# 05 – LANGUAGE MODELING

## MACHINE LEARNING FOR NATURAL LANGUAGE PROCESSING, AIMS 2024

Lecture 05

Dr. Elvis Ndah

# LECTURE OBJECTIVE

- Why do we need language modeling.
- What is language modeling
- Formal definition of a language model
- Probabilistic (n-gram) Language Models
- Causal Language Model with RNN (LSTM)

# WHY DO WE NEED LANGUAGE MODELLING?

**Most NLP task (systems) build on text generation.**
Language models are essentially text generation models ("generative models").

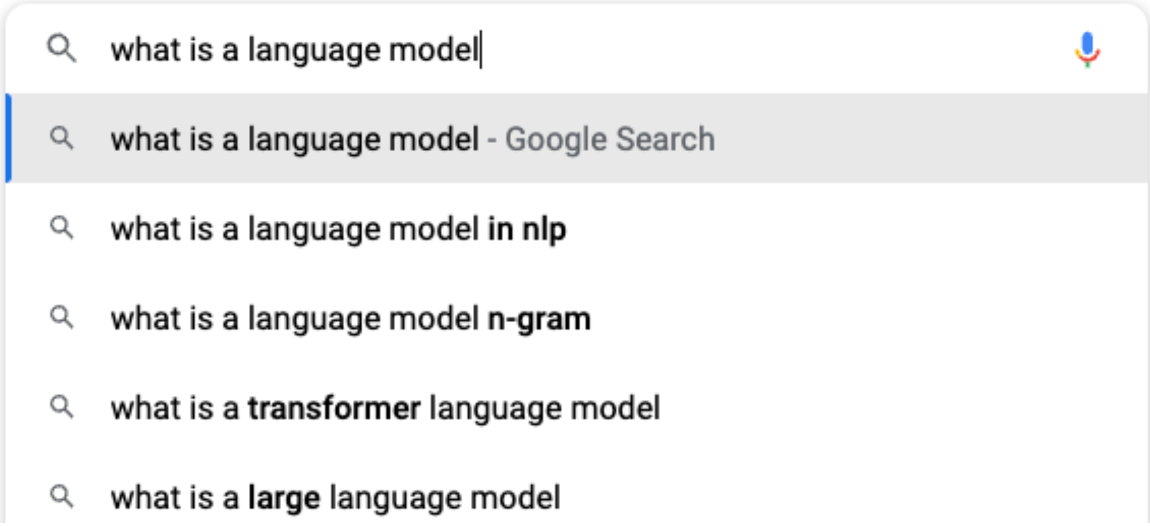
- ***Machine translation:*** *”vents violents ce soir”*
  *prob(high winds tonight) > prob(large winds tonight)*

- ***Speech recognition:*** *return a transcript of what was spoken*
  *prob(I saw a van) >> prob(eyes awe of an)*

- ***Spell-correction:*** *”the office is about fifteen minuets from my house”*
  *prob(about fifteen minutes from) > prob(about fifteen minuets from)*

- Summarization, question answering, etc.

# WHY DO WE NEED LANGUAGE MODELLING?



# WHAT IS LANGUAGE MODELLING?

Given a finite vocabulary V of words or tokens and let Ω be a set of sequences of words from V, we define a formal language *Ω* as over all possible sequence of words from the vocabulary ***V***

*$\forall x \in \Omega$, x is called a sentence*

**Formal definition:**

A *Language Modeling* is a defined as a probability distribution over a sequence of words (tokens) from the formal language *Ω*

$$P : \Omega \rightarrow [0, 1] \qquad \sum_{x \in \Omega} P(x) = 1$$

# WHAT IS LANGUAGE MODELLING?

- Given a formal language, a language model aims to answer the following questions
  - *Which sequence of tokens are more likely?*
  - *How can we assign a probability distribution to each sequence of tokens?*

- Consider the following sequence of tokens from English Language
  - *I would like to eat.*
  - *I would like eat to.*
  - *I like would eat to.*
  - *I to like would eat.*

# WHAT IS LANGUAGE MODELLING?

**Given a well-defined language model:**

- We expect that regular and grammatically correct sentences will occur more often in text and speech than other weird sequences.
- They should have higher probabilities scores.
  - *prob(I would like to eat) > prob(I like would eat to)*
  - *prob(I would like to eat) > prob(I like would eat to)*
  - *prob(I would like to eat) > prob(I to like would eat)*

# PROBABILISTIC LANGUAGE MODEL

# PROBABILISTIC LANGUAGE MODEL

- Given a sequence of tokens $x^1, x^2, \dots, x^t$ from a vocabulary $V$
- How do we compute the probability of a sequence of tokens $x^1, x^2, \dots, x^t$ from a vocabulary $V$?
- The probability of $x^1, x^2, \dots, x^t$ is the joint probability $P(x^1, x^2, \dots, x^t)$
- Estimated using the chain rule of probability:

$$
\begin{aligned}
P(x^1, x^2, \dots, x^{t-1}, x^t) &= P(x^1 | x^2, \dots, x^{t-1}, x^t) P(x^2, \dots, x^{t-1}, x^t) \\
&= P(x^1) P(x^2 | x^1) P(x^3 | x^1, x^2) \dots . P(x^t \,| x^1, \dots, x^{t-1}) \\
&= P(x^1) \prod_{k=1}^{t} P(x^k | x^1, \dots, x^{k-1})
\end{aligned}
$$

# PROBABILISTIC LANGUAGE MODEL

- A language model is a distribution over all word-sequences $x^1, x^2, \dots, x^n$ in a vocabulary $V$

$$\sum_{\langle x^1, x^2, \dots, x^n \rangle} P(x^1, x^2, \dots, x^n) = 1$$

- By the chain rule of probability is given by

$$P(x^1, x^2, \dots, x^{t-1}, x^t) = P(x^1) \prod_{k=1}^{t} P\left(x^k \middle| x^1, \dots, x^{k-1}\right)$$

- To estimate the probability of each sequence we need to
  - First estimate $P(x^1)$
  - Estimate probabilities $P\left(x^k \middle| x^1, \dots, x^{k-1}\right)$ for all $x^1, \dots, x^k$

# PROBABILISTIC LANGUAGE MODEL

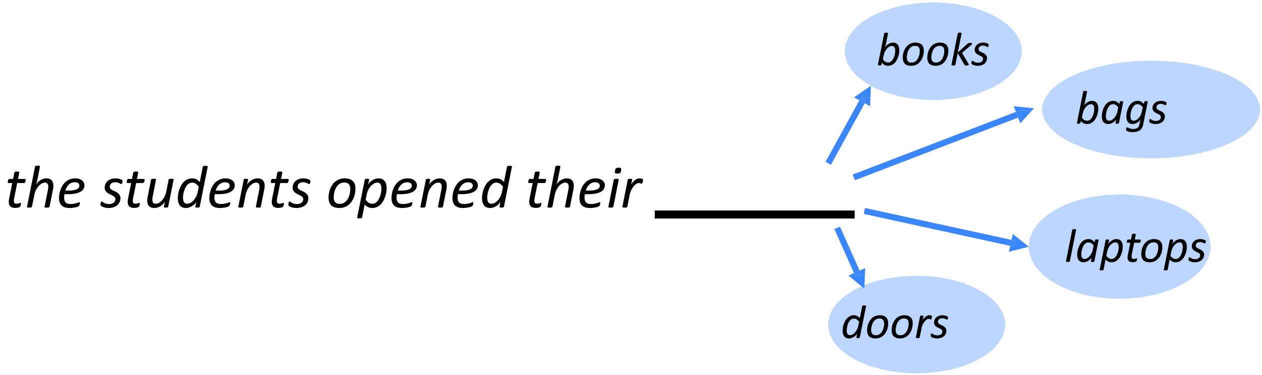
- Consider the sequence of token, *the student open their books* from the English vocabulary
- How do we compute the probability of *P(the student open their books)*

$$P(the\ student\ open\ their\ books) = P(books|the\ student\ open\ their) * \\ P(their|the\ student\ open) * \\ P(open|the\ student) * \\ P(student|the) * \\ P(the)$$

# PROBABILISTIC LANGUAGE MODEL

**Language Modeling** can be reduced to the task of predicting the next

$$P(x^1, x^2, \ldots, x^{t-1}, x^t) = P(x^1) \prod_{n=2}^{t} \boldsymbol{P(x^n | x^1, \ldots, x^{n-1})}$$




Given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$ compute the probability distribution of the next word $x^{(t+1)}$

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

# ESTIMATING PROBABILITIES – N-GRAM

- N-gram language models are based on probabilities of chunks of word.

*The student open their*

- An n-gram is a chunk of n consecutive words.
  - *Uni-gram: unit of single word - the, student, opened, their*
  - *Bi-grams: unit of double words - the student, student opened, opened their*
  - *Tri-gram: unit of triple words - the student opened, student opened their*
  - *Four-gram: unit of 4 words - the student opened their*

# PROBABILISTIC LANGUAGE MODEL

- Probabilistic language models are based on the grouping of words into chunks called n-grams in order to estimate

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

- **The main idea:**
  - collect statistics about the frequency of different *n-grams*
  - n-gram probabilities are estimated by counting the frequency of occurrence
  - use this to estimate the next word given a history of observed words

# ESTIMATING PROBABILITIES

- Relative frequency from a corpus

$$P(x_i|x_1, \dots, x_{i-1}) = \frac{count(x_1, \dots, x_{i-1}, x_i)}{count(x_1, \dots, x_{i-1})}$$

$$= \frac{count(x_1, \dots, x_{i-1}, x_i)}{\sum_{x \in V} count(x_1, \dots, x_{i-1}, \ x)}$$

- We estimate probability for all sequences of length *i*
- Suppose /V/ = 1000,
  - all sentences are approximately 10 word long
  - then we need to estimate $1000^{10}$ probabilities.

