Deep Learning

Recurrent Neural Network

Lionel Fillatre

2024-2025

Outline

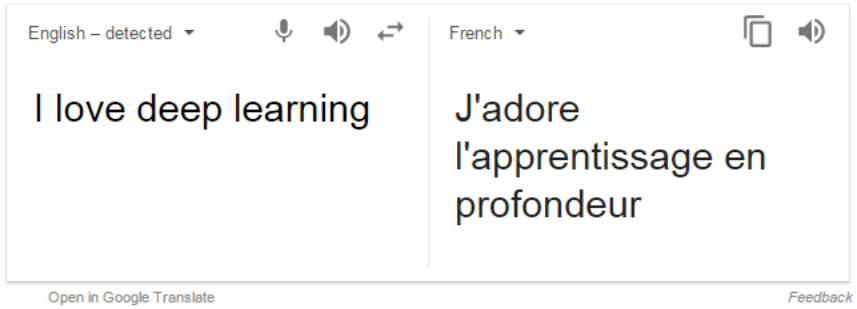
- Introduction
- Language Model
- N-gram
- Word Embedding
- Word2Vec
- Neural Language Model
- Recurrent Neural Networks (RNN)
- Long Short-Term Memory (LSTM)
- Conclusion

Introduction

Natural Language Processing (NLP) Applications

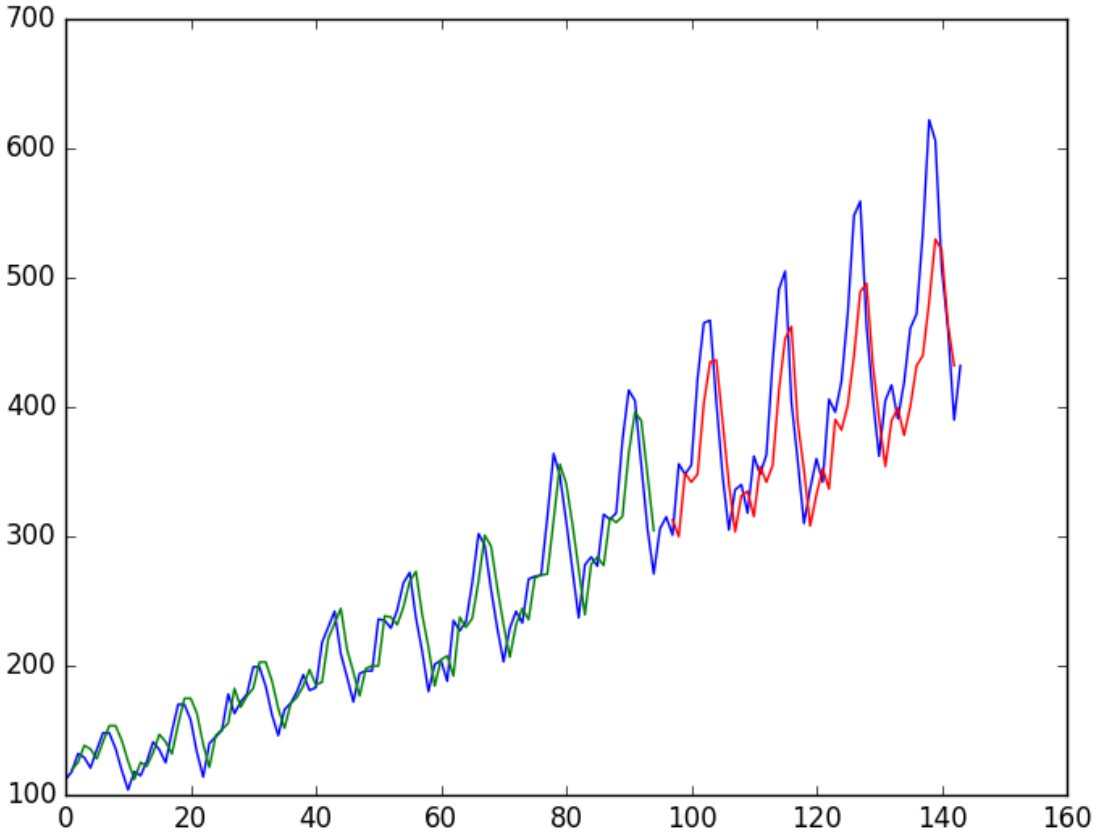
- Sentiment Analysis
- Email Filters
- Voice Recognition
- Information Extraction
- Translation
- Large language model-based chatbot
- …

	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



Time Series

Blue: groundtruth
Green: train prediction
Red: test prediction



LSTM Trained on Regression Formulation
of Passenger Prediction Problem

https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Language Model

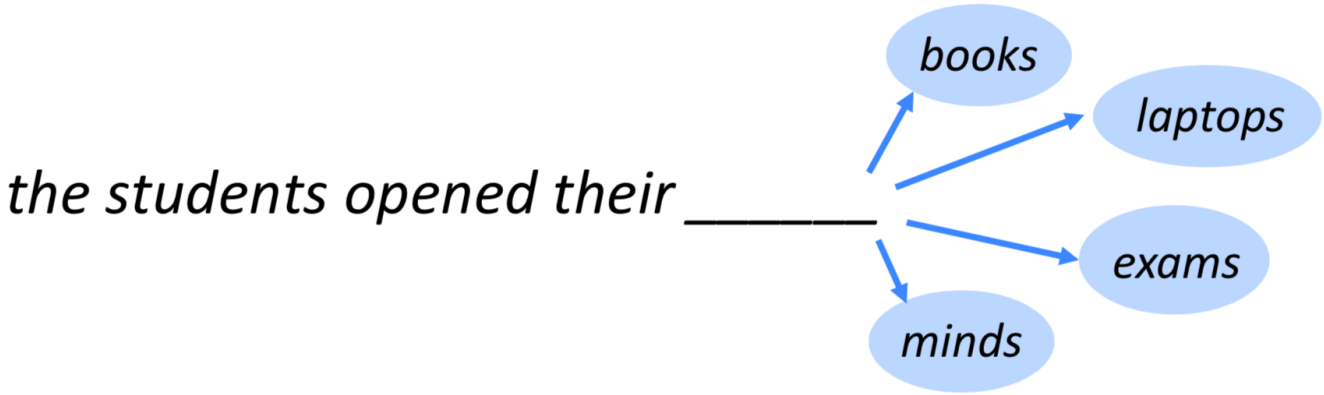
Uses of Language Models

- 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!

Language Modeling

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



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

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

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

- 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 $w_1,w_2,\ldots,w_T$, then the probability of this text (according to the Language Model) is:

$$\Pr(w_1,w_2,\cdots,w_T) = \prod_{t=1}^{T} \Pr\left(w_t \middle| w_{t-1},\ldots,w_1\right)$$

- Example: Pr(its water was so transparent) = Pr(its)* Pr(water|its)* Pr(was|its water)* Pr(so|its water was)* Pr(transparent|its water was so)

- Language Modeling provides $\Pr\left(w_t \middle| w_{t-1},\ldots,w_1\right)$

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(phone | 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(phone)
  - Bigram: P(phone | cell)
  - Trigram: P(phone | 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 kth-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

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

- Conditional probabilities:

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

- **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:

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

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]

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

- 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.

Problem with words as discrete symbols

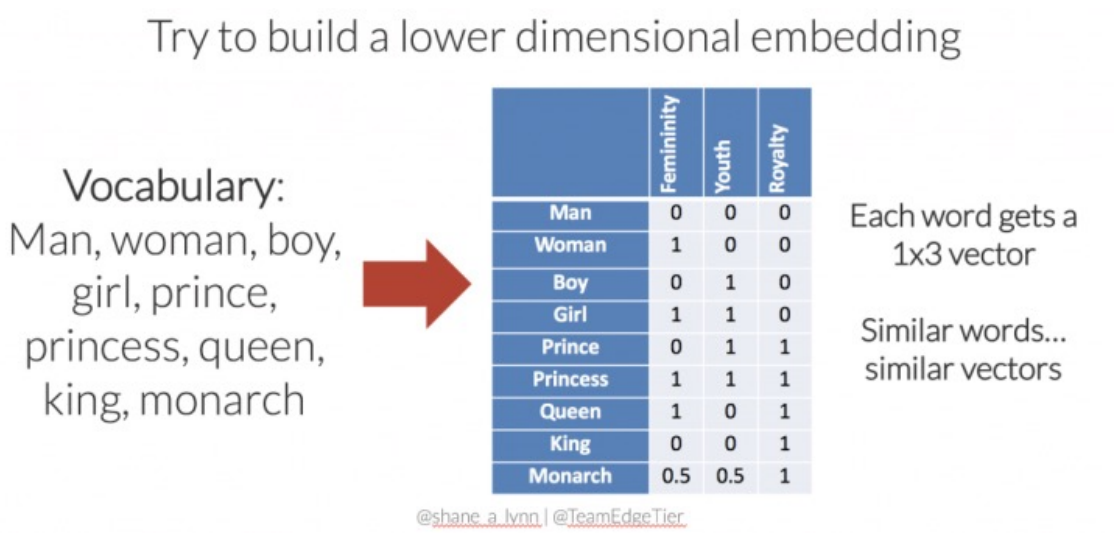
- **Example:** in web search, if user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”.
- But:
  - motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
  - hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
  - These two vectors are orthogonal.

