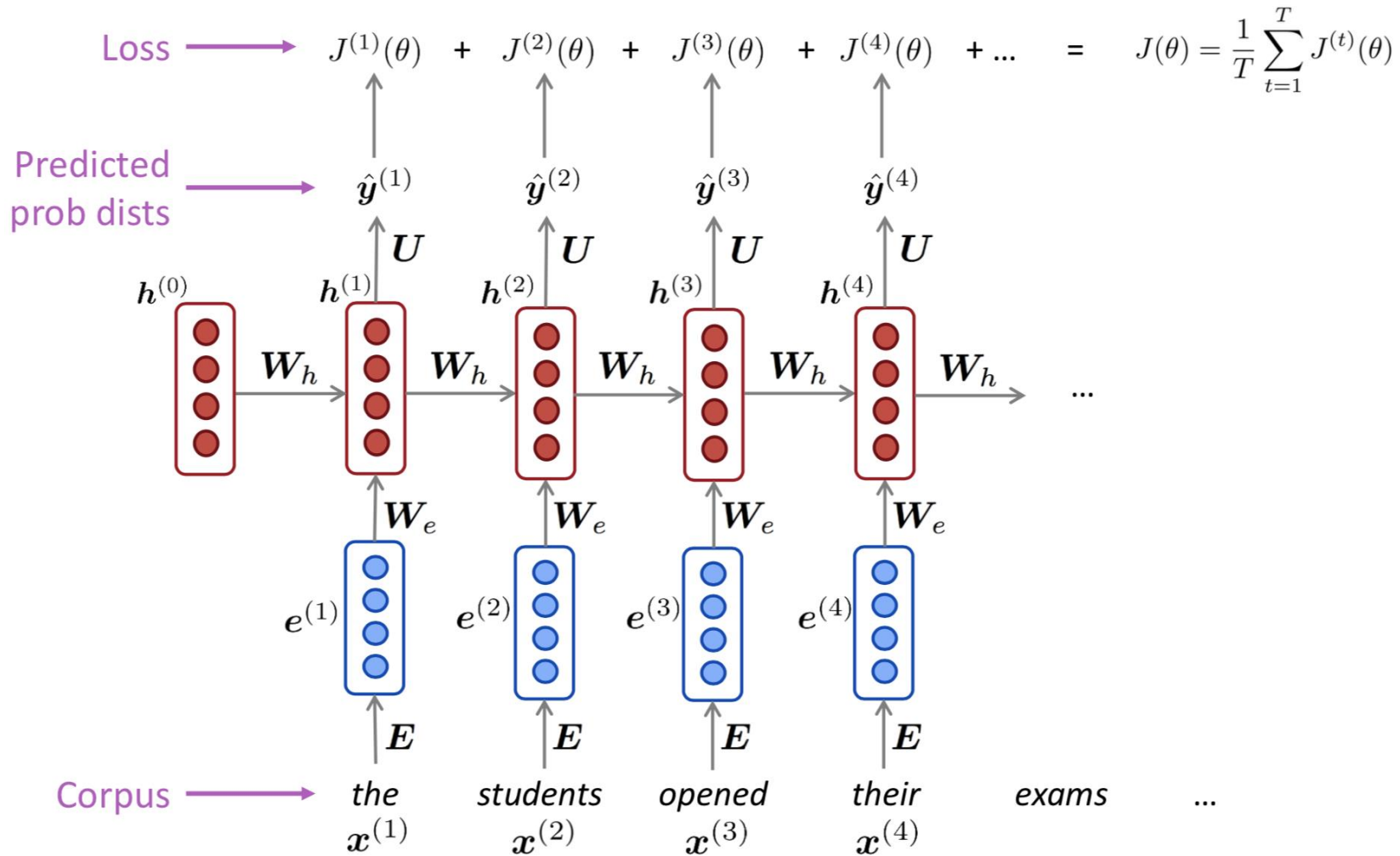
# Train a RNN Language Model



# Train a RNN Language Model



# Train a RNN Language Model



# Training a RNN Language Model

- However: Computing loss and gradients across entire corpus  $x^{(1)}, x^{(2)}, \dots, x^{(T)}$  is too expensive!