• **NOTE:** No corpus is large enough to obtain an unbiased estimate of the probabilities

# ESTIMATING PROBABILITIES – MARKOV ASSUMPTION

**Markov assumption**

- Independent and identical trials
- There is a fixed and finite *k* such that all word depends only on the preceding *k-1* words

$$P(x_{i+1}|x_1, \dots, x_i) \approx P(x_{i+1}|x_{i-k}, \dots, x_i)\ \forall k \geq 0$$

- Model: an $k^{th}$- order Markov model
- n-gram: statistics of an k-order Markov model is k $+1 - gram$ model

$$P(x_i|\boxed{x_{i-k}}, \dots, x_{i-1}) = \frac{count(\boxed{x_{i-k}} \dots, x_{i-1}, x_i)}{\sum_{x \in V} count(\boxed{x_{i-k}} \dots, x_{i-1},\ x)}$$

# ESTIMATING PROBABILITIES – MARKOV ASSUMPTION

The order of a Markov model is defined by the length of its history or *n-gram* *(n = k+1)*

$0^{th} - order$: $P(x_1, \ldots, x_n) \approx P(x_1) \prod_{i=1}^{n-1} P(x_{i+1}) \approx P(x_1)P(x_2) \ldots P(x_n)$

|history| = 0

$1^{st} - order$: $P(x_1, \ldots, x_n) \approx P(x_1) \prod_{i=1}^{n-1} P(x_{i+1} | x_i)$

|history| = 1

$2^{nd} - order$: $P(x_1, \ldots, x_n) \approx P(x_1) \prod_{i=1}^{n-1} P(x_{i+1} | x_i, x_{i-1})$

|history| = 2

$k^{th} - order$: $P(x_1, \ldots, x_n) \approx P(x_1) \prod_{i=1}^{n-1} P(x_{i+1} | x_i, \ldots, x_{i-k})$

|history| = k

# ESTIMATING PROBABILITIES – FROM N-GRAM PROBABILITIES

In a trigram model

$$P(x^1x^2x^3) = P(x^1)P(x^2|x^1)P(x^3|x^1x^2)$$

- The only trigram is $P(x^3|x^1x^2)$
- $P(x^1)$ and $P(x^2|x^1)$ are not trigrams thus are from a different probability distribution

- **Solution:**

  add *n-1* beginning of sentence (**<s>**) symbols

  <s><s>$x^1x^2x^3$……

  similarly add *n-1* end of sentence symbols

  …..$x^1x^2x^3$ </s></s>

# ESTIMATING PROBABILITIES – FROM N-GRAM PROBABILITIES

- With the start or end of sentence token(s) we define a new vocabulary

$$V^* = V \; U \; \{</s>\}$$
$$or$$
$$V^* = V \; U \; \{<s>\}$$

- With the new vocabulary we can get a single distribution over strings of any length

- **Why?**
  - because P(</s>|…) will be high enough that we are always guaranteed to stop after generating a finite number of words.

# ESTIMATING PROBABILITIES – EXAMPLE

Maximum likelihood estimate

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

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

Example

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

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

$P(\texttt{I} \mid \texttt{<s>}) = \frac{2}{3} = .67$ $\quad P(\texttt{Sam} \mid \texttt{<s>}) = \frac{1}{3} = .33$ $\quad P(\texttt{am} \mid \texttt{I}) = \frac{2}{3} = .67$

# ESTIMATING PROBABILITIES - PRACTICAL ISSUES

- We do everything in log space
  - Avoid underflow
  - Computationally adding is faster than multiplying

$$\log(p_1 \times p_2 \times p_3 \times p_4) = \log p_1 + \log p_2 + \log p_3 + \log p_4$$

# ESTIMATING PROBABILITIES - PRACTICAL ISSUES

- Estimating the number of parameter per n-gram language model.
- Given a vocabulary V of $|V|$ unique tokens, where $|V| = 10^4$

  - **Unigram** model: $|V|$ parameters ⇔ $10^4$ parameters
  - **Bigram** model: $|V|^2$ parameters ⇔ $10^8$ parameters
  - **Trigram** model: $|V|$ parameters ó $10^{12}$ parameters

# ESTIMATING PROBABILITIES – SHAKESPEARE AS CORPUS

- Number of words (symbols) = 884,647
- Tokens, V=29,066
- Shakespeare produced 300,000 bigrams
- bigram types out of $V^2$= 844 million possible bigrams
- So, 99.96% of the possible bigrams were never seen (*have zero entries in the table*)

  *844 million – 300,000 unused bigrams*
- Quadrigrams worse:
  - What's coming out looks like Shakespeare because it ***is*** Shakespeare

# EVALUATION: HOW GOOD IS OUR MODEL?

- Does our language model prefer good sentences to bad ones?
- Assign higher probability to "real" or "frequently observed" sentences
  - than "ungrammatical" or "rarely observed" sentences?
- We train parameters of our model on a **training set**.
- We test the model's performance on data we haven't seen.
  - A **test set** is an unseen dataset that is different from our training set, totally unused.
  - An **evaluation metric** tells us how well our model does on the test set.

# EXTRINSIC EVALUATION OF N-GRAM MODELS

- Best evaluation for comparing models A and B
- Embed each model in a task
  - spelling corrector,
  - speech recognizer,
  - Machine Translation system
- Run the task, get an accuracy for A and for B
  - How many misspelled words corrected properly
  - How many words translated correctly
- Compare accuracy for A and B

Time-consuming; can take days, weeks or even months

# INTRINSIC EVALUATION

- Sometimes use **intrinsic** evaluation: **perplexity**
- Bad approximation
  - unless the test data looks **just** like the training data
  - So **generally, only useful in pilot experiments**
- But is helpful to think about.

# PERPLEXITY

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

$$perplexity(W) = P(x_1 x_2, \ldots, x_N)^{-1/N} = \sqrt[N]{\frac{1}{P(x_1 x_2, \ldots, x_N)}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(x_i | x_1, \ldots, x_{i-1})}}$$

- For unigram $perplexity(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(x_i)}}$

- For bigram $perplexity(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(x_i | x_{i-1})}}$

NB: Minimizing perplexity is the same as maximizing probability

# LIMITATIONS – STORAGE PROBLEMS

**Storage Problem:**
The need to store count for all n-grams you saw in the corpus.

$$P(\boldsymbol{w}|\ I\ want\ to\ eat) = \frac{\boxed{count(I\ want\ to\ eat\ \boldsymbol{w})}}{count(I\ want\ to\ eat)}$$

Increasing *n* or increasing corpus size <=> increases model size

# LIMITATIONS - SPARSITY PROBLEM

**Sparsity Problem 1:**

- What if "*I want to w*" never occurred in the corpus?
- Then *w* has probability 0

**Solution - smoothing:** Add small $\delta$ to the count for every $w \in V$.

$$P(\boldsymbol{w} | waiting\ for\ the) = \frac{\boxed{count(I\ want\ to\ \boldsymbol{w})}}{\boxed{count(I\ want\ to)}}$$

**Sparsity Problem 2:**

- What if "*I want to*" never occurred in the corpus?
- Then we cannot calculate probabiliity of any *w.*

**Solution - backoff:** condition on the (n-1)-gram, i.e. "*I want*"

Larger *n* makes sparsity problem worse. Typically *n* should be less than or equal to 5

# THE PERILS OF OVERFITTING - SPARSITY

N-grams only work well for word prediction if the test corpus looks like the training corpus.

- In real life, it often doesn't
  - We need to train robust models that generalize!

- One kind of generalization: Zeros!
  - Things that don't ever occur in the training set but occur in the test set

# SMOOTHING METHODS

Smoothing methods

- Additive smoothing
- Good-Turing estimate
- Jelinek-Mercer smoothing (interpolation)
- Katz smoothing (backoff)
- Witten-Bell smoothing
- Absolute discounting
- Kneser-Ney smoothing

# NEURAL LANGUAGE MODEL

# NEURAL LANGUAGE MODEL

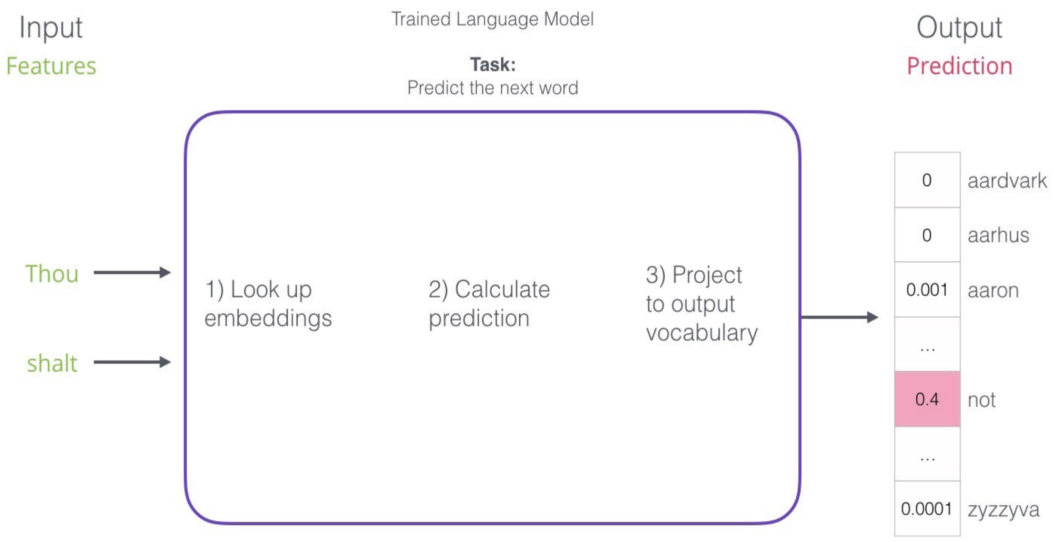
Language Modeling task:

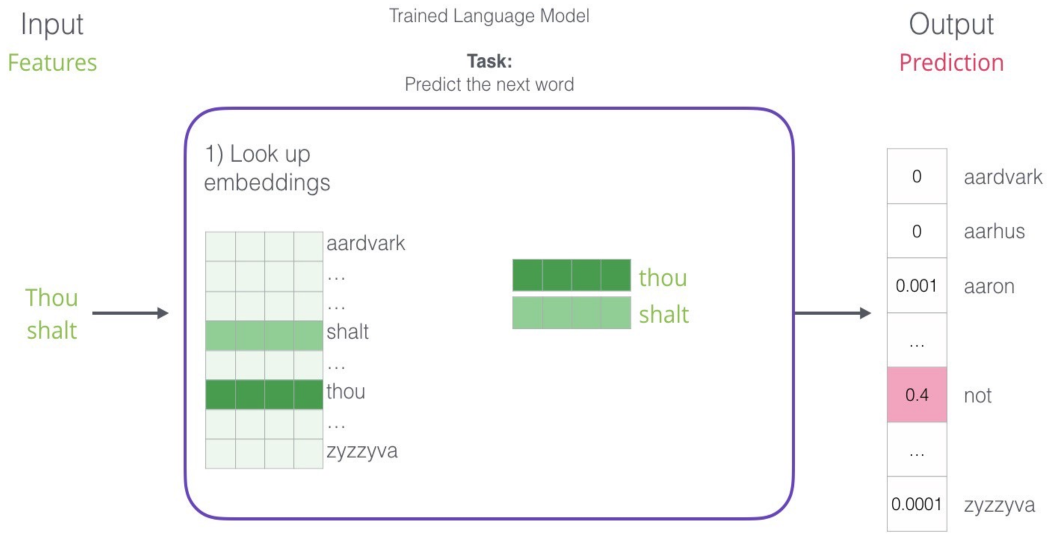
- Input: sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$
- Output: prob dist of the next word $P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$

Neural network language model

- Fixed window neural network
- Recurrent neural network (RNN) based
- Encoder-Decoder Architecture