- 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

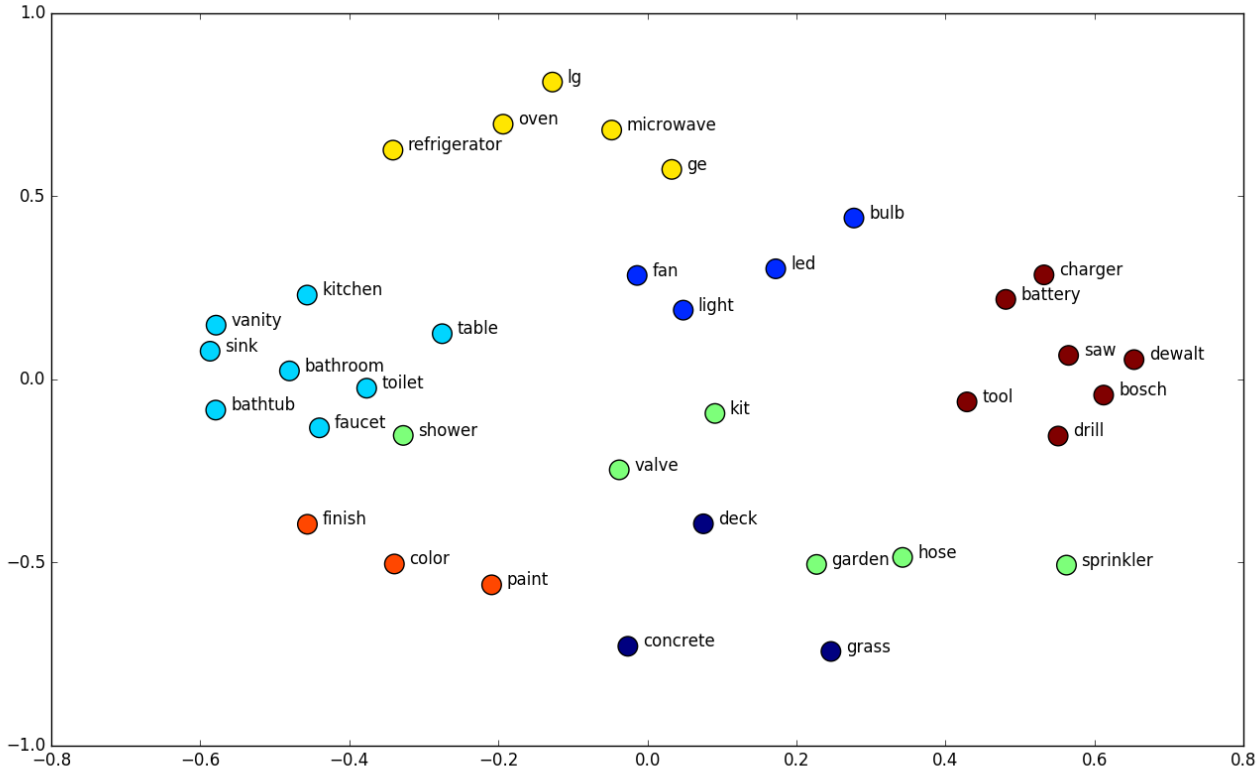


banking = $\begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$

- 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



https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/

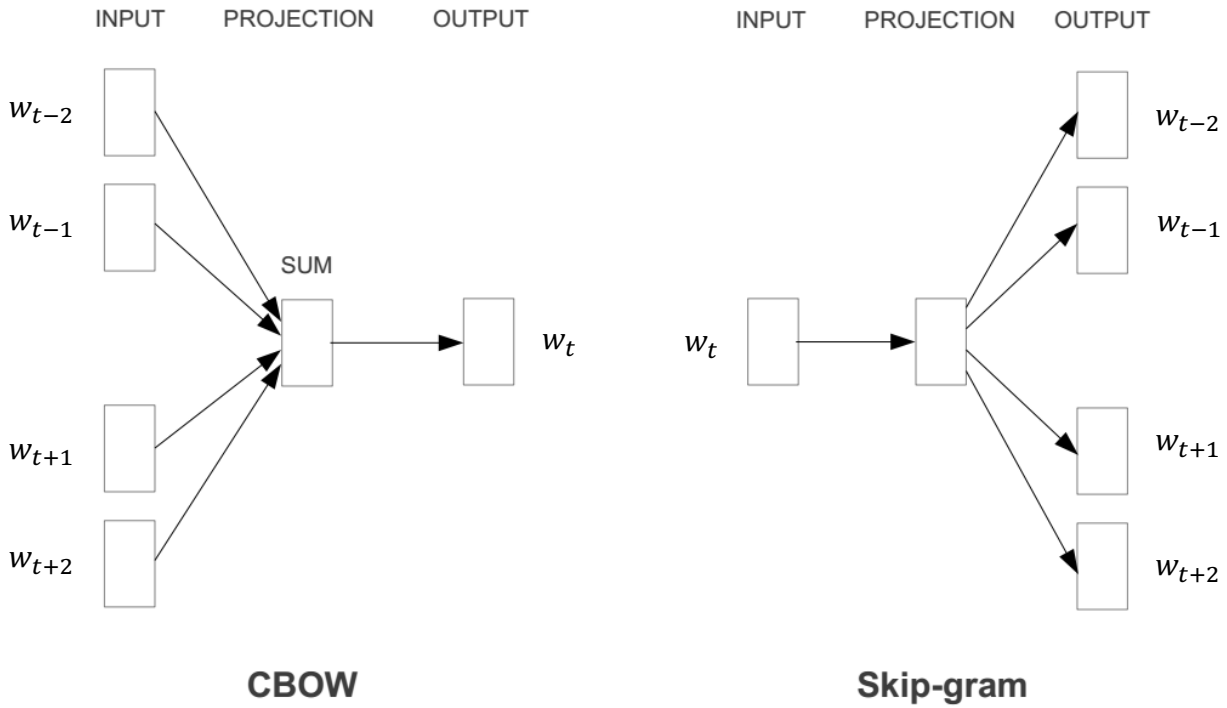
Word2Vec

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

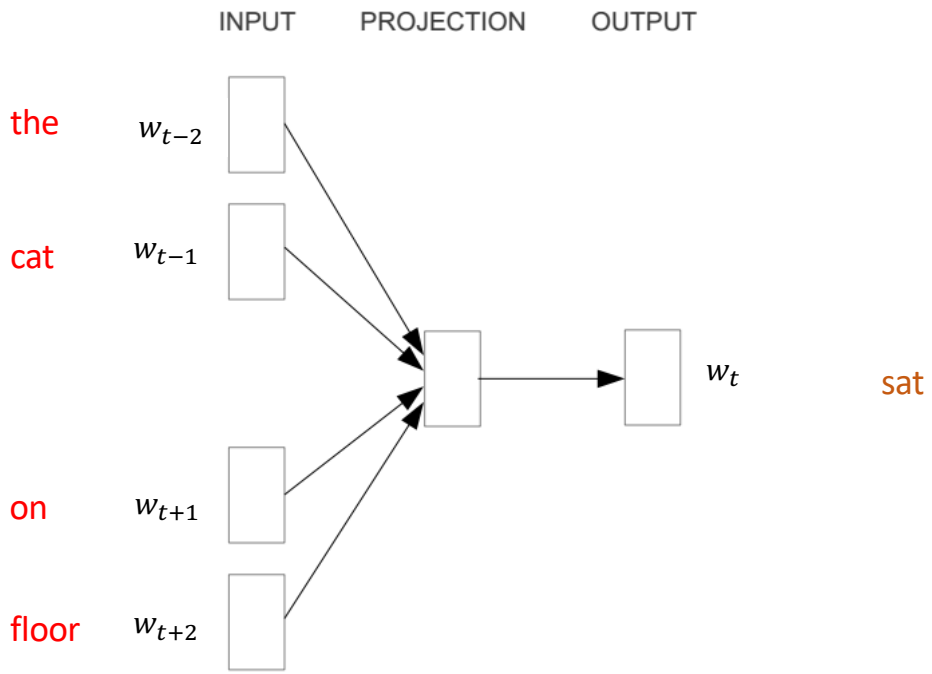
Represent the meaning of word – word2vec

- 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.