$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta)$$

- In practice, consider  $x^{(1)}, x^{(2)}, \dots, x^{(T)}$  as a sequence and compute recursively the gradient.
- Reminder: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data.



# Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:



“Sorry,” Harry shouted, panicking—“I’ll leave those brooms in London, are they?”

“No idea,” said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry’s shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn’t felt it seemed. He reached the teams too.

# A Problem for $N$ -Grams: Long Distance Dependencies

- Many times local context does not provide the most useful predictive clues, which instead are provided by ***long-distance dependencies***.
  - Syntactic dependencies
    - “The ***man*** next to the large oak tree near the grocery store on the corner **is** tall.”
    - “The ***men*** next to the large oak tree near the grocery store on the corner **are** tall.”
- More complex models of language are needed to handle such dependencies.

# How to fix long dependencies forgetting?

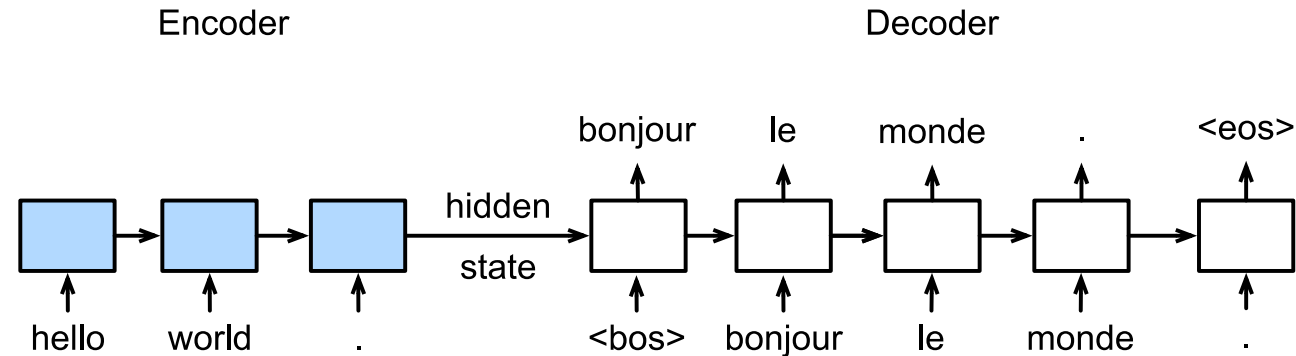
- The main problem is that it's too difficult for the RNN to learn to preserve information over many timesteps.
- In a vanilla RNN, the hidden state is constantly being rewritten

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$$

- First solutions:
  - LSTM (Long Short-Term Memory )
  - GRU (Gated Recurrent Unit)
- Current solutions:
  - Transformers