# NEURAL LANGUAGE MODEL – RELEVANT STEPS

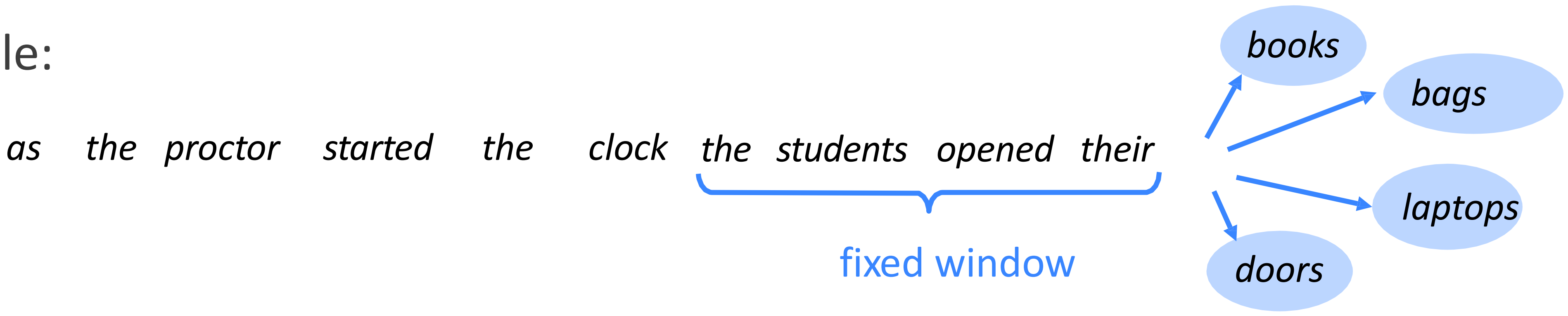


# NEURAL LANGUAGE MODEL – FIXED WINDOW

- Fixed window based define a sliding window of +/- n words
- Similarly, to n-gram
- Train a neural network to predict next word

- Example:



# A FIXED-WINDOW NEURAL LANGUAGE MODEL

output distribution

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

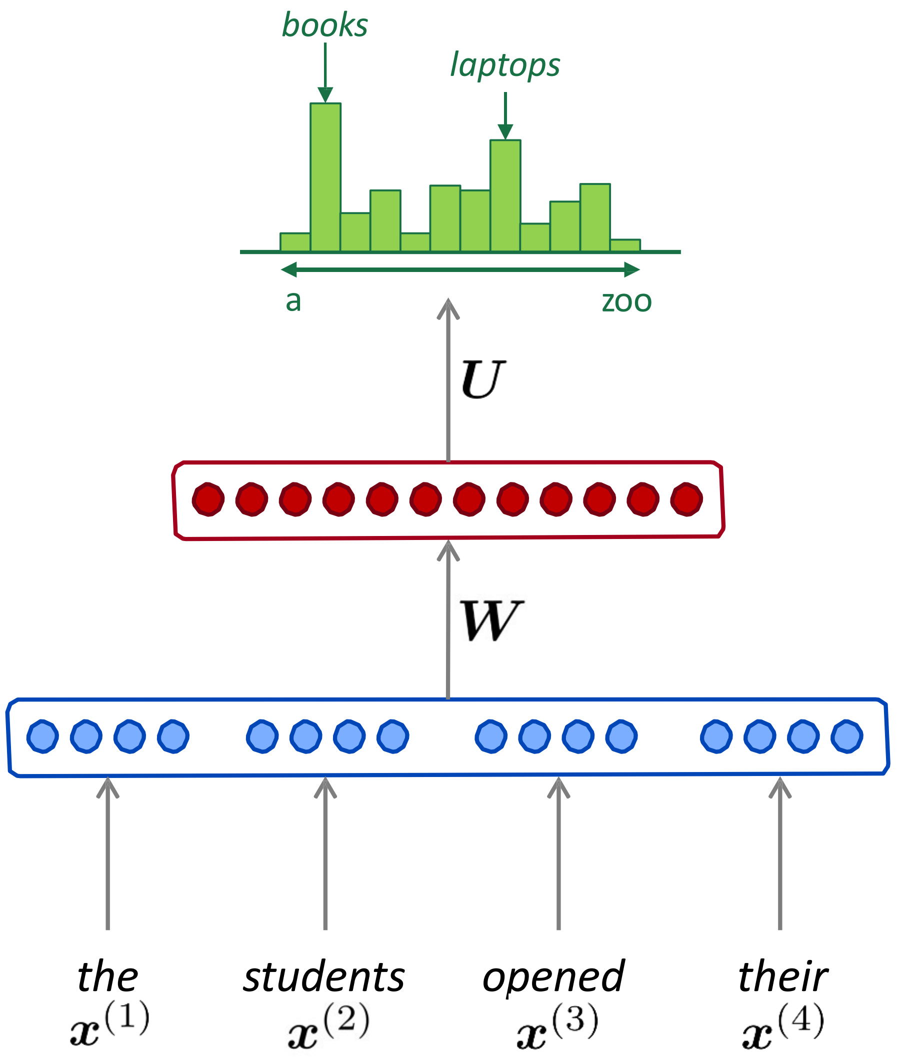
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

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






Credits: Chris Manning

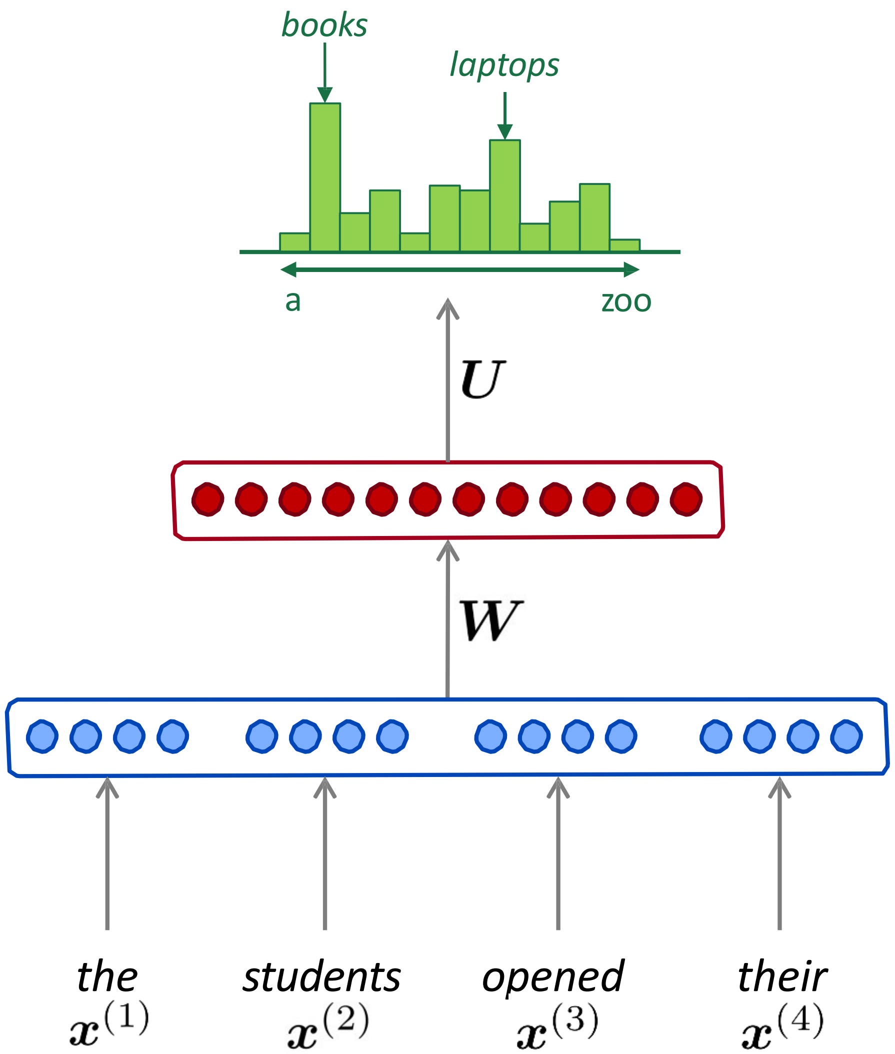
# A FIXED-WINDOW NEURAL LANGUAGE MODEL

- **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 $\boldsymbol{W}$
- Window can never be large enough!
- $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ are multiplied by completely different weights in $\boldsymbol{W}$. No symmetry in how the inputs are processed.

We need a neural architecture that can process *any length input*

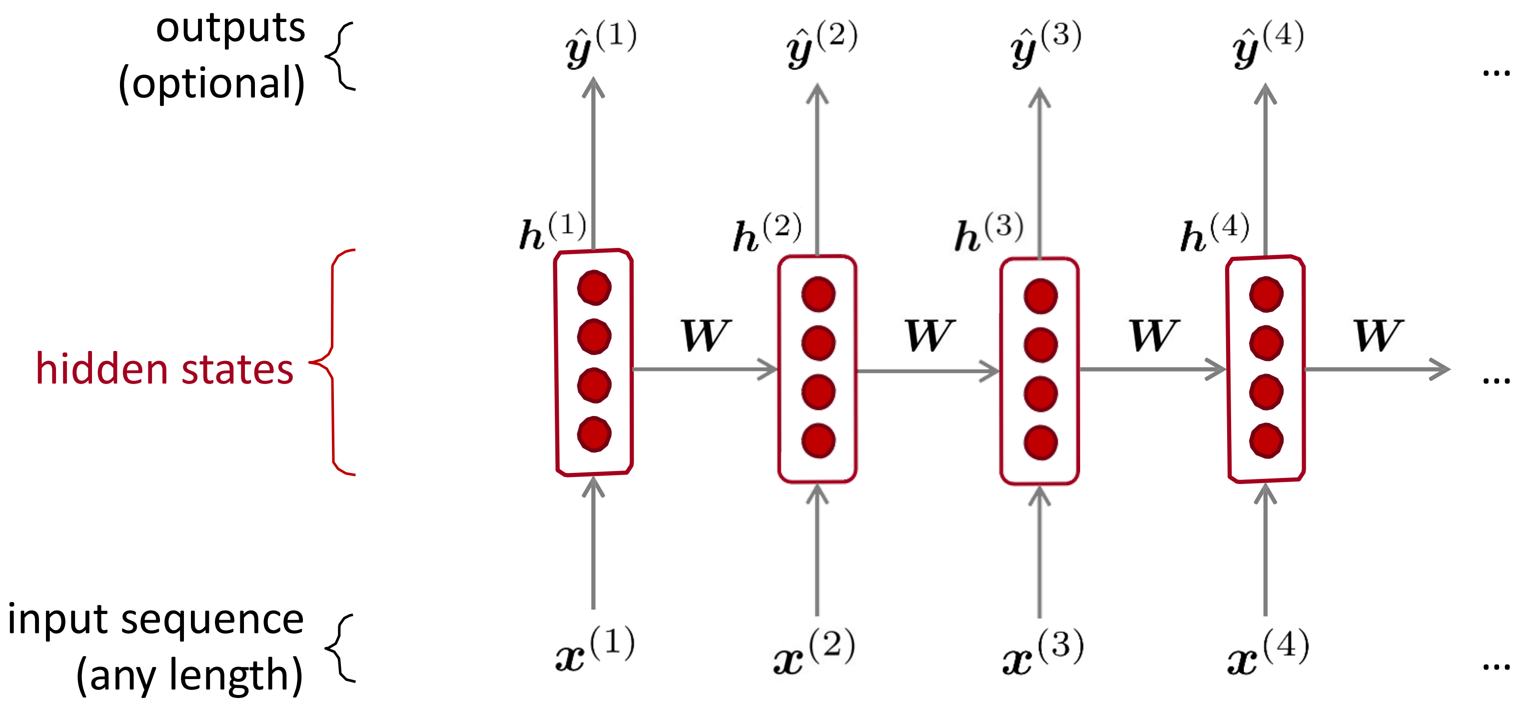





Credits: Chris Manning

# RECURRENT NEURAL NETWORKS (RNN)

- RNN apply the same weights (W) repeatedly
- Hidden state *$h^{(t)}$* depends on the output of the previous state *$h^{(t-1)}$*, *$h^{(t-1)}$* is a variant of *$h^{(t)}$*.