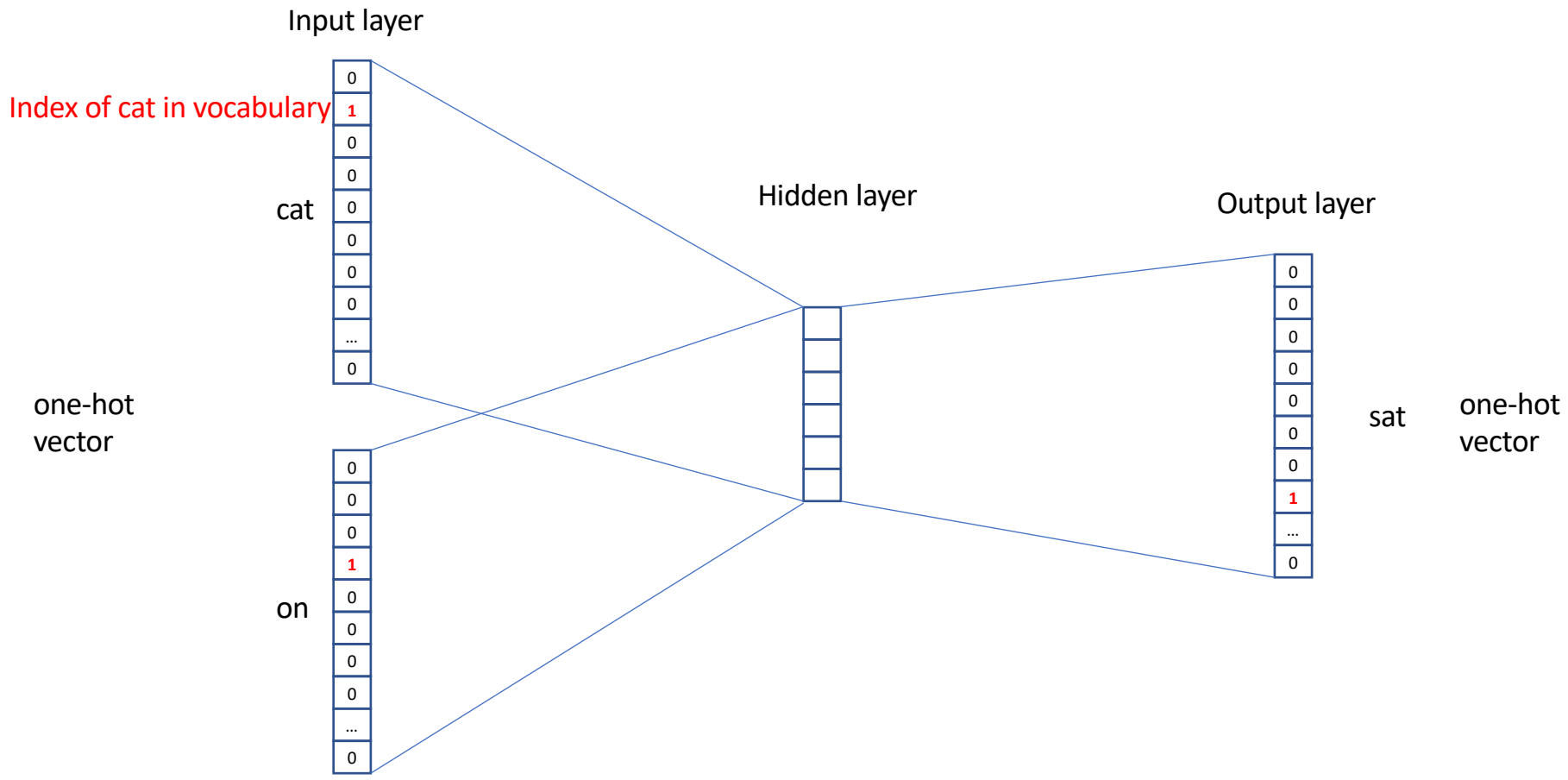


Word2vec – Continuous Bag of Word

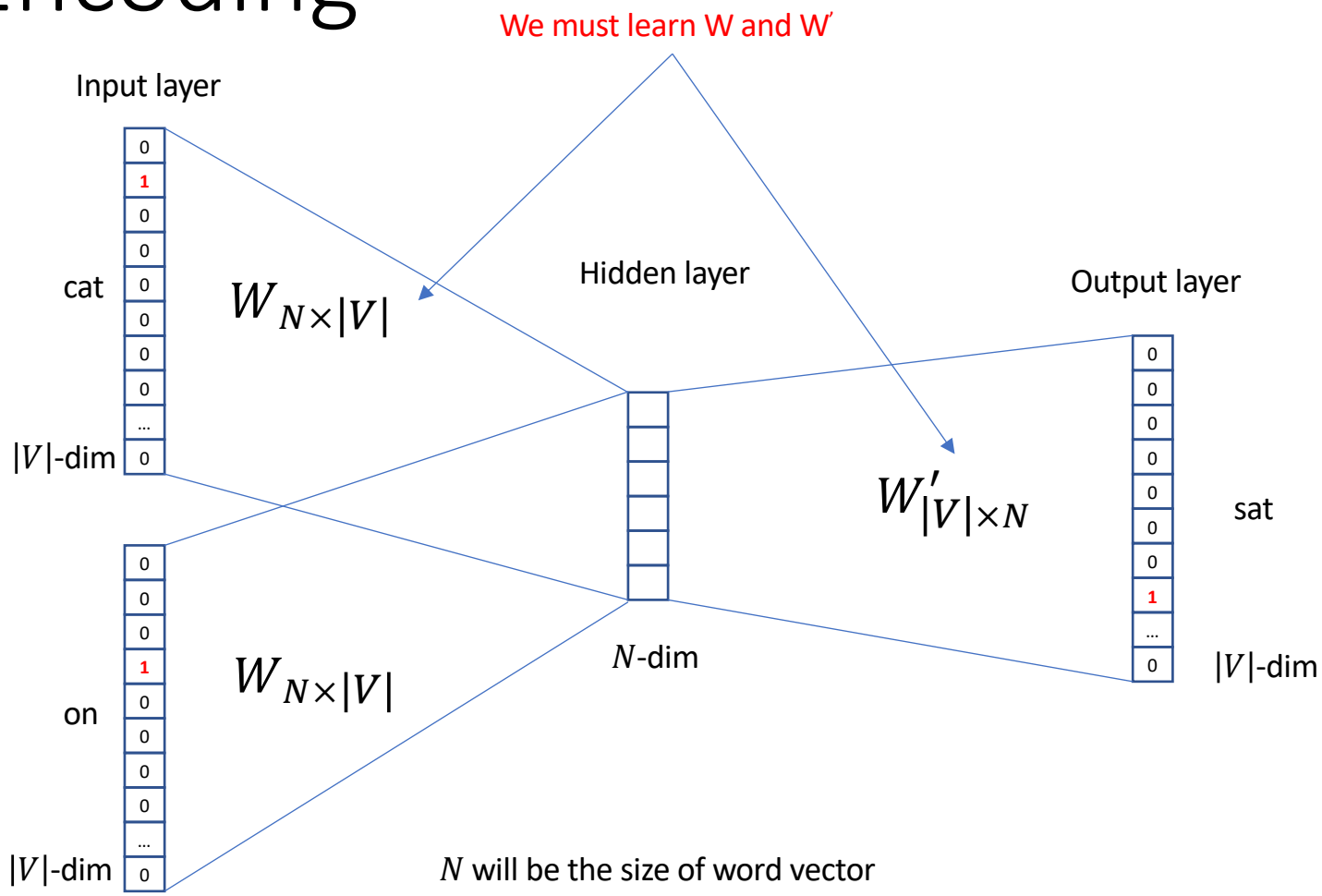
- E.g. “The cat sat on floor”
  - Window size = 2



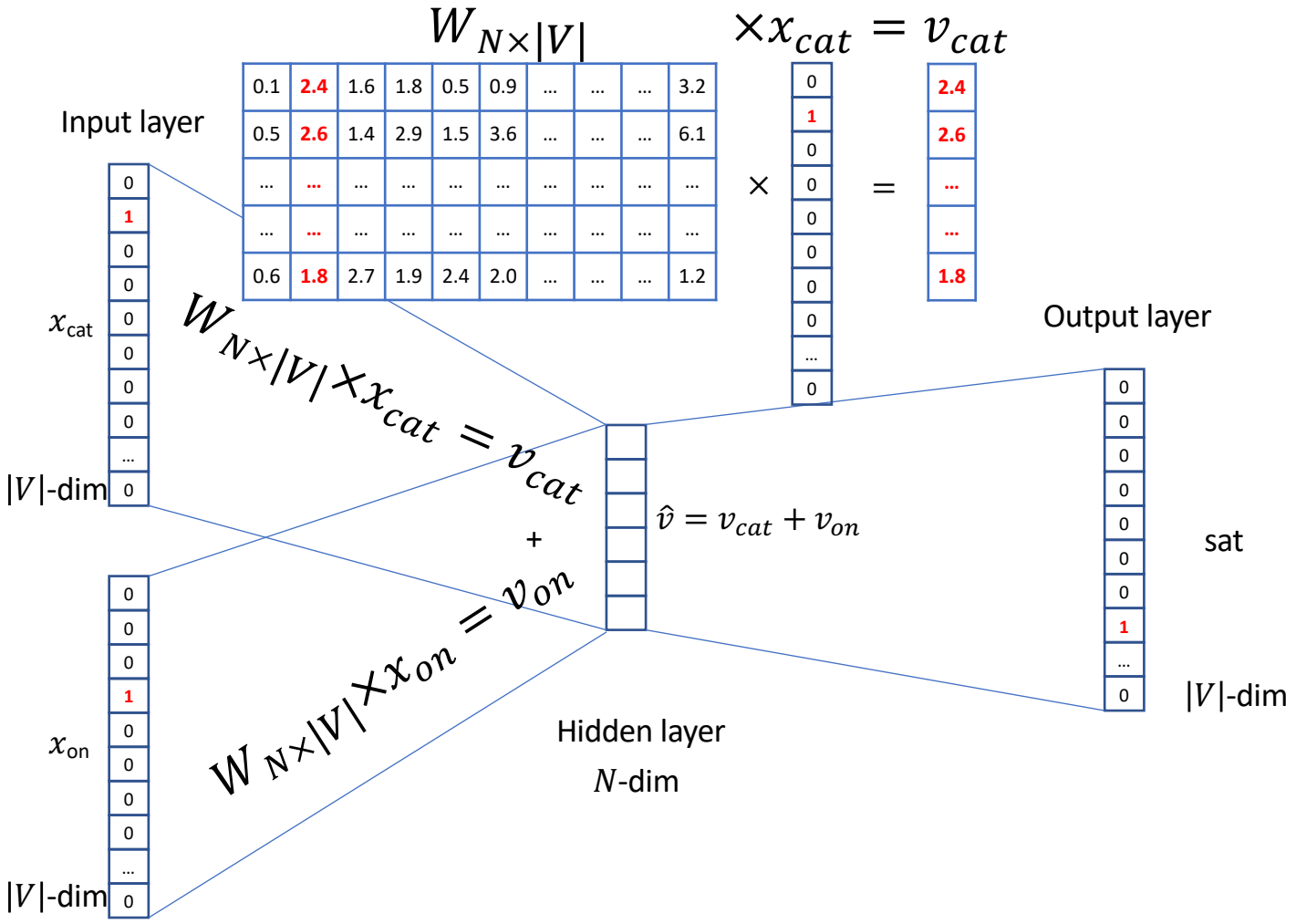
Input Encoding (window size = 1)