# Sequence to Sequence

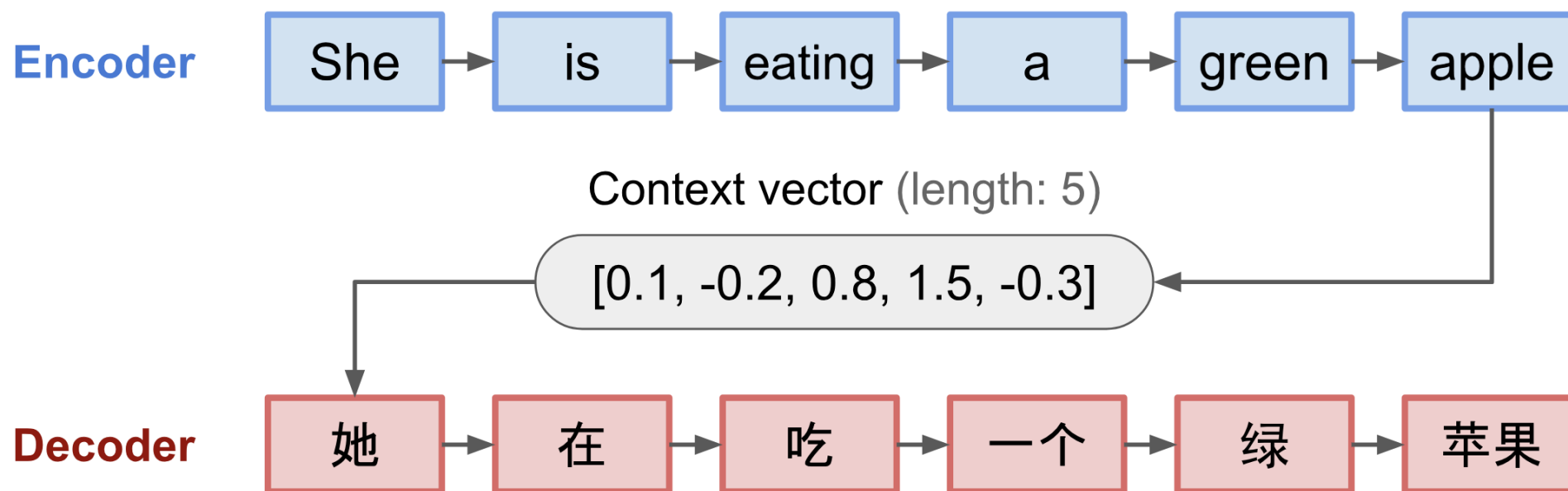
# Seq2Seq Models



- Note that we require that each sentence ends with a special end-of-sentence symbol **<EOS>**, which enables the model to define a distribution over sequences of all possible lengths.
- We also use a begin-of-sentence symbol **<BOS>** for the decoder.

# Encoder/Decoder

- The encoder-decoder model, translating the sentence “she is eating a green apple” to Chinese.
- The visualization of both encoder and decoder is unrolled in time.



# Encoder

- From a probabilistic perspective, the encoder/decoder model is a general method to learn the conditional distribution  $p(y_1, \dots, y_{T'} | x_1, x_2, \dots, x_T)$
- An encoder reads the input sentence, a sequence of vectors  $x = (x_1, x_2, \dots, x_T)$ , into a context vector  $c$
- The most common approach is to use an RNN
  - RNN hidden state:  $h_t = f(x_t, h_{t-1})$
  - Context vector:  $c = q(\{h_1, \dots, h_T\})$

where  $h_t \in \mathbb{R}^n$  is a hidden state at time  $t$ , and  $c$  is a vector generated from the sequence of the hidden states.

- $f$  and  $q$  are some nonlinear functions.

Ilya Sutskever, Oriol Vinyals, Quoc V. Le, 2014, "Sequence to Sequence Learning with Neural Networks, » pp. 3104–311 in NIPS 2014

Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio:  
Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. EMNLP 2014: 1724-1734

# Decoder

- The decoder is often trained to predict the next word  $y_{t'}$  given the context vector  $c$  and all the previously predicted words  $\{y_1, y_2, \dots, y_{t'-1}\}$ .
- The decoder defines a probability over the translation  $y = (y_1, \dots, y_{T'})$  by decomposing the joint probability into the ordered conditionals:

$$p(y|x) = \prod_{t'=1}^{T'} p(y_{t'}|y_1, \dots, y_{t'-1}, x) \approx \prod_{t'=1}^{T'} p(y_{t'}|y_1, \dots, y_{t'-1}, \textcolor{red}{c})$$

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.

K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. EMNLP, page 1724-1734. ACL (2014)



# Decoder with context vector

- The decoder is also an RNN
  - RNN decoder hidden state:  $s_{t'} = f(y_{t'-1}, s_{t'-1}, c)$
  - The context vector  $c$  can be reused for any hidden state in the decoder

- With an RNN, each conditional probability is modeled as

$$p(y_{t'} | y_1, \dots, y_{t'-1}, x) = g(y_{t'-1}, s_{t'}, c)$$

where  $g$  is a nonlinear, potentially multi-layered, function that outputs the probability of  $y_{t'}$ .

- **Drawback of the single context vector:**
  - A critical and apparent disadvantage of this fixed-length context vector design is incapability of remembering long sentences.
  - Often it has forgotten the first part once it completes processing the whole input.
  - The attention mechanism was born to resolve this problem.

# Conclusion

# Conclusion

- Word embedding is a powerful preprocessing
- Used also in more advanced processing (graph processing for example)
- Embedding is not really deep learning but...
- Embedding is a very important step for deep neural network architecture dedicated to sequence analysis
- RNNs are able to manage an internal state to learn the dependencies between the inputs and to process variable length sequences of inputs.
- Long dependencies forgetting is an important issue!
- Encoder/decoder architecture become the current solution to address this difficulty.