*Credits: Chris Manning*

# RECURRENT NEURAL NETWORKS (RNN)

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

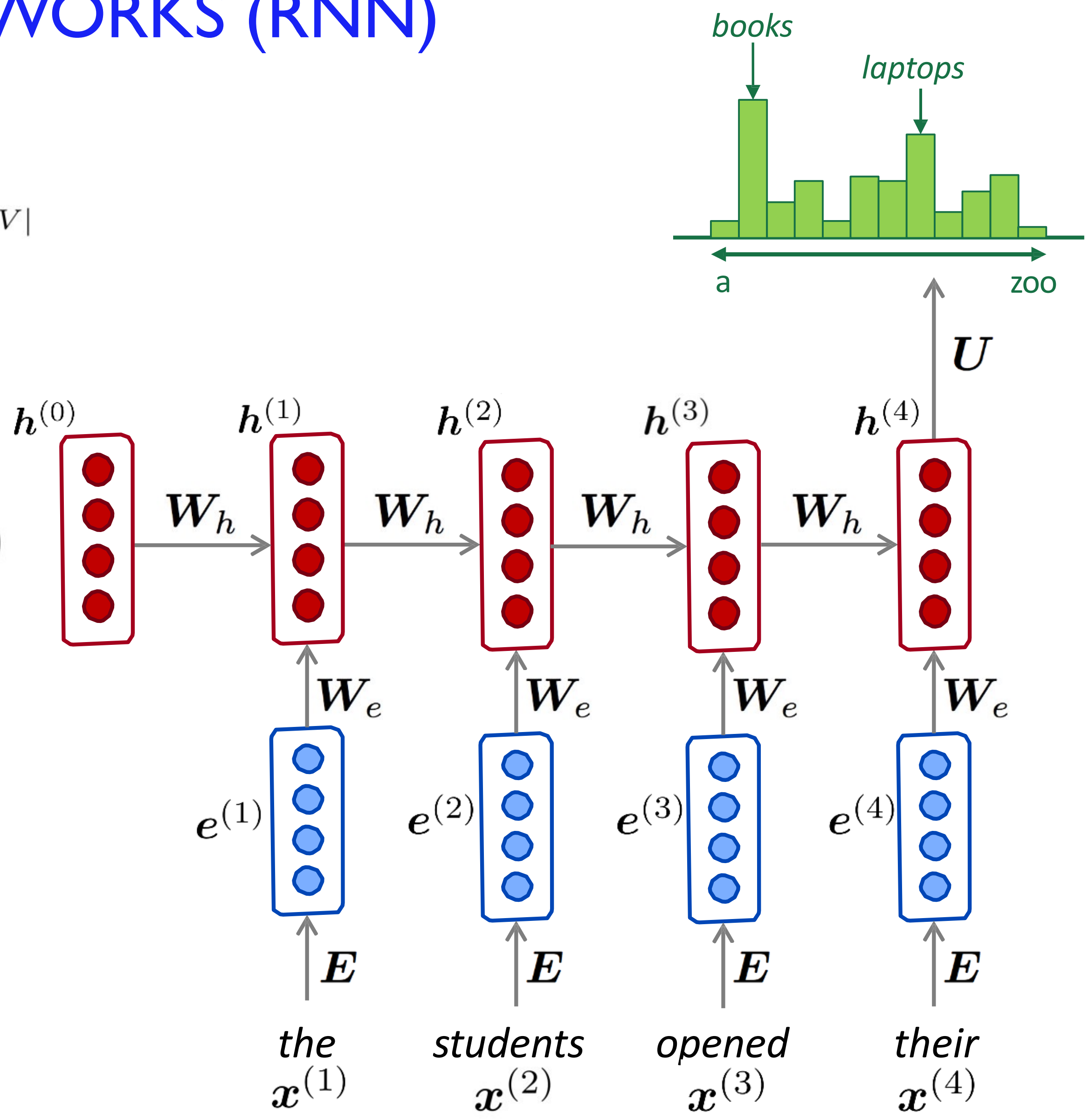
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



# TRAINING AN RNN LANGUAGE MODEL

- Get a sufficiently large corpus of text which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$
- Feed the sequence of text into a RNN-LM
- Compute output distribution $\hat{y}^{(t)}$ for every time step t.
  - i.e., predict probability distribution of every word given so far

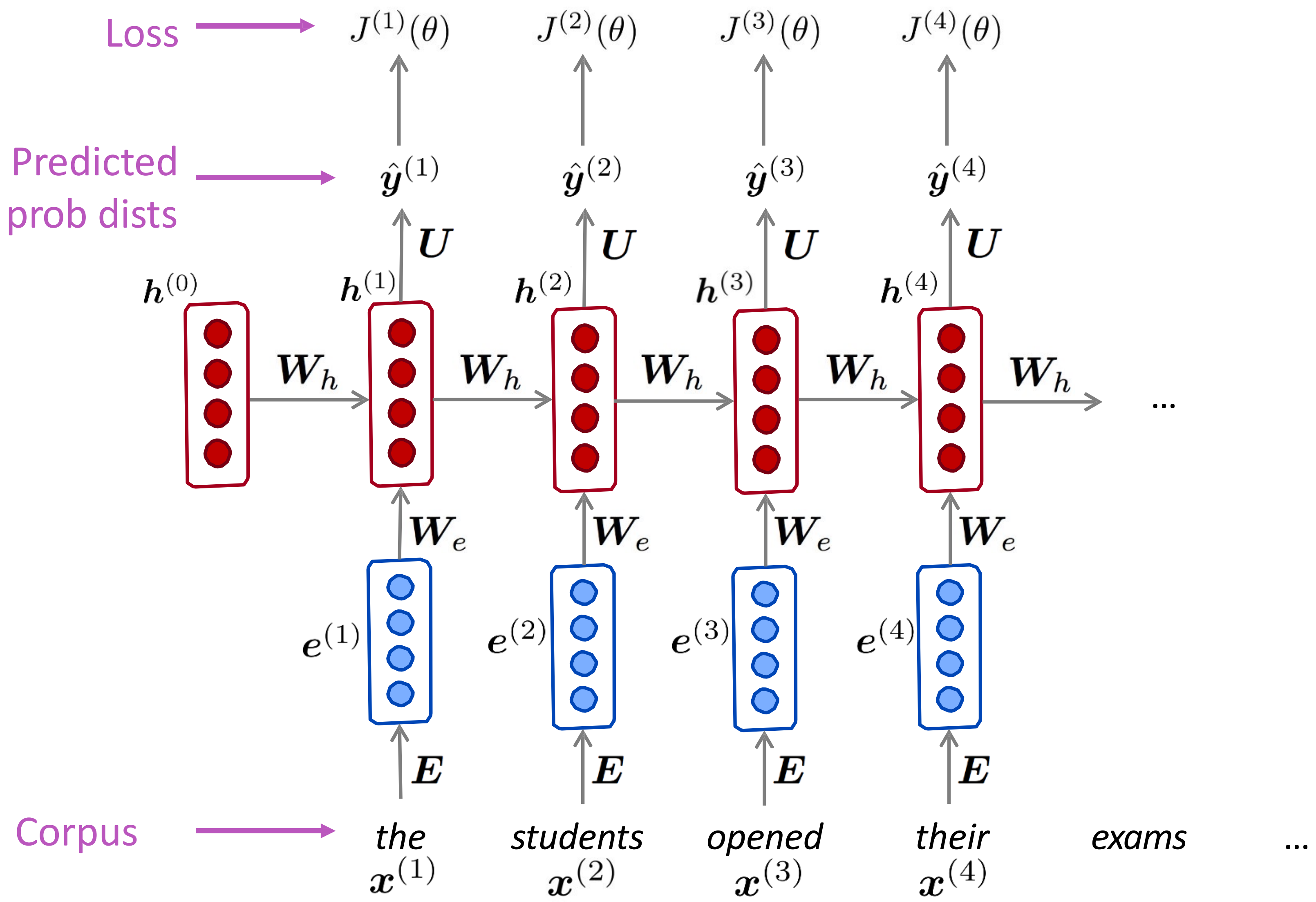
- **Loss function:** cross entropy between predicted probability distribution $\hat{y}^{(t)}$, and the next word vector $y^{(t)}$

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Average this to get the **overall loss** for entire training set