Matrix Encoding

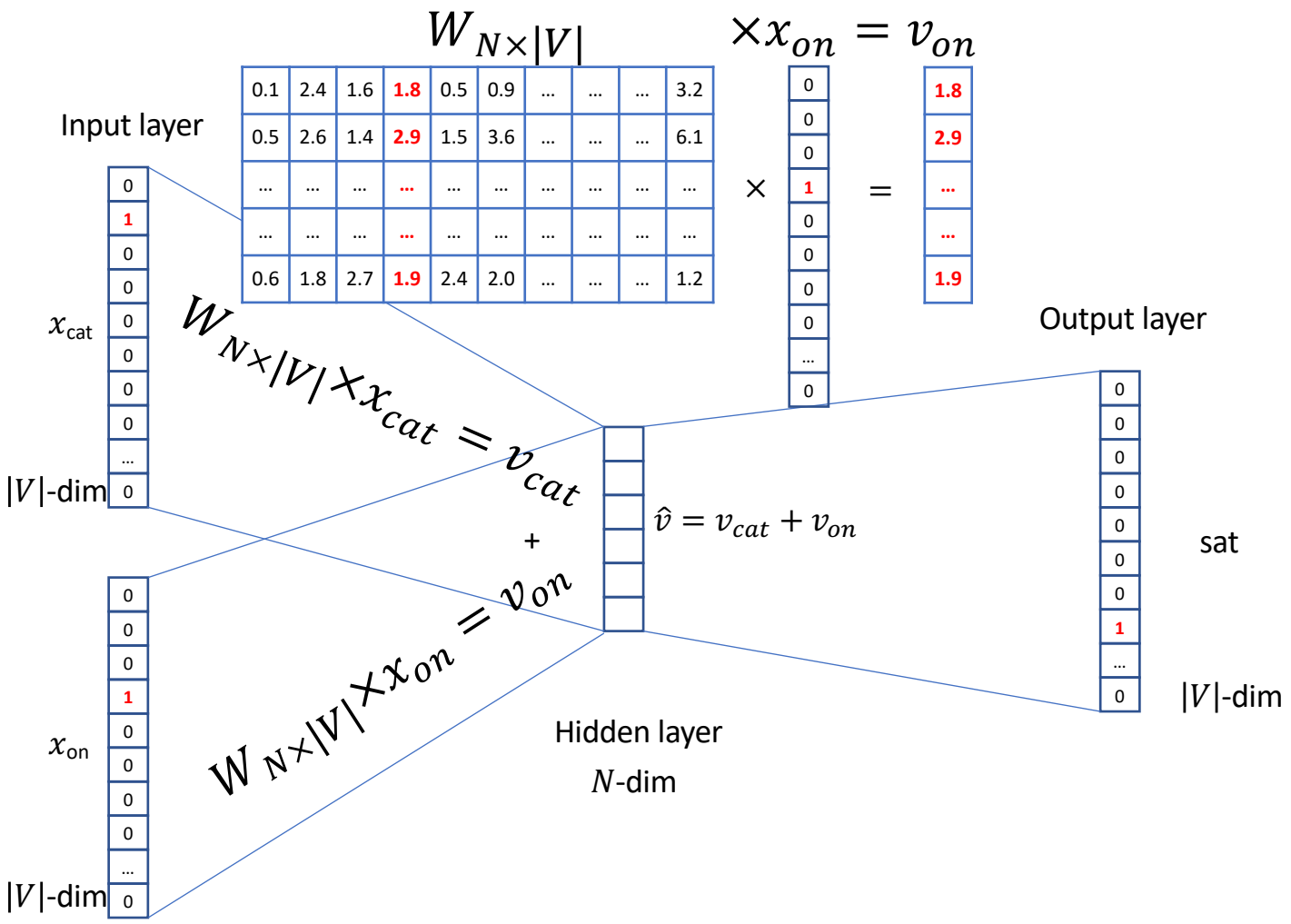


Example

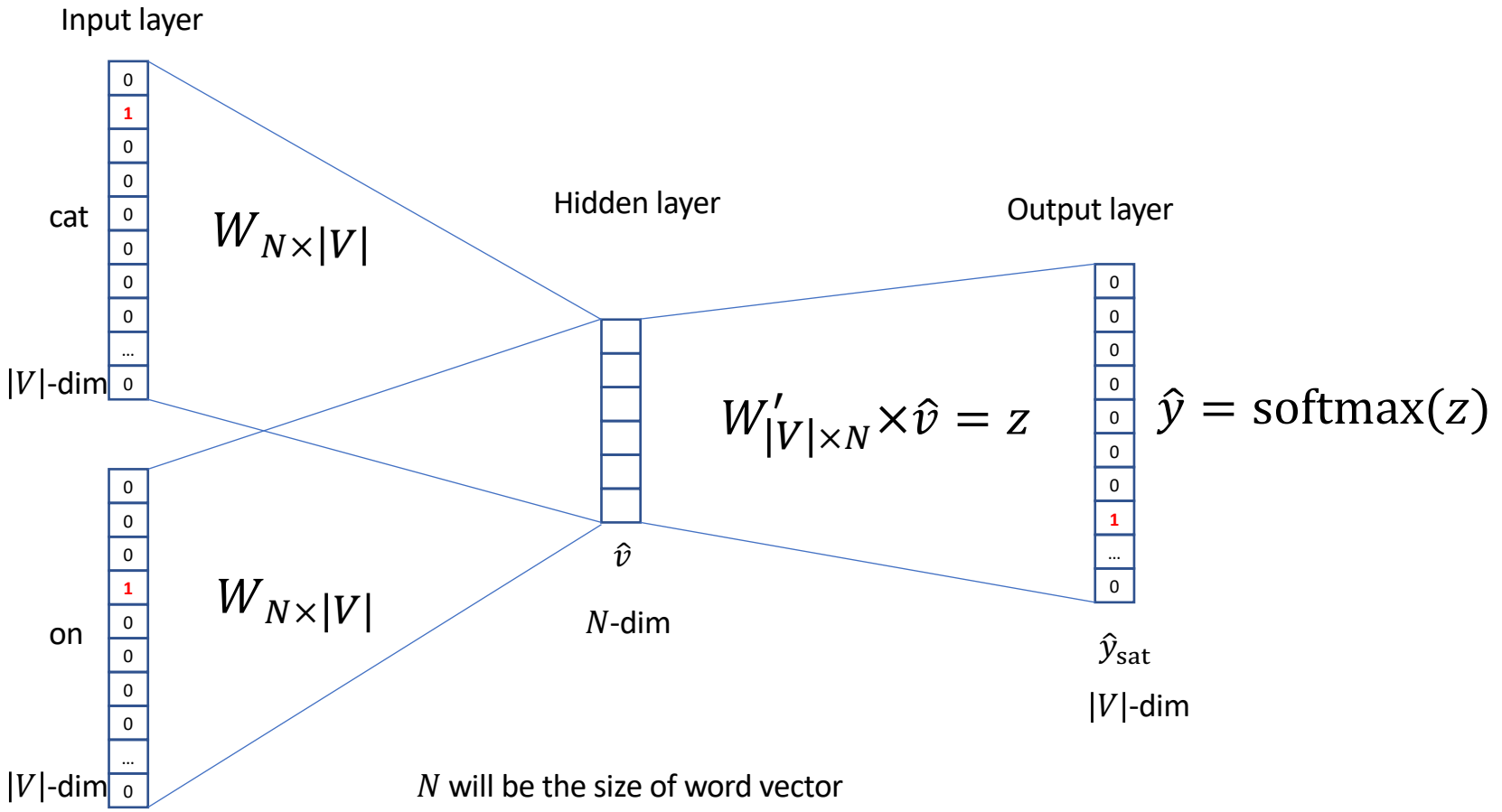


Note that we can take $\hat{v} = \frac{v_{cat}+v_{on}}{2}$ instead of $\hat{v} = v_{cat} + v_{on}$

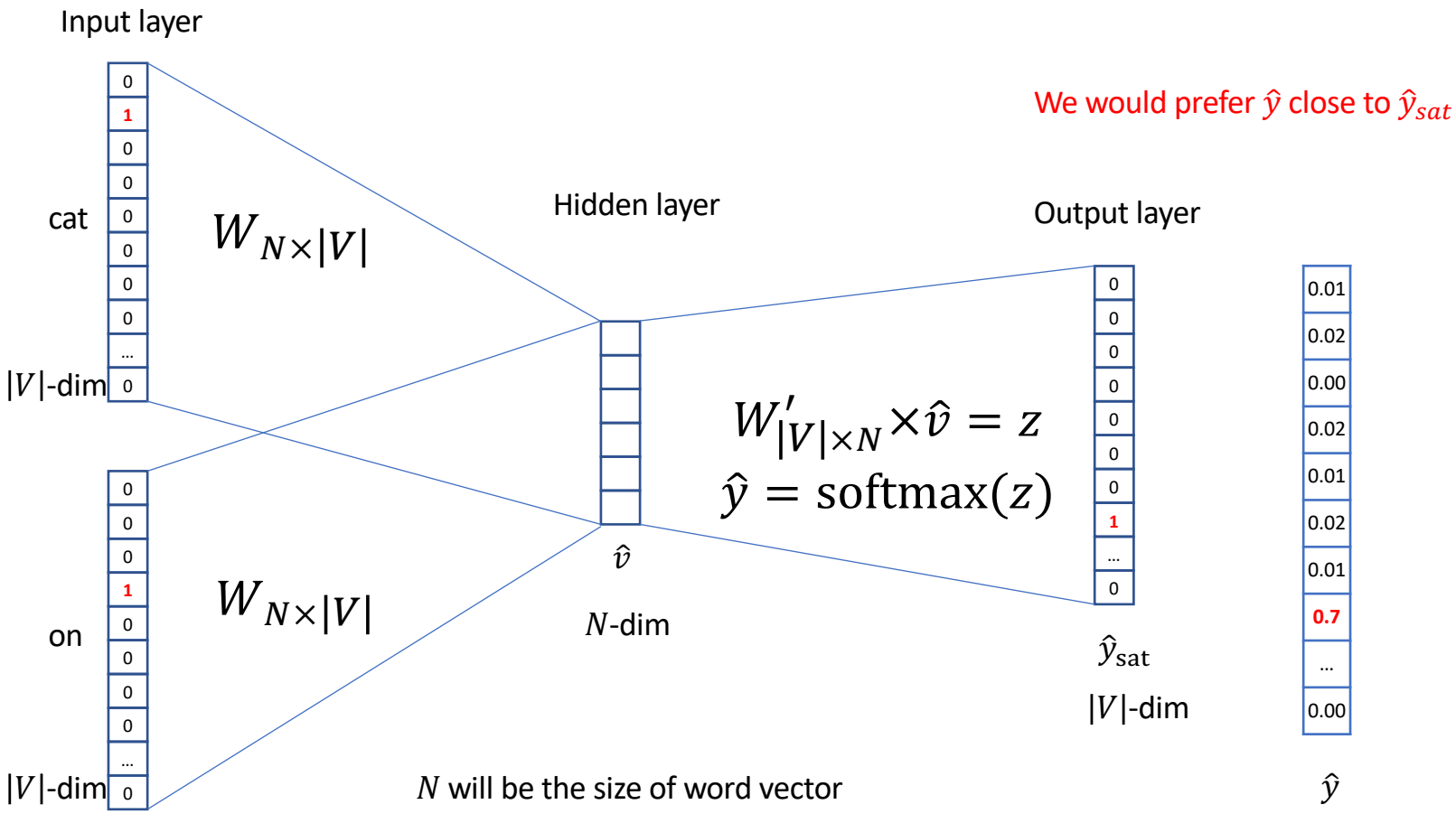
Example


$W_{N\times|V|}$
$\times x_{on} = v_{on}$
Input layer
x_{cat}
$|V|$-dim
$W_{N\times|V|}x_{cat} = v_{cat}$
+
$W_{N\times|V|}x_{on} = v_{on}$
x_{on}
$|V|$-dim
$\hat{v} = v_{cat} + v_{on}$
Hidden layer
N-dim
Output layer
sat
$|V|$-dim

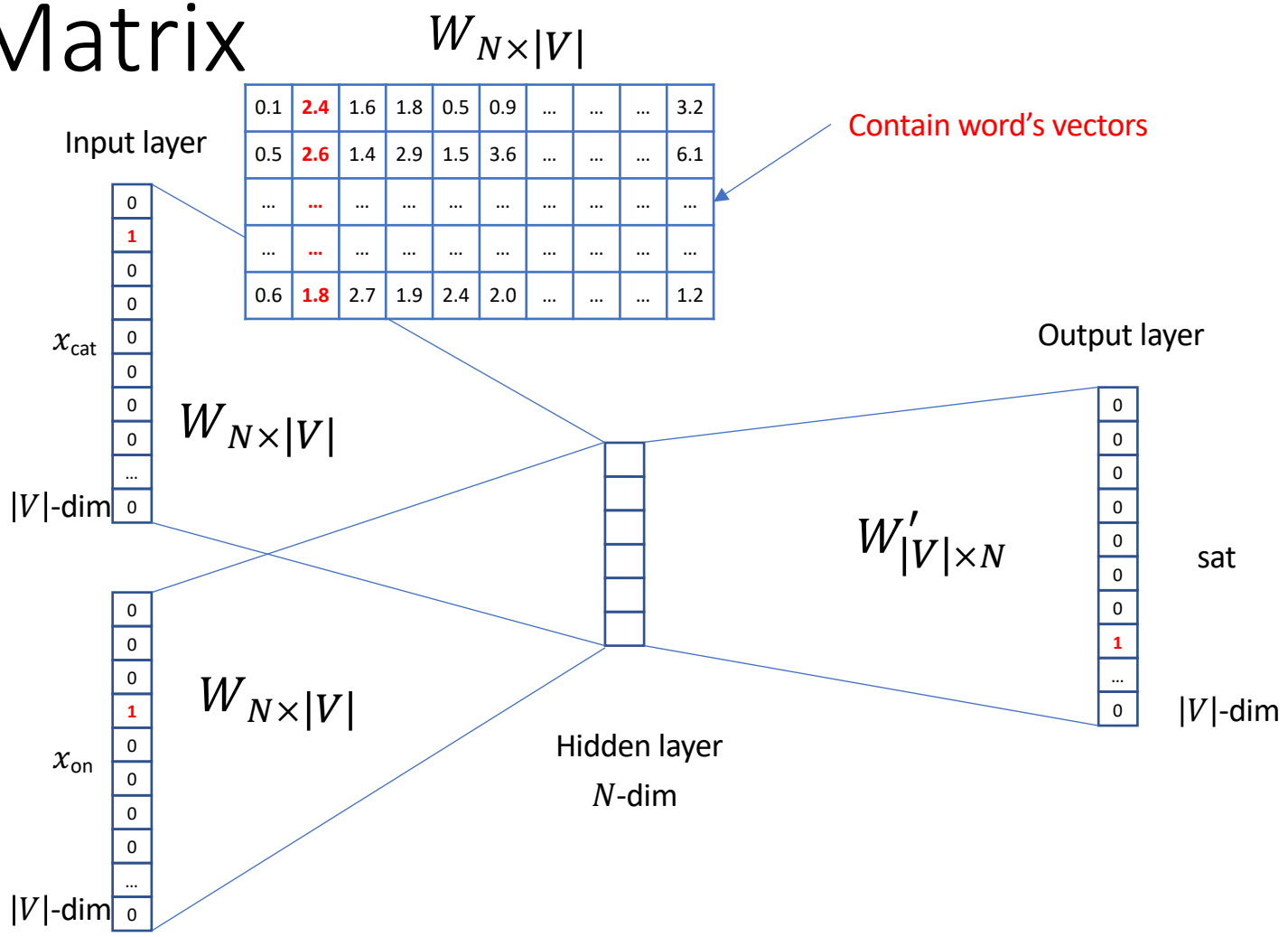
Output Layer



Output


Input layer
cat
|V|-dim
on
|V|-dim
$W_{N\times|V|}$
$W_{N\times|V|}$
Hidden layer
$\hat{v}$
N-dim
$W'_{|V|\times N}\times\hat{v}=z$
$\hat{y}=\text{softmax}(z)$
Output layer
$\hat{y}_{\text{sat}}$
|V|-dim
We would prefer $\hat{y}$ close to $\hat{y}_{sat}$
0.01
0.02
0.00
0.02
0.01
0.02
0.01
0.7
0.00
$\hat{y}$
N will be the size of word vector

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

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

  - Reminder: $\hat{v}_k$ is the vector representation (hidden vector) of the context $\{w_{k+j}, -m \leq j \leq m, j \neq 0\}$ 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 θ 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:

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

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

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

Negative sampling

- Let c_t be the context $c_t = \{w_{k+j}, -m \leq j \leq m, j \neq 0\}$
- We need to maximize

$$\underset{\theta}{\operatorname{argmax}} \sum_{t=1}^{T} \log \Pr(w_t | c_t; \theta) = \sum_{t=1}^{T} \left(\log \exp(\hat{v}_k^T u_k) - \log \sum_{j=1}^{|V|} \exp(\hat{v}_k^T u_j) \right)$$

- It is computationally expensive because the sum $\sum_{j=1}^{|V|} \exp(\hat{v}_k^T u_j)$ is very expensive to compute due to the summation over the entire vocabulary
- Instead of looping over the entire vocabulary, we can just randomly select a handful of words!
- It is a bit tricky because we must modify
  - The objective function
  - The sampling