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

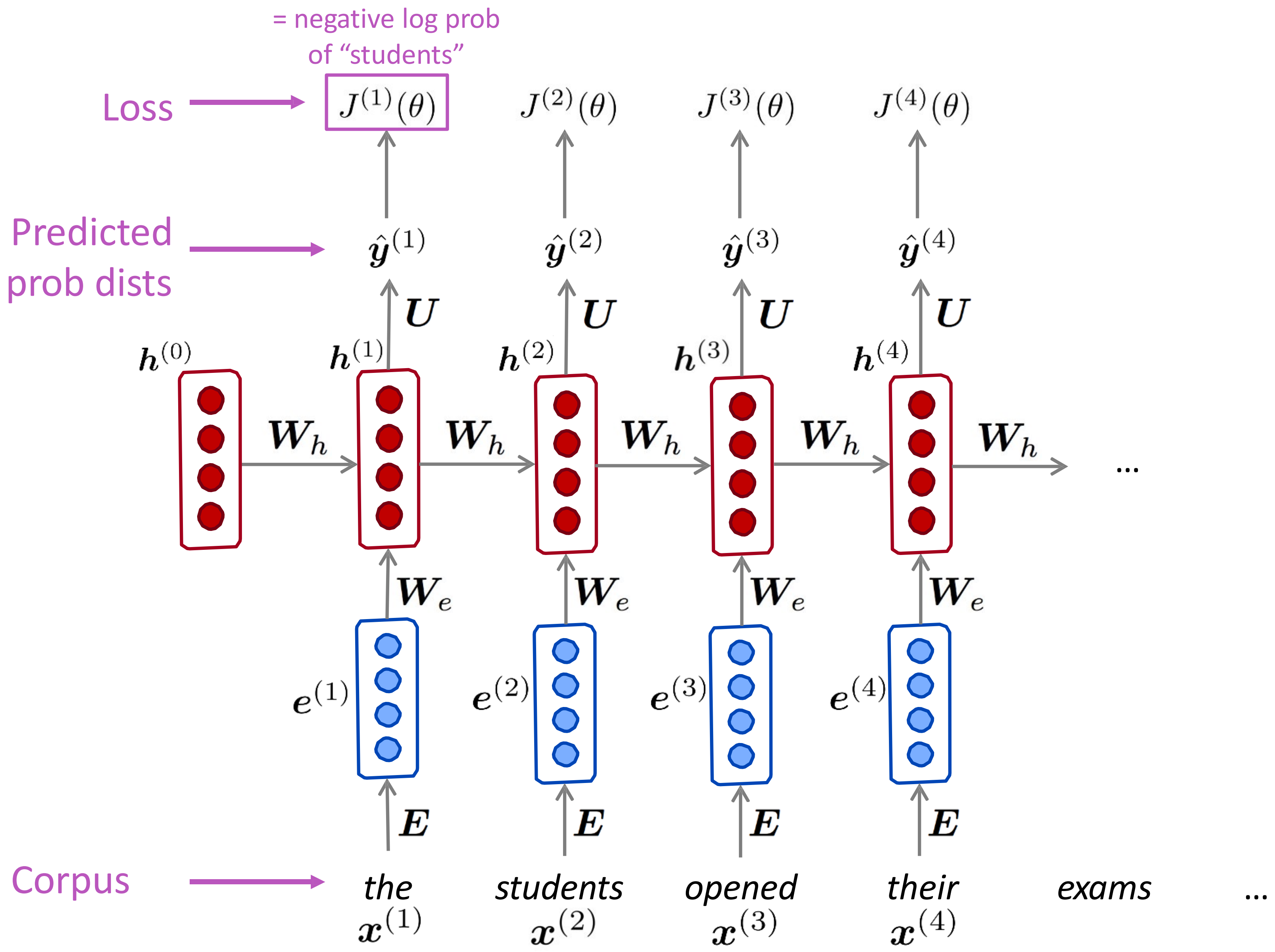
# TRAINING RNN LANGUAGE MODEL






Credits: Chris Manning

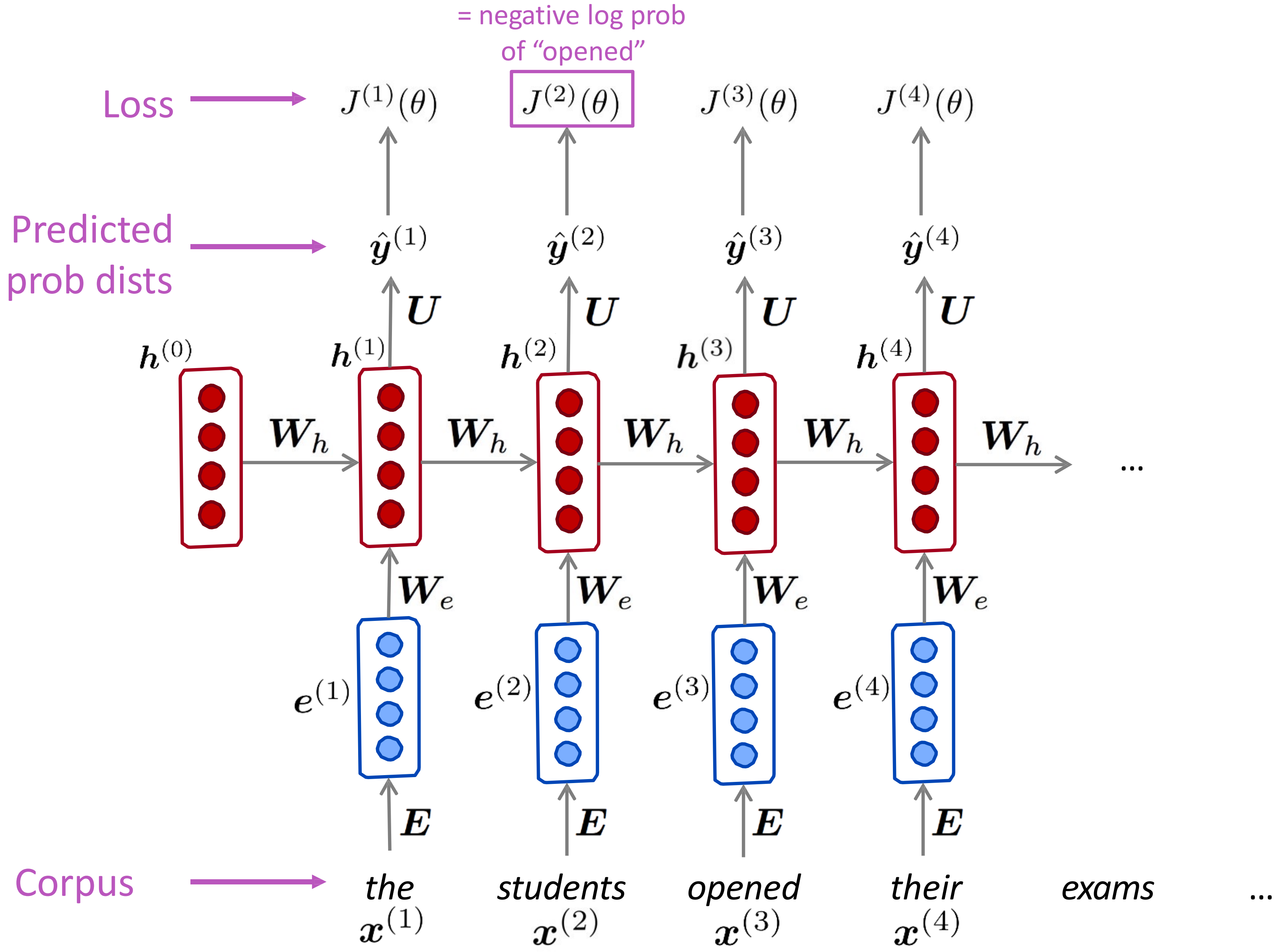
# TRAINING A RNN LANGUAGE MODEL





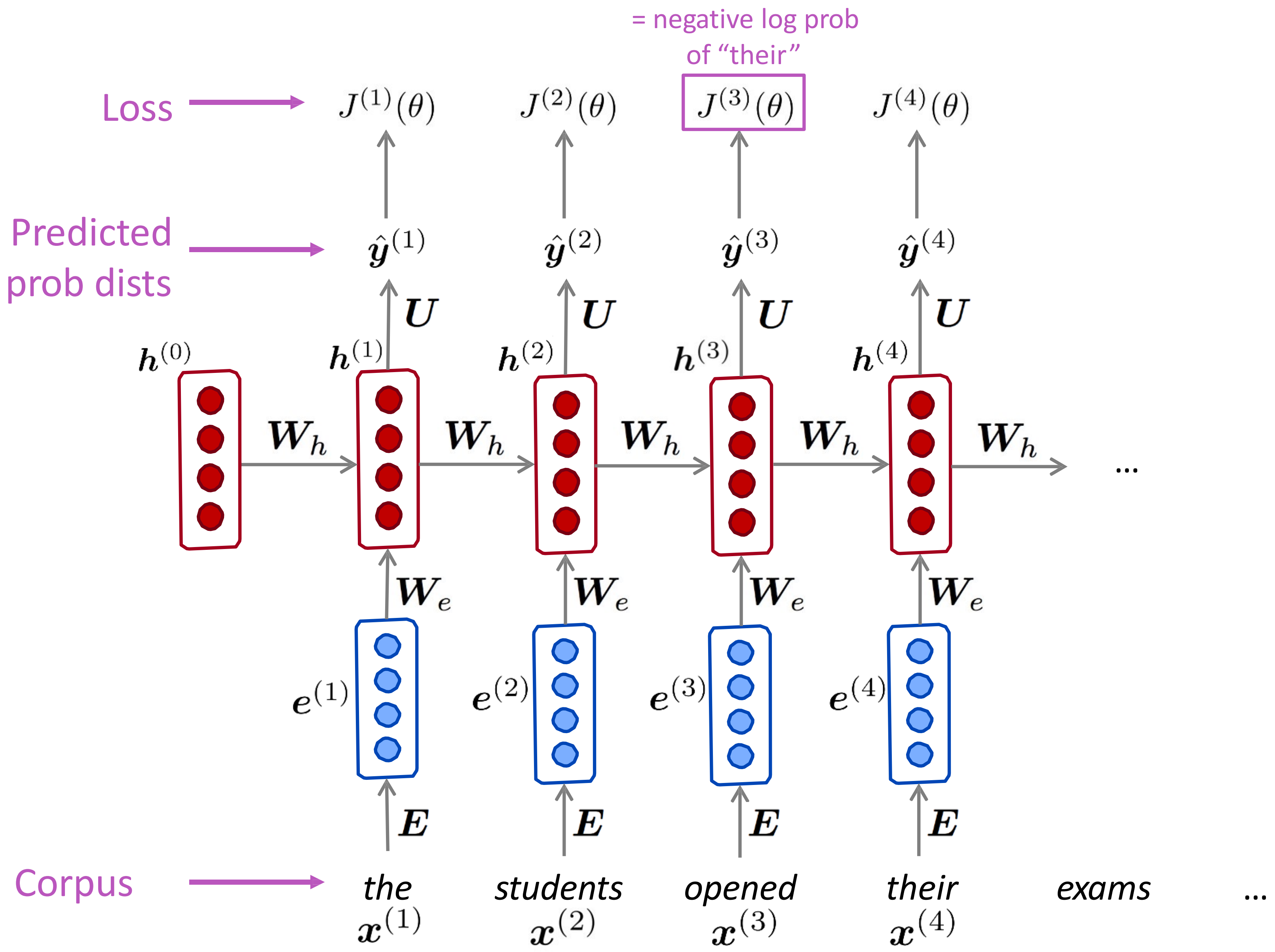

Credits: Chris Manning

# TRAINING A RNN LANGUAGE MODEL




Credits: Chris Manning

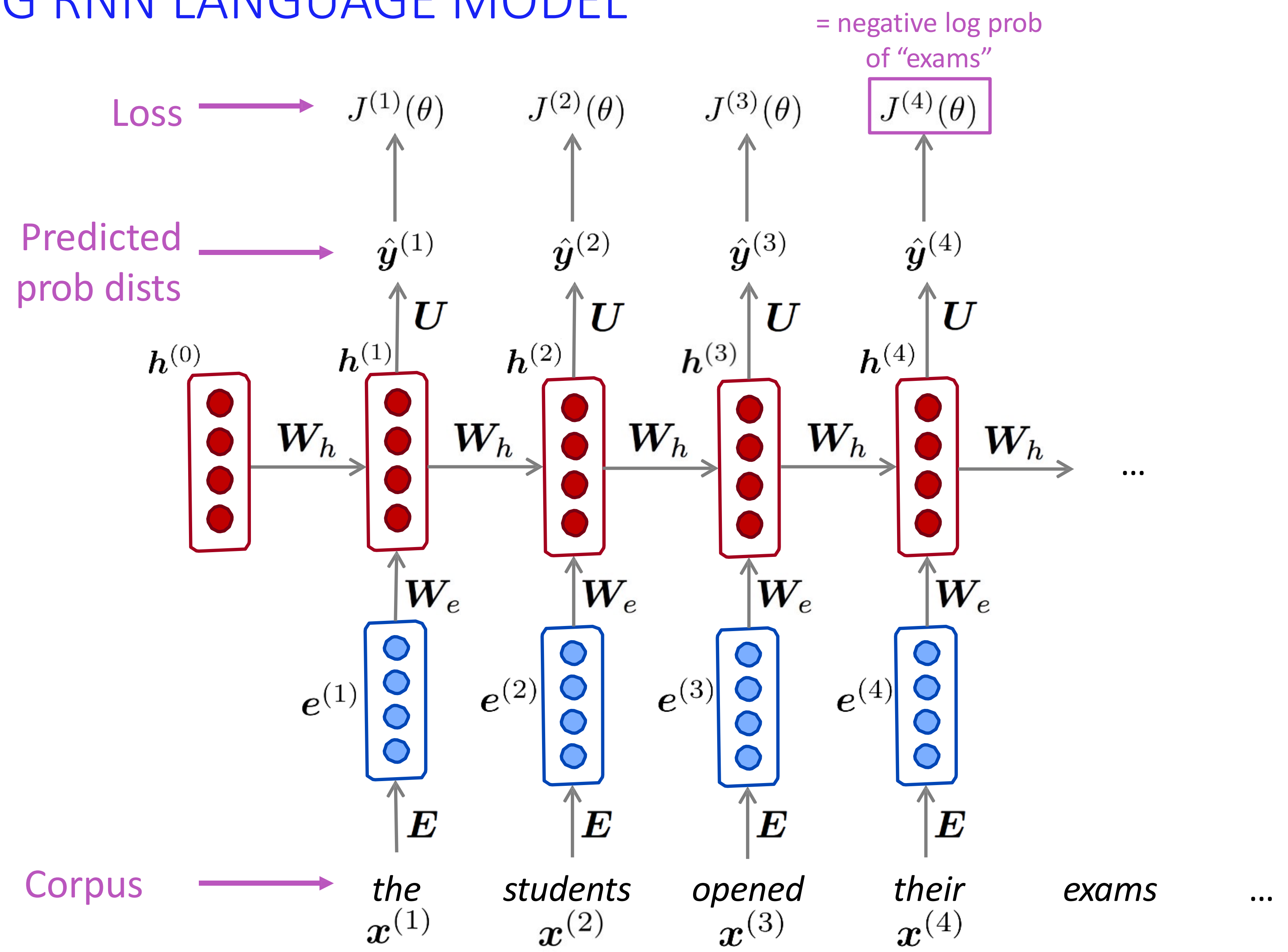
# TRAINING RNN LANGUAGE MODEL






Credits: Chris Manning

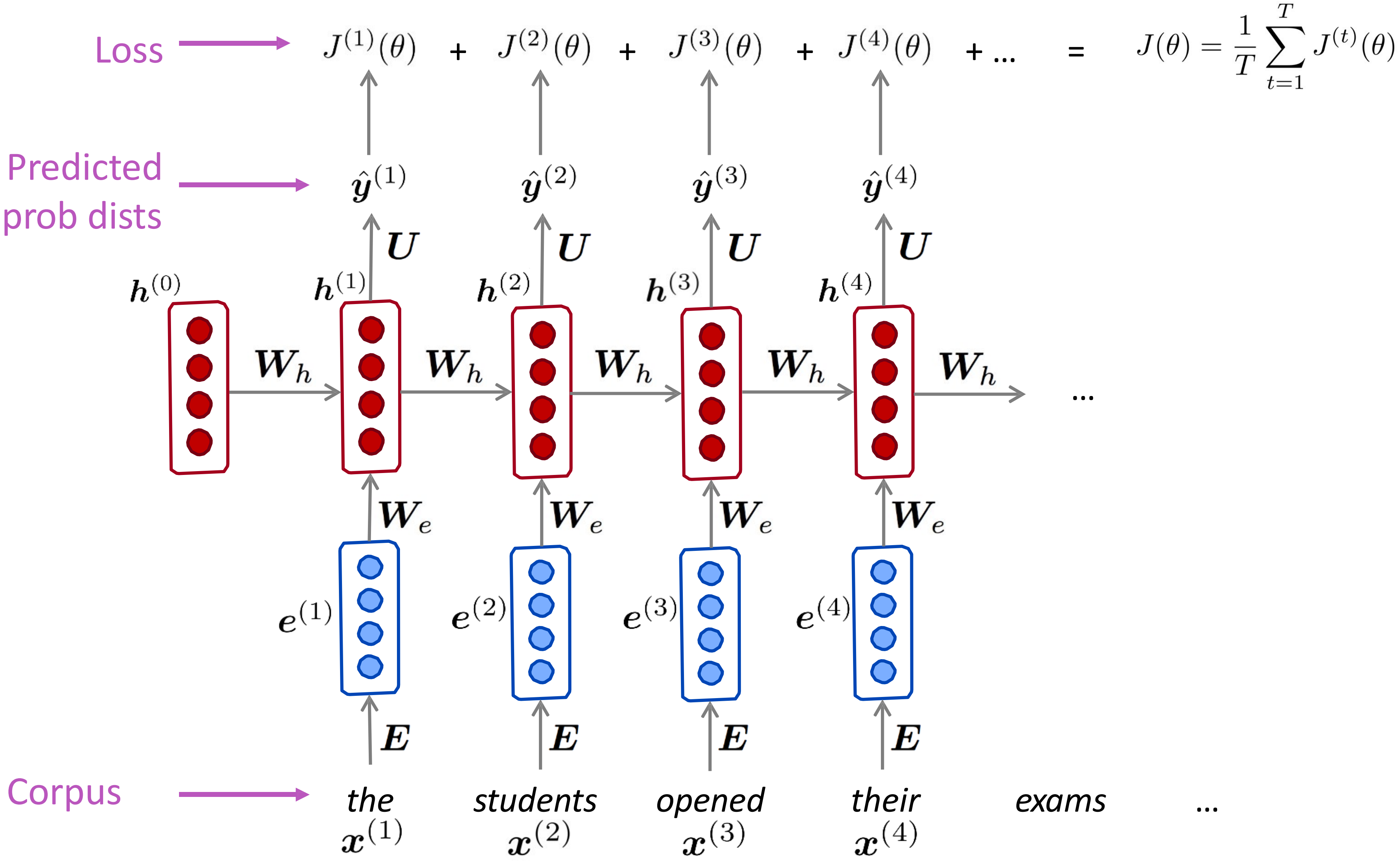
# TRAINING RNN LANGUAGE MODEL






Credits: Chris Manning

# TRAINING RNN LANGUAGE MODEL






Credits: Chris Manning

# TRAINING A RNN LANGUAGE MODEL

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

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

- In practice, consider the sequence of tokens $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$
- Stochastic Gradient Descent allows us to compute and update the loss and gradients for small chunk of data.
  - Compute loss $J(\theta)$ for a sentence (a batch of sentences)
  - Compute gradients and update weights. Repeat.

# EVALUATING LANGUAGE MODELS

- The standard evaluation metric for Language Models is perplexity.

$$\underbrace{\text{perplexity} = \prod_{t=1}^{T} \left( \frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)}_{\text{Inverse probability of corpus, according to Language Model}}^{1/T}$$

Normalized by number of words

- This is equal to the exponential of the cross-entropy loss $J(\theta)$ :

$$= \prod_{t=1}^{T} \left( \frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left( \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$



$\boxed{\textbf{Lower}\text{ perplexity is better!}}$

# LIMITATIONS OF 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

# LIMITATIONS OF RNN LANGUAGE MODEL

Apply nonlinear activation function $\sigma$ on $h^{(t-1)}$ and $x^{(t)}$ to estimate $h^{(t)}$ can lead to numerous limitations especially on very long sequence:

- Exploding gradients (e.g. when $\sigma$ = *ReLU*)
  - When the gradient becomes too large
  - Model takes large steps and might not find optimal solution

- Vanishing gradients (e.g. when $\sigma$ = *tanh*):
  - when the gradient turns to 0

# WHY IS EXPLODING GRADIENT A PROBLEM?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_{\theta} J(\theta)}_{\text{gradient}}$$

- This can cause bad updates: we take too large a step and reach a bad parameter configuration (with large loss)
- In the worst case, this will result in Inf or NaN in your network (then you have to restart training from an earlier checkpoint)

# SOLUTION TO EXPLODING GRADIENT

Gradient clipping

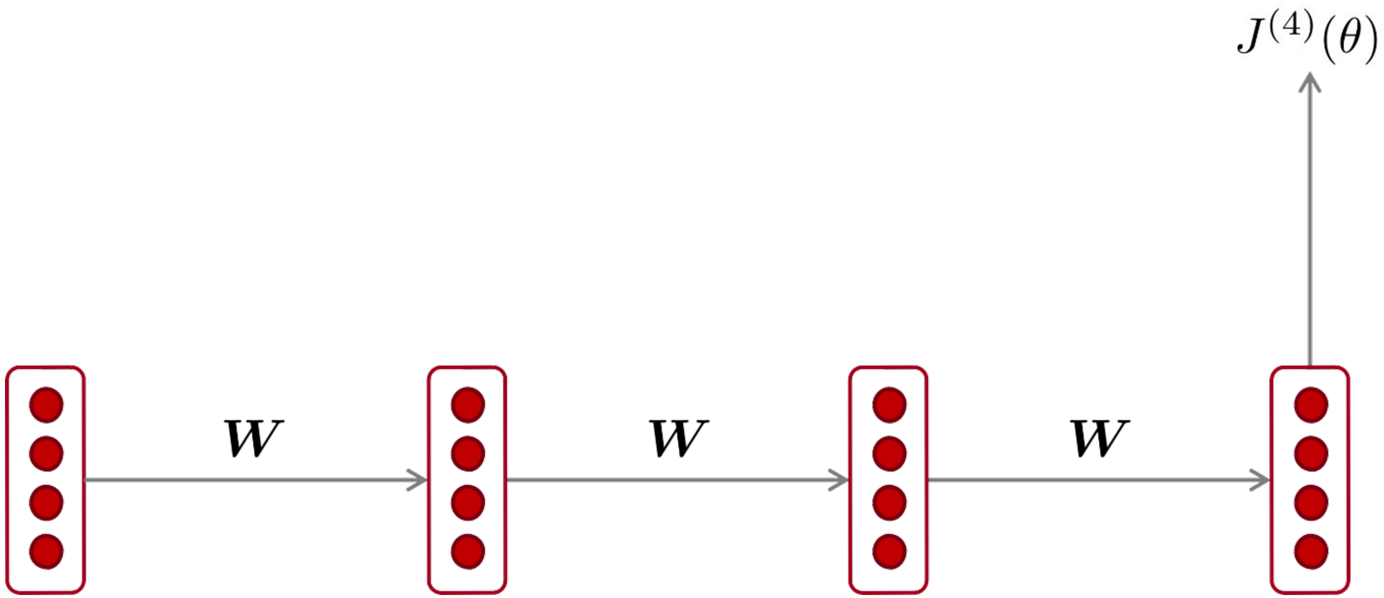
- define gradient threshold
- clip all gradients greater than the threshold

**Algorithm 1** Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq \mathit{threshold}$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{\mathit{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
**end if**

This allow the RNN model to take smaller steps in the same direction.