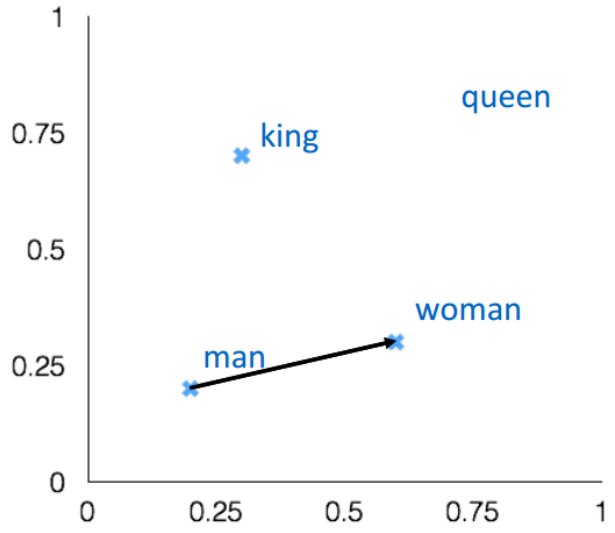
Some interesting results

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

$$\Pr\left(x^{(t+1)} \middle| x^{(t)}, x^{(t-1)}, \ldots, x^{(1)}\right)$$

- 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

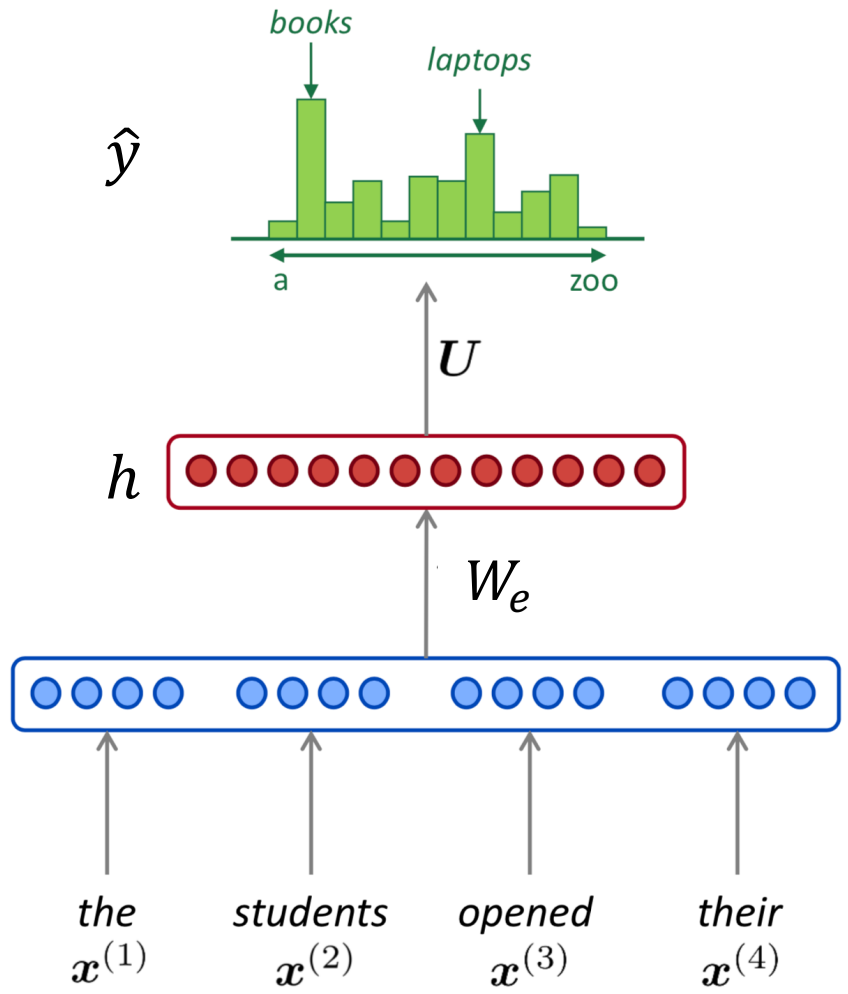
$h = f(W_e e + b_1)$

- Concatenated word embeddings

$e = \left[e^{(1)}, e^{(2)}, e^{(3)}, e^{(4)}\right]$

- Words / one-hot vectors

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

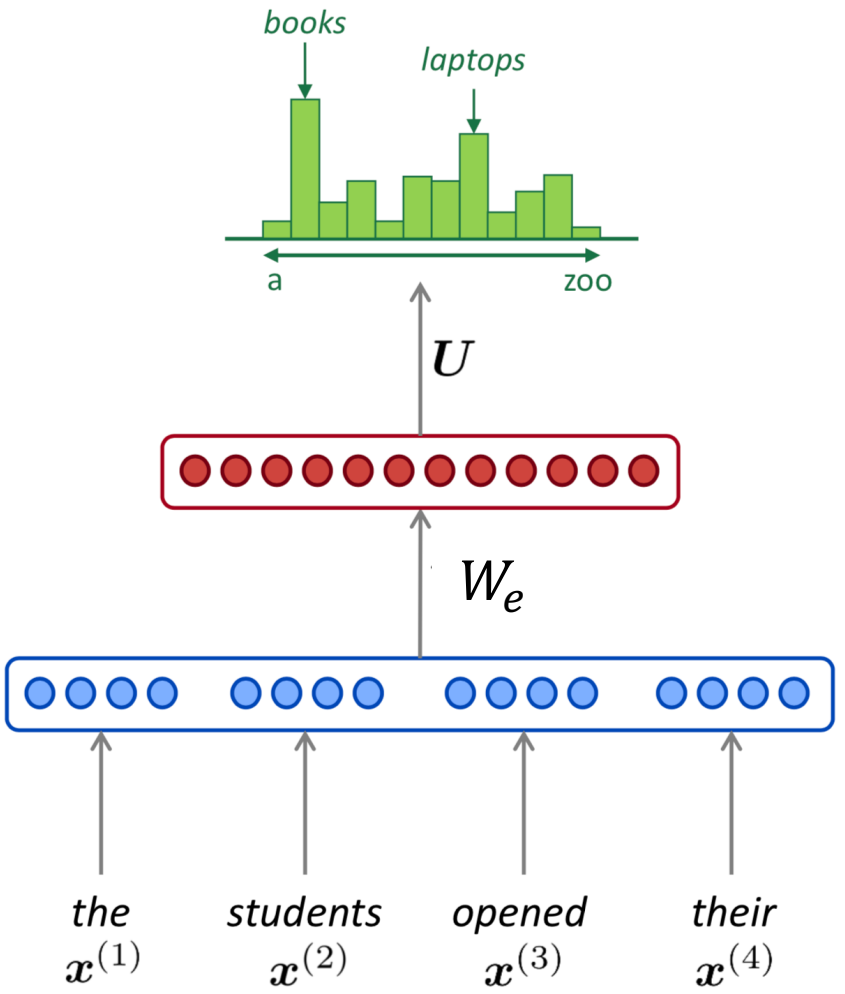


A fixed-window neural Language Model

- 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

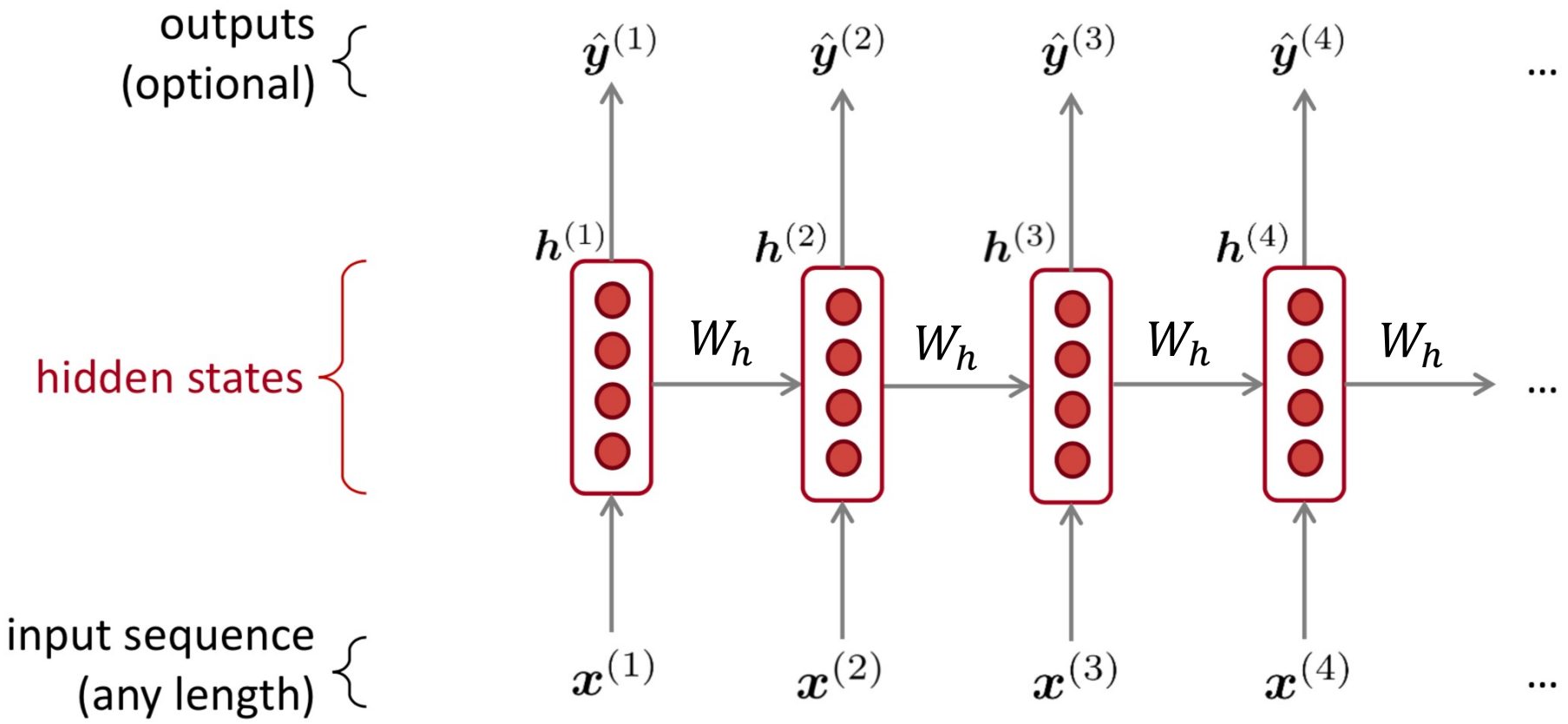
    ⇒ No symmetry in how the inputs are processed.
  - We need a neural architecture that can process any length input



Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN)