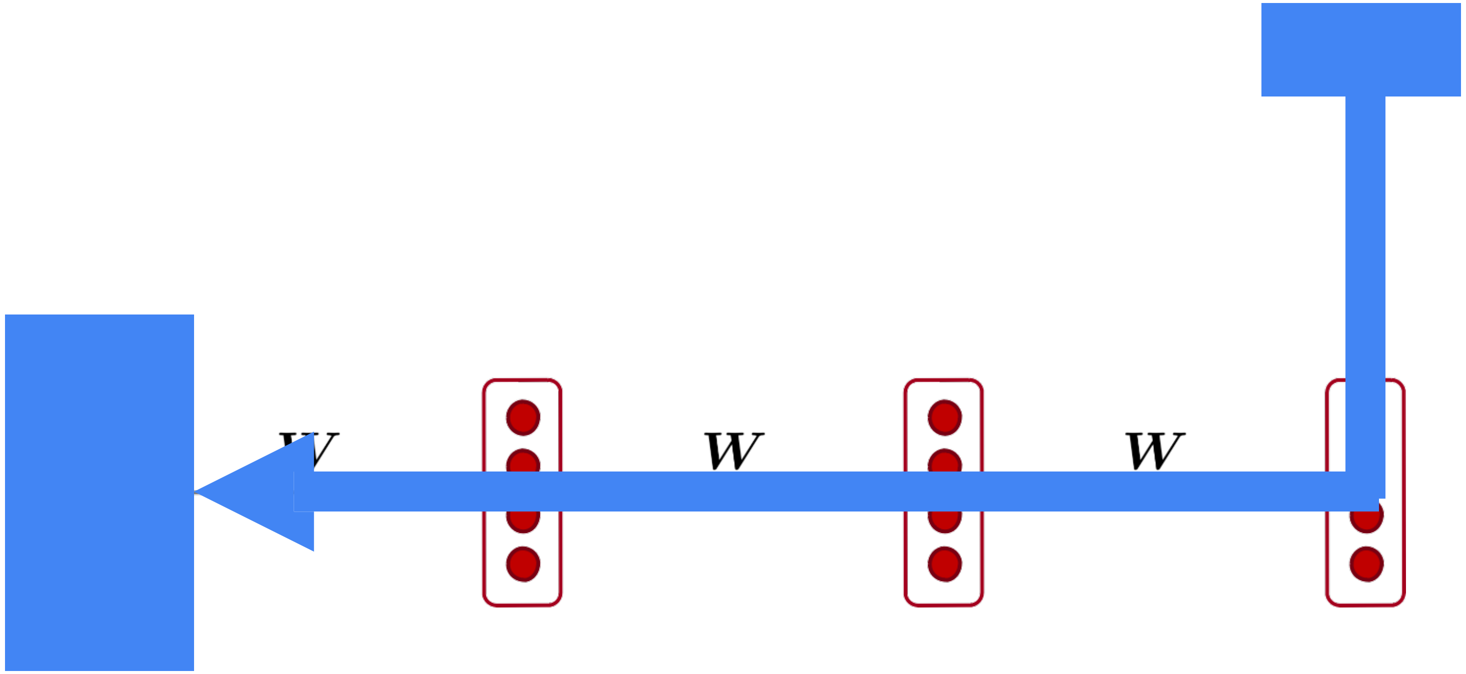
# VANISHING GRADIENT






*Credits: Chris Manning*

# VANISHING GRADIENT

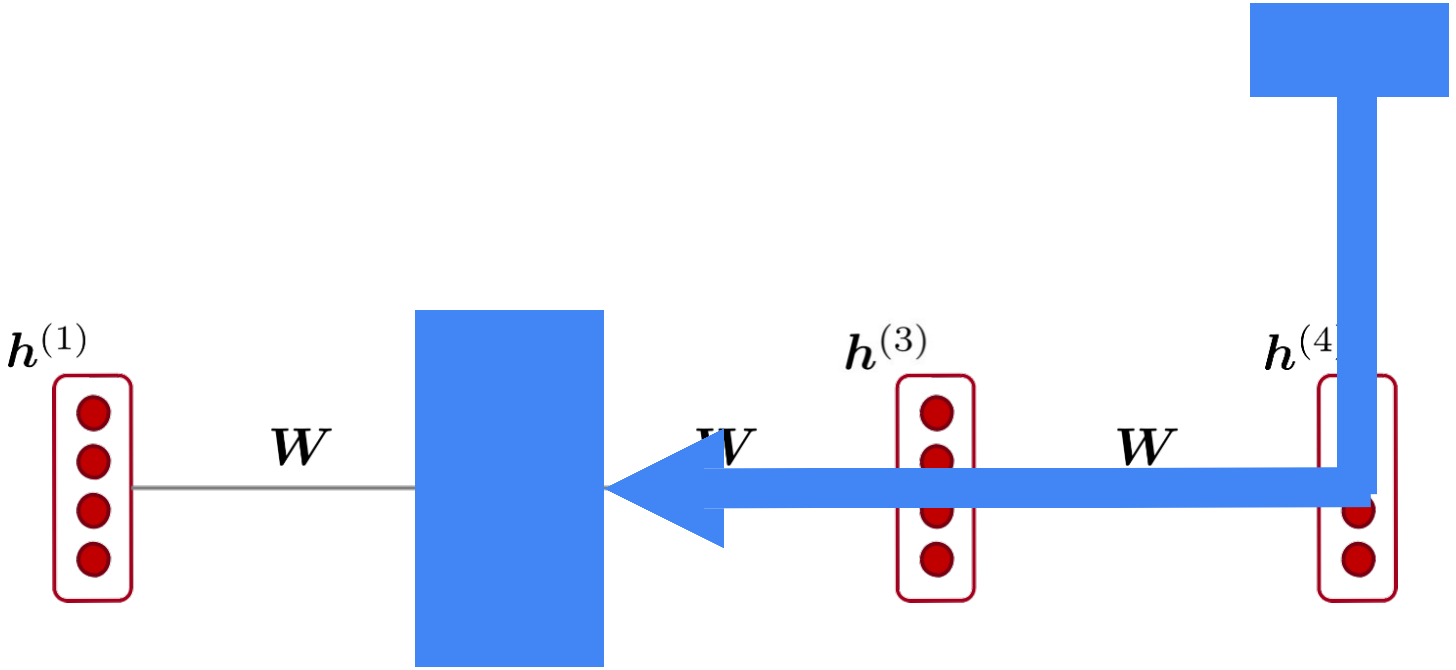



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = ?$$



*Credits: Chris Manning*

# VANISHING GRADIENT

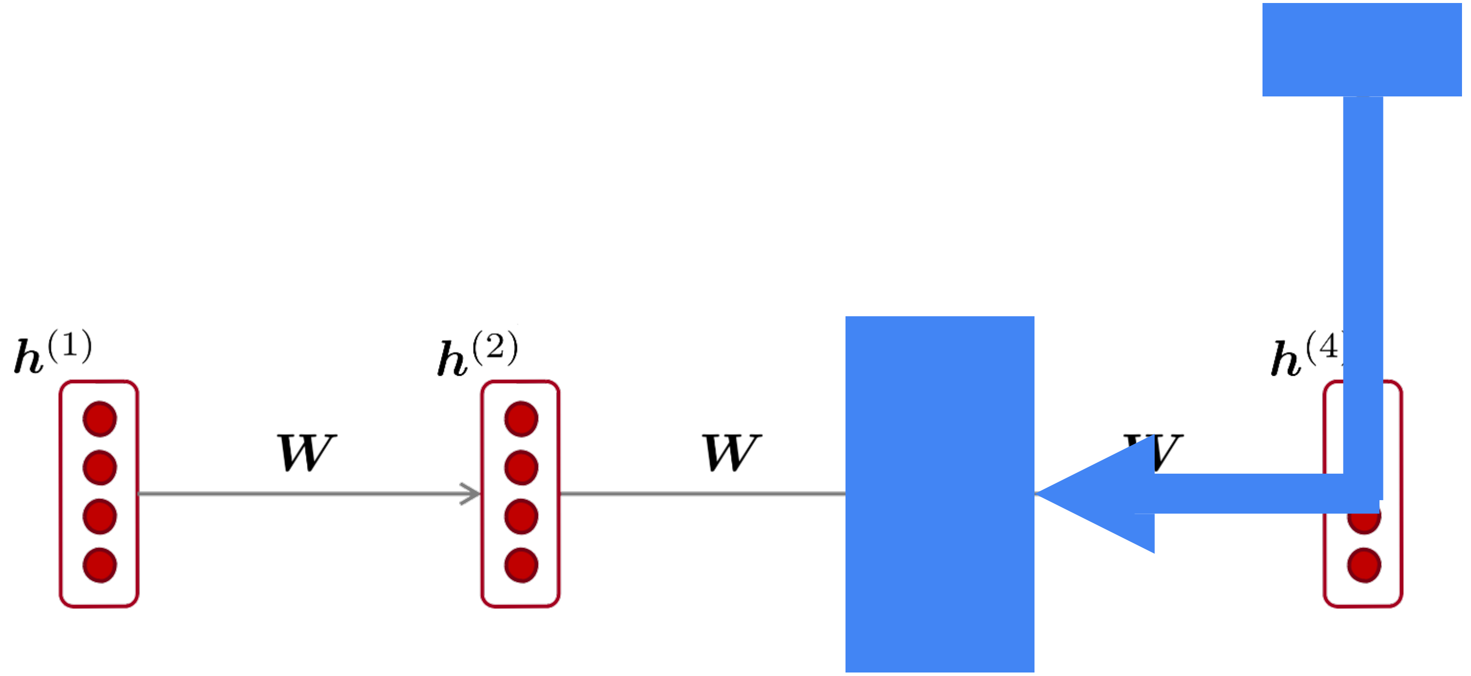



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(2)}}$$

chain rule!