- Core idea: Apply the same weights repeatedly



A RNN Language Model

- Output distribution

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

- Hidden states

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

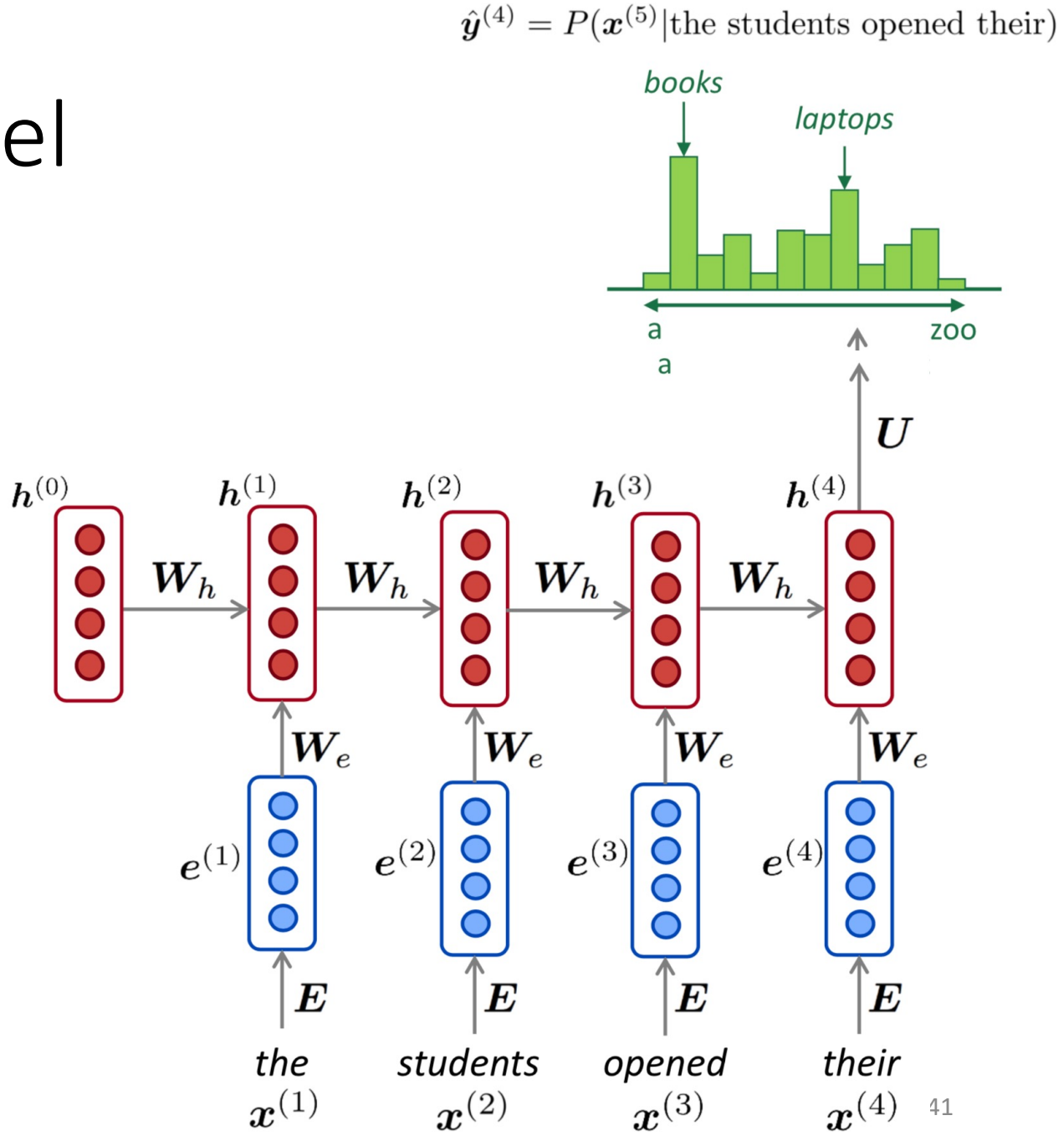
$h^{(0)}$ is the initial hidden state

- 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 $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$

- 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

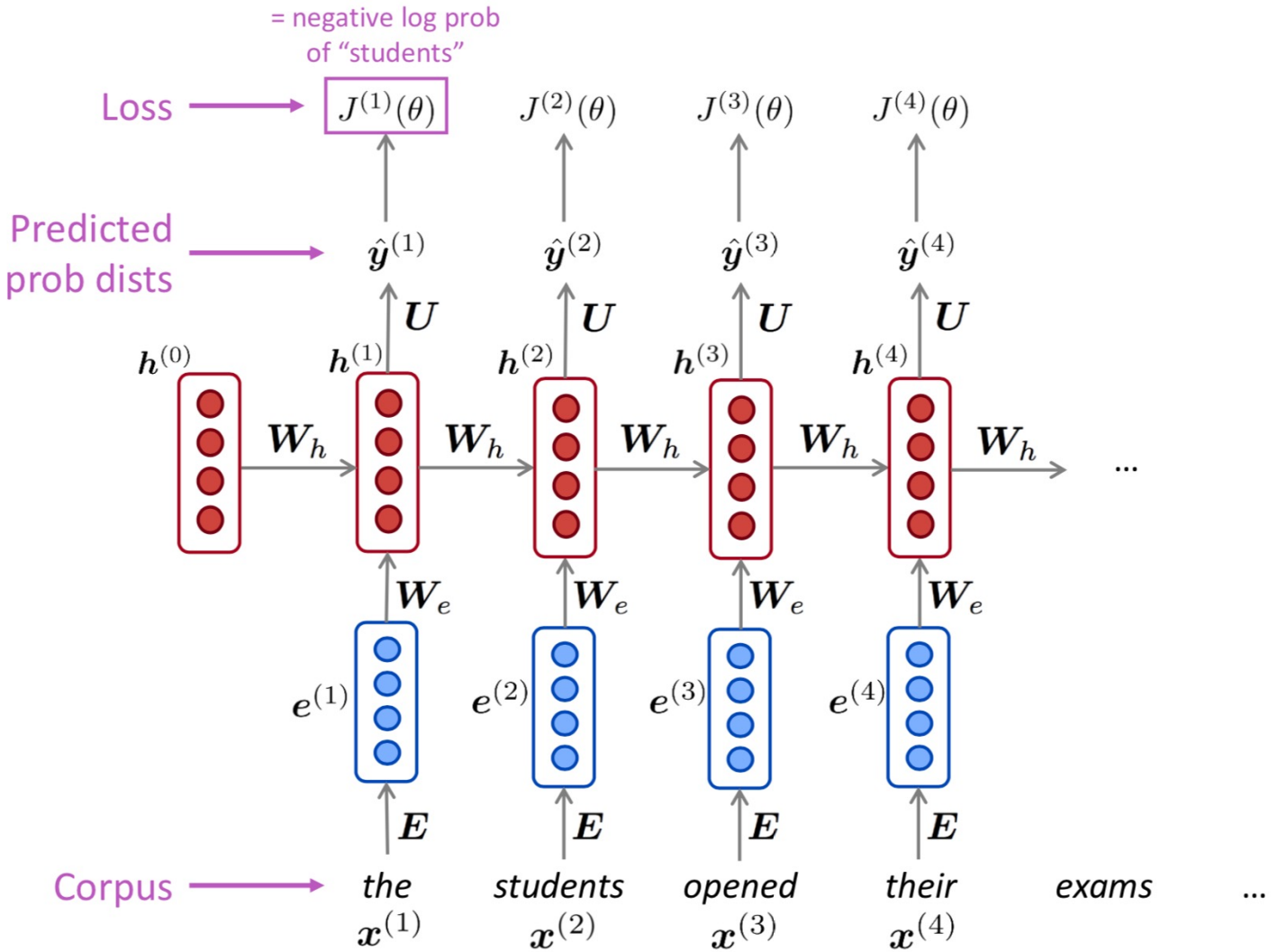
- Loss function on step t is cross-entropy between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot encoding for $x^{(t+1)}$):

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

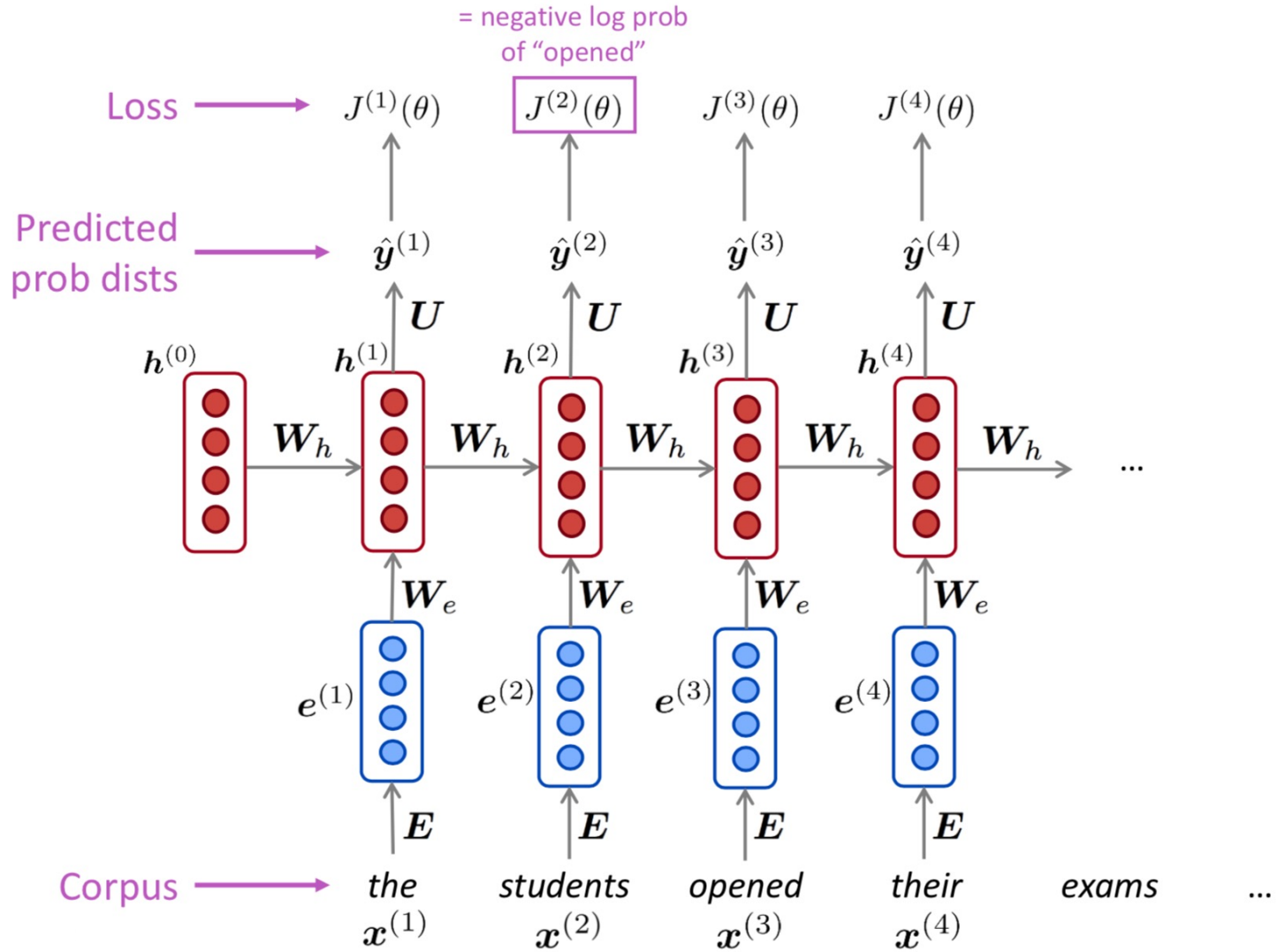
- Average this to get overall loss for entire training set:

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

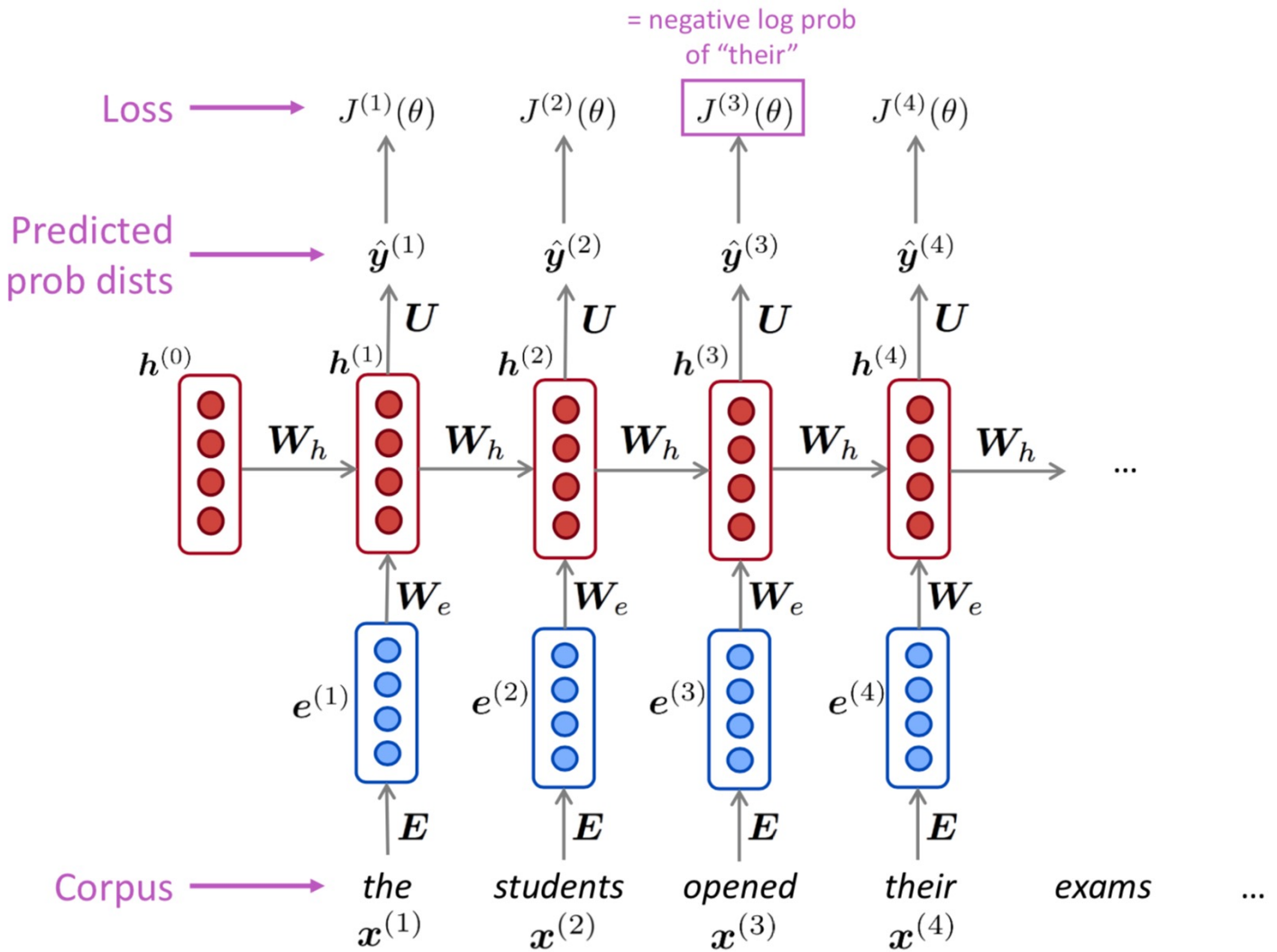
Train a RNN Language Model



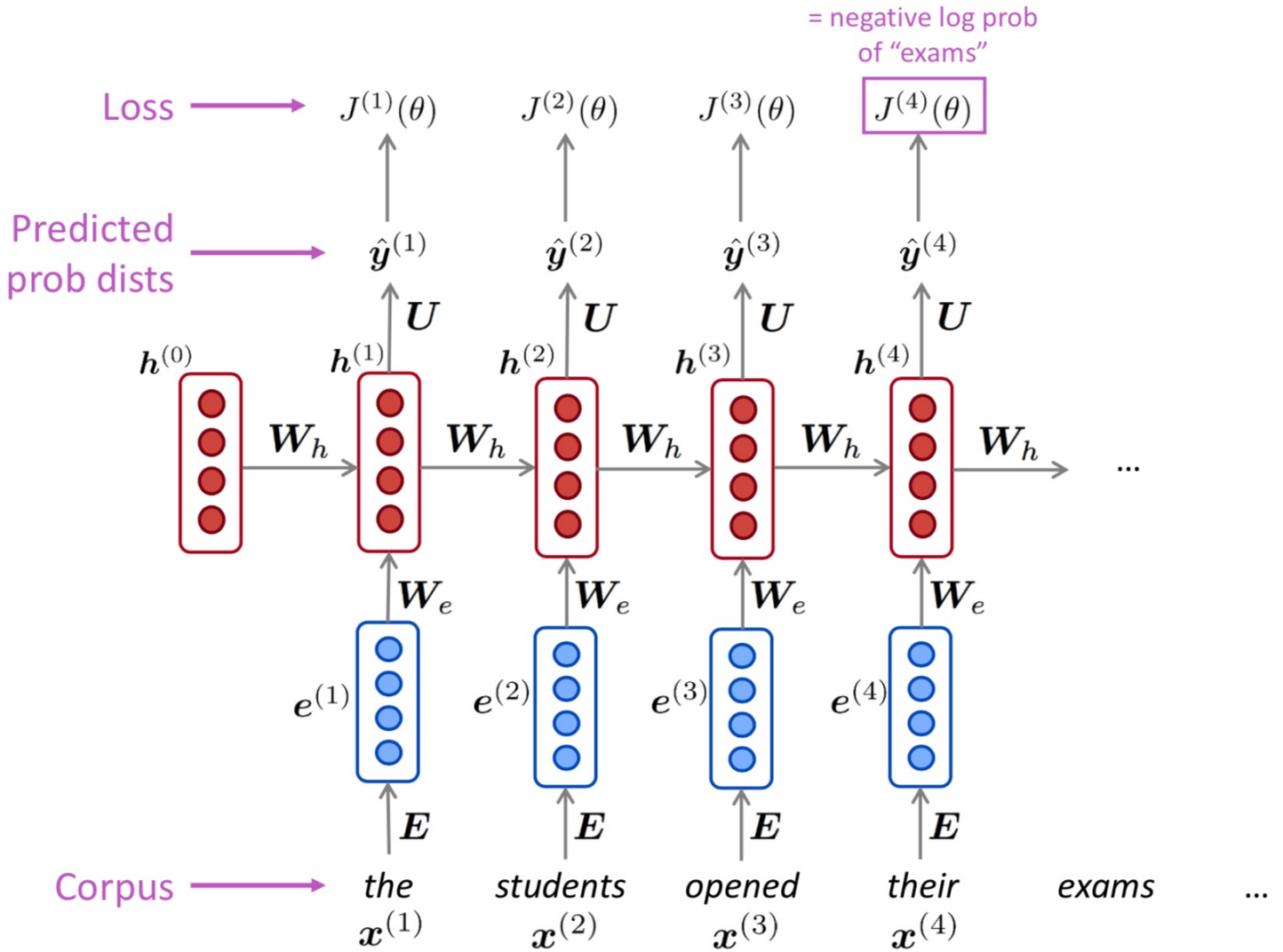
Train a RNN Language Model



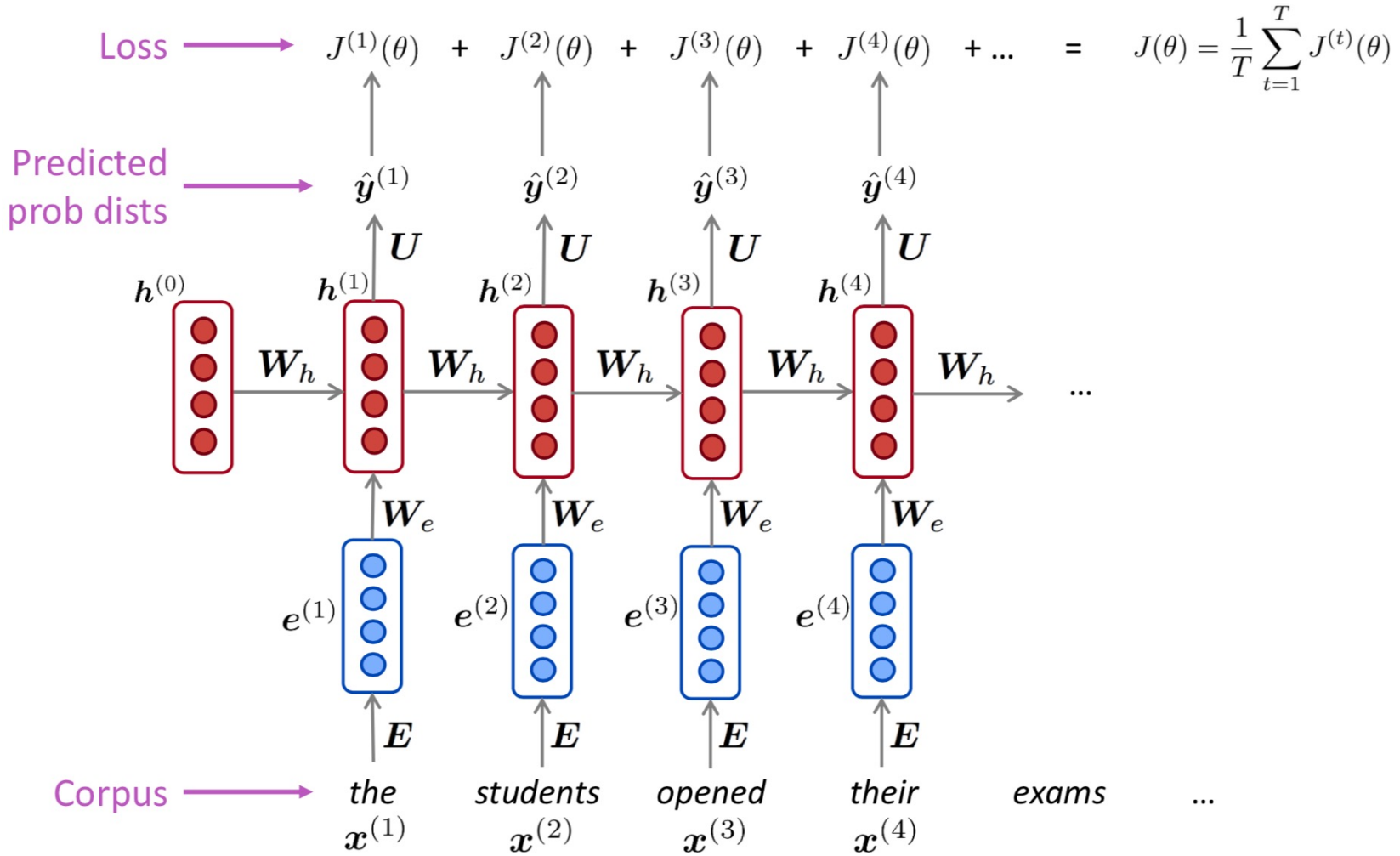
Train a RNN Language Model



Train a RNN Language Model



Train a RNN Language Model



Training a RNN Language Model

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

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

- In practice, consider $x^{(1)}$, $x^{(2)}$,…, $x^{(T)}$ 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.

Source: https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6

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

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

- How about a RNN with separate memory?

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM)

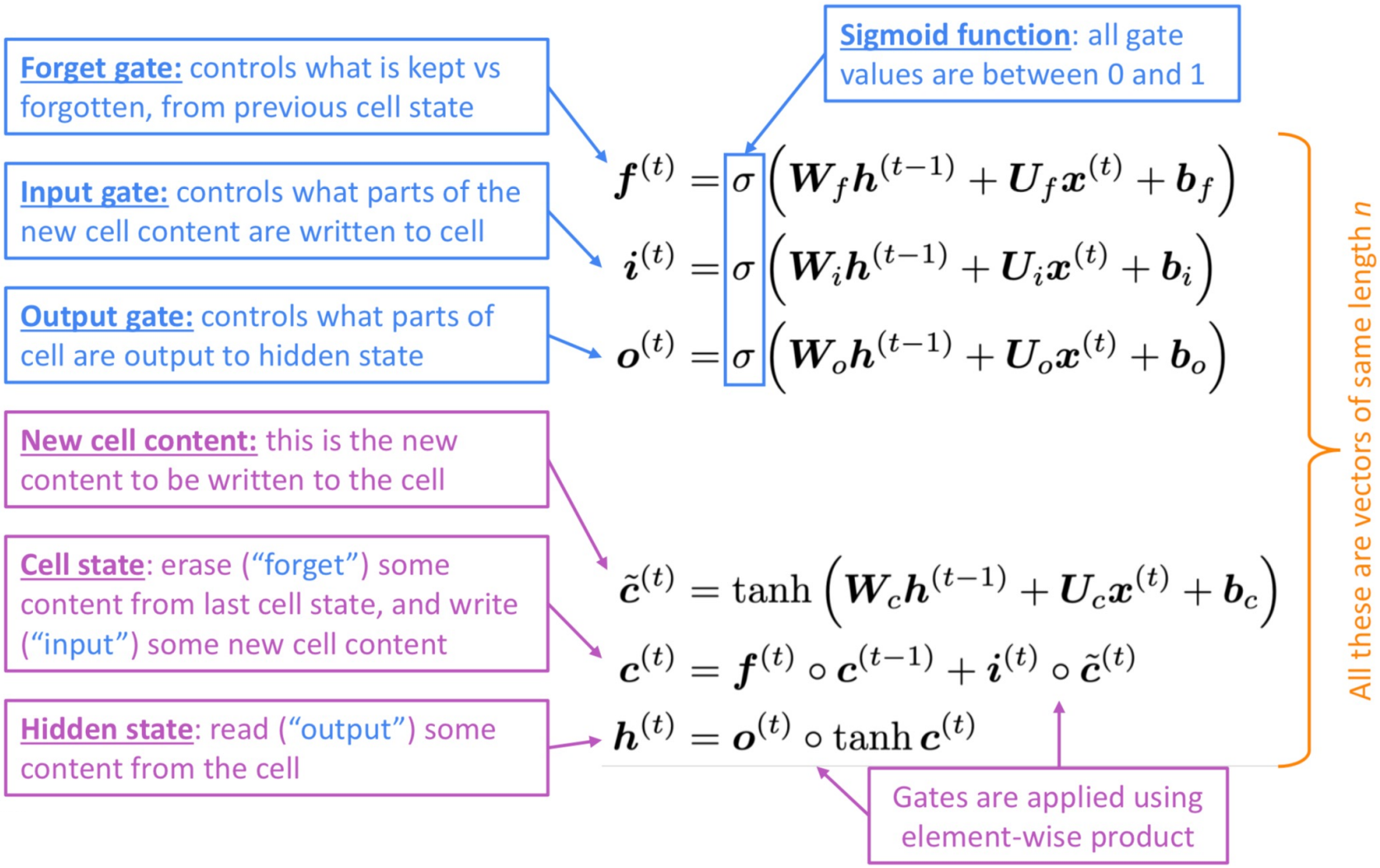
- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.

- On step t, there is a hidden state $h^{(t)}$ and a cell state $c^{(t)}$
  - Both are vectors length n
  - The cell stores long-term information
  - The LSTM can erase, write and read information from the cell

- The selection of which information is erased/written/read is controlled by three corresponding gates
  - The gates are also vectors length n
  - On each timestep, each element of the gates can be open (1), closed (0), or somewhere in-between.
  - The gates are dynamic: their value is computed based on the current context

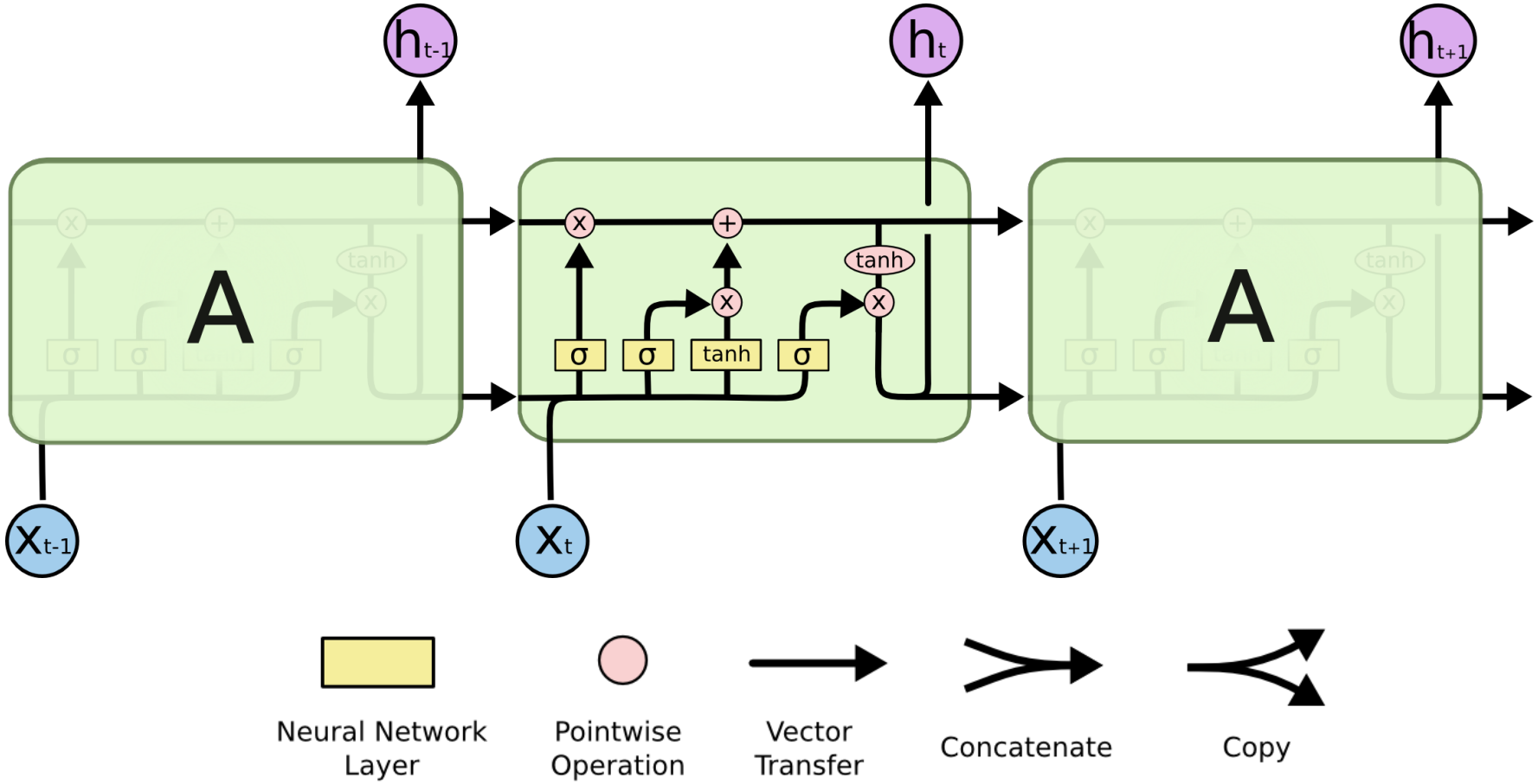
Long Short-Term Memory (LSTM)

- We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t:



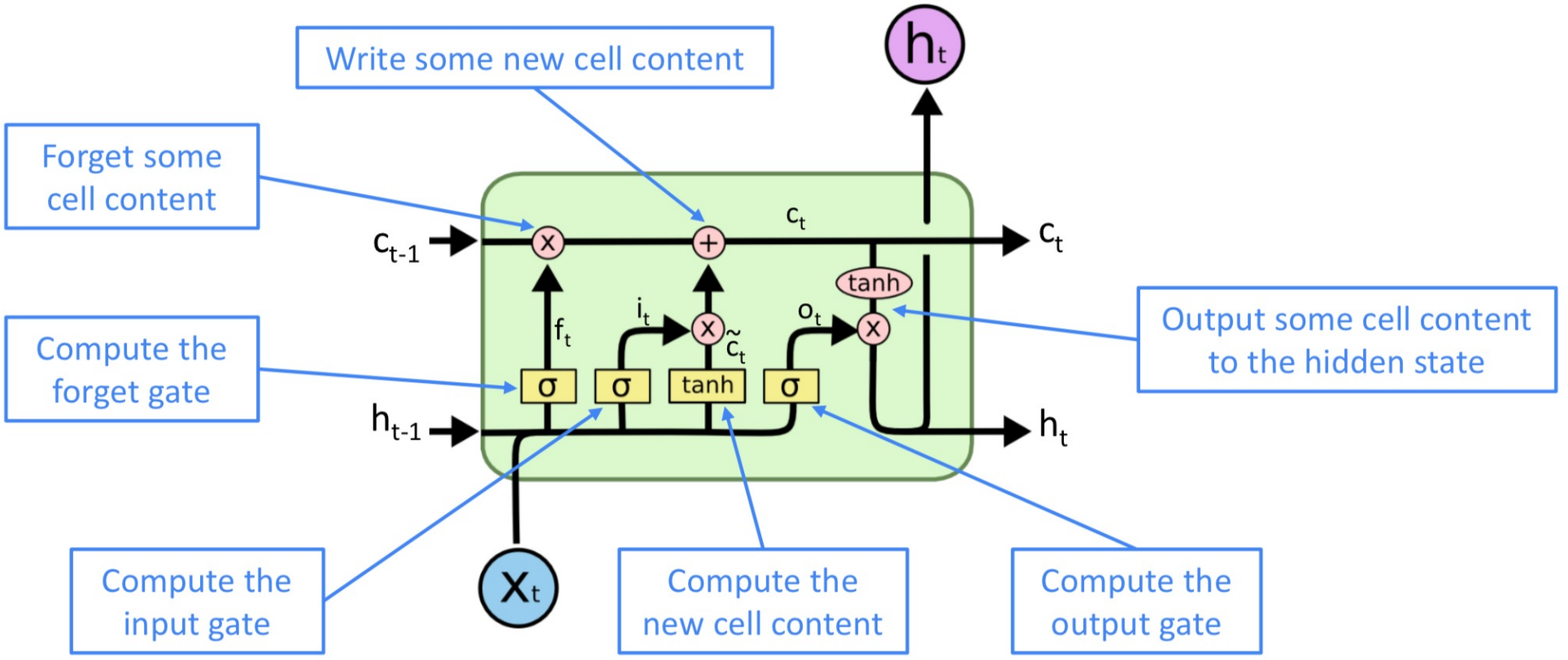
Long Short-Term Memory (LSTM)

- You can think of the LSTM equations visually like this:



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory (LSTM)



How does LSTM solve vanishing gradients?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
  - The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.
  - e.g. if the forget gate is set to remember everything on every timestep, then the information in the cell is preserved indefinitely

- By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves information in hidden state

- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

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 too process variable length sequences of inputs.
- LSTMs are powerful to combat the vanishing gradient