Credits: Chris Manning

# VANISHING GRADIENT INTUITION



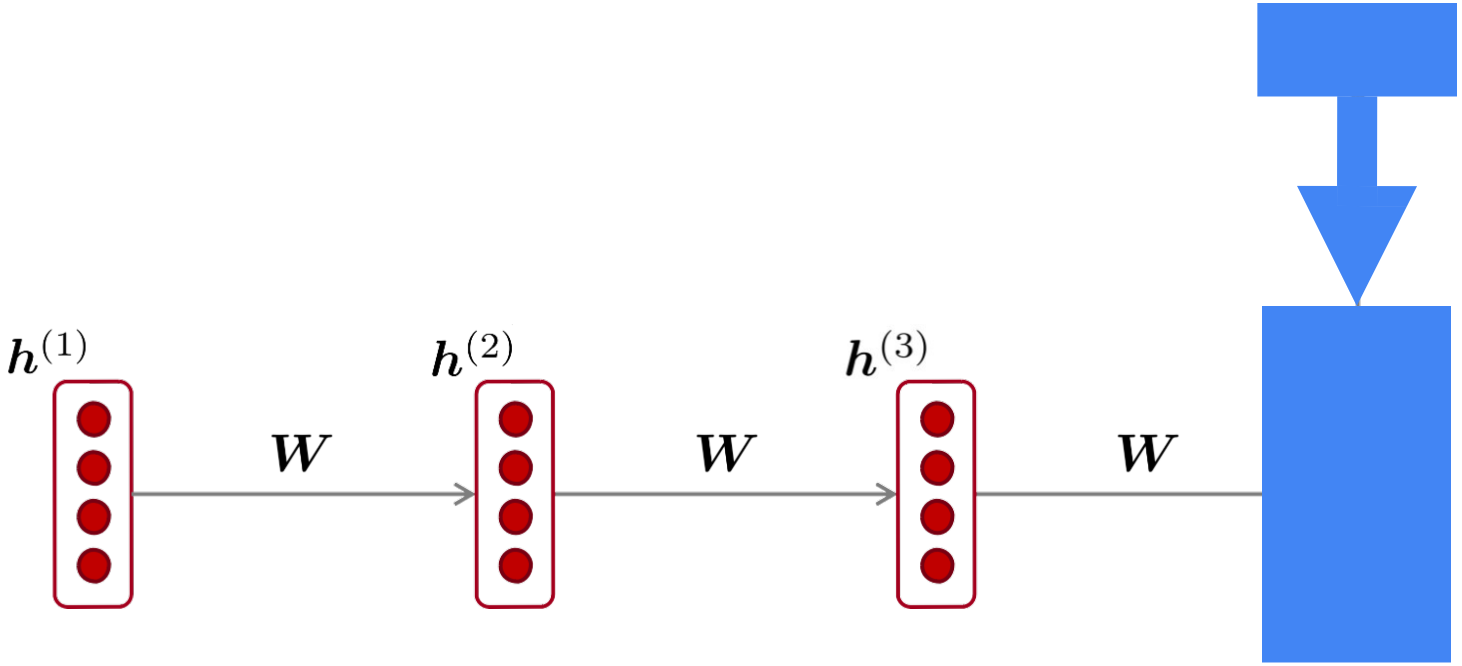

$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \qquad \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(3)}}$$



chain rule!

*Credits: Chris Manning*

# VANISHING GRADIENT INTUITION



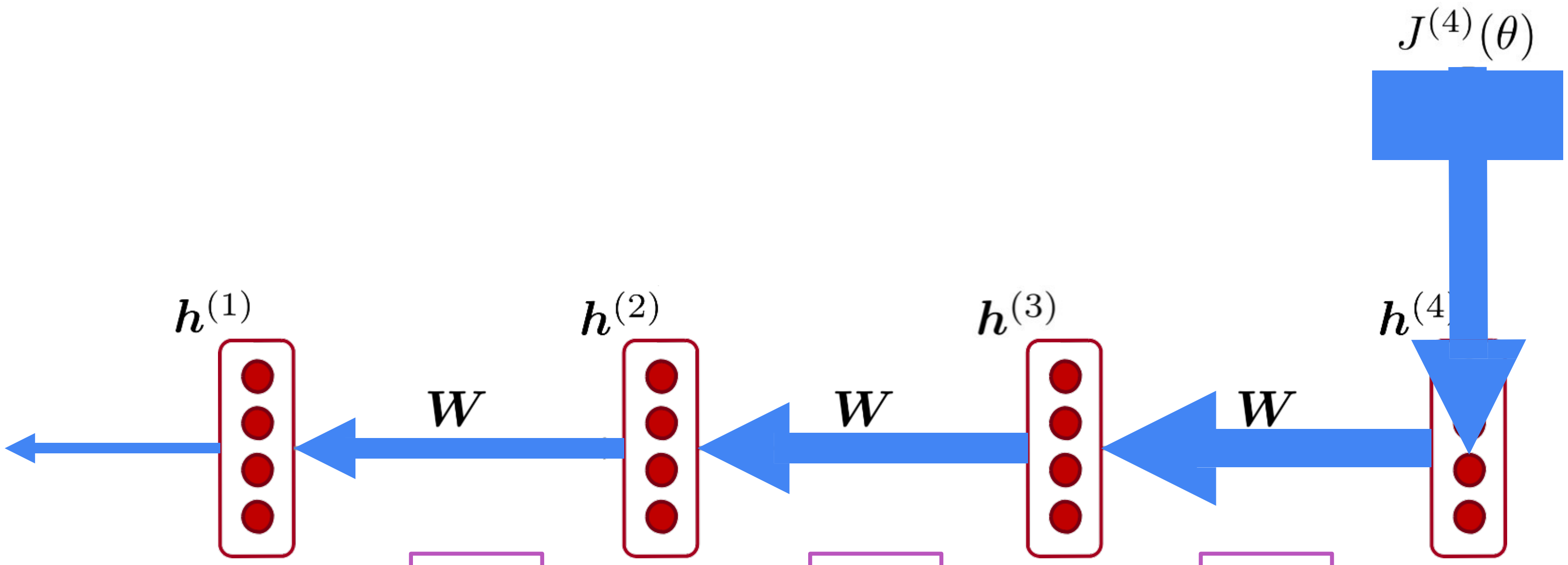

$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$



chain rule!

*Credits: Chris Manning*

# VANISHING GRADIENT INTUITION




$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \boxed{\frac{\partial h^{(2)}}{\partial h^{(1)}}} \times \boxed{\frac{\partial h^{(3)}}{\partial h^{(2)}}} \times \boxed{\frac{\partial h^{(4)}}{\partial h^{(3)}}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

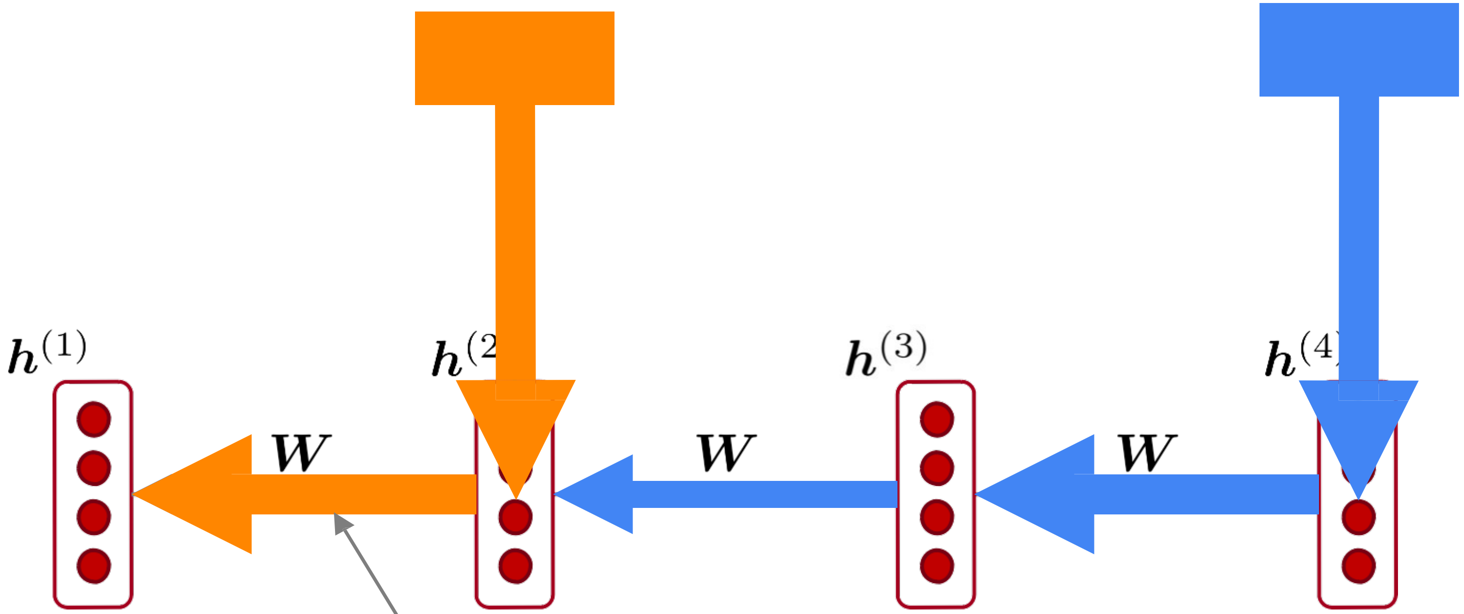
What happens if these are small?

**Vanishing gradient problem:** When these are small, the gradient signal gets smaller and smaller as itbackpropagates further



*Credits: Chris Manning*

# WHY IS VANISHING GRADIENT A PROBLEM?




Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated only with respect to near effects, not long-term effects.



Credits: Chris Manning

# WHY IS VANISHING GRADIENT A PROBLEM?

- Another explanation:
  - Gradient can be viewed as a measure of *the effect of the past on the future*

- If the gradient becomes vanishingly small over longer distances (step *t* to *t+n*), then we can't tell whether:
  - There's no dependency between step *t* and *t+n* in the data
  - We have wrong parameters to capture the true dependency between *t* and *t+n*

# EFFECT OF VANISHING GRADIENT ON RNN-LM

**LM task:**
*When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* ________________________________________

- To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.

- But if gradient is small, the model can't learn this dependency
  - the model is unable to predict similar long-distance dependencies at test time

# EFFECT OF VANISHING GRADIENT ON RNN-LM

*is*

*are*

- **LM task:** *The writer of the books* __________

- **Correct answer**: *The writer of the books is planning a sequel*

- **Syntactic recency:** *The writer of the books is* (correct)

- **Sequential recency:** *The writer of the books are* (incorrect)

- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]



"Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies", Linzen et al, 2016. https://arxiv.org/pdf/1611.01368.pdf

# HOW TO FIX VANISHING GRADIENT PROBLEM?

- The main problem is that *it's too difficult for the RNN to learn to*
  - *preserve information over many timesteps*.

- In a RNN, the hidden state is constantly being rewritten at each time step

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

- Two architectures have been designed to allow RNN to have memory
  - LSTM
  - GRU

# 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 state $c^{(t)}$ and timestep $t$

**Forget gate:** controls what is kept vs forgotten, from previous cell state

**Input gate:** controls what parts of the new cell content are written to cell

**Output gate:** controls what parts of cell are output to hidden state

**Sigmoid function**: all gate values are between 0 and 1

$$\boldsymbol{f}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

$$\boldsymbol{i}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

$$\boldsymbol{o}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

**New cell content:** this is the new content to be written to the cell

**Cell state**: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

**Hidden state**: read (“output”) some content from the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

All these are vectors of same length *n*

Gates are applied using element-wise product

*Credits: Chris Manning*

# HOW DOES LSTM SOLVE VANISHING GRADIENTS?

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

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

# GATED RECURRENT UNITS (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep $t$ we have input $x^{(t)}$ and hidden state $h^{(t)}$ (no cell state).

**Update gate:** controls what parts of hidden state are updated vs preserved

**Reset gate:** controls what parts of previous hidden state are used to compute new content

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

**New hidden state content:** reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

**Hidden state:** update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h(\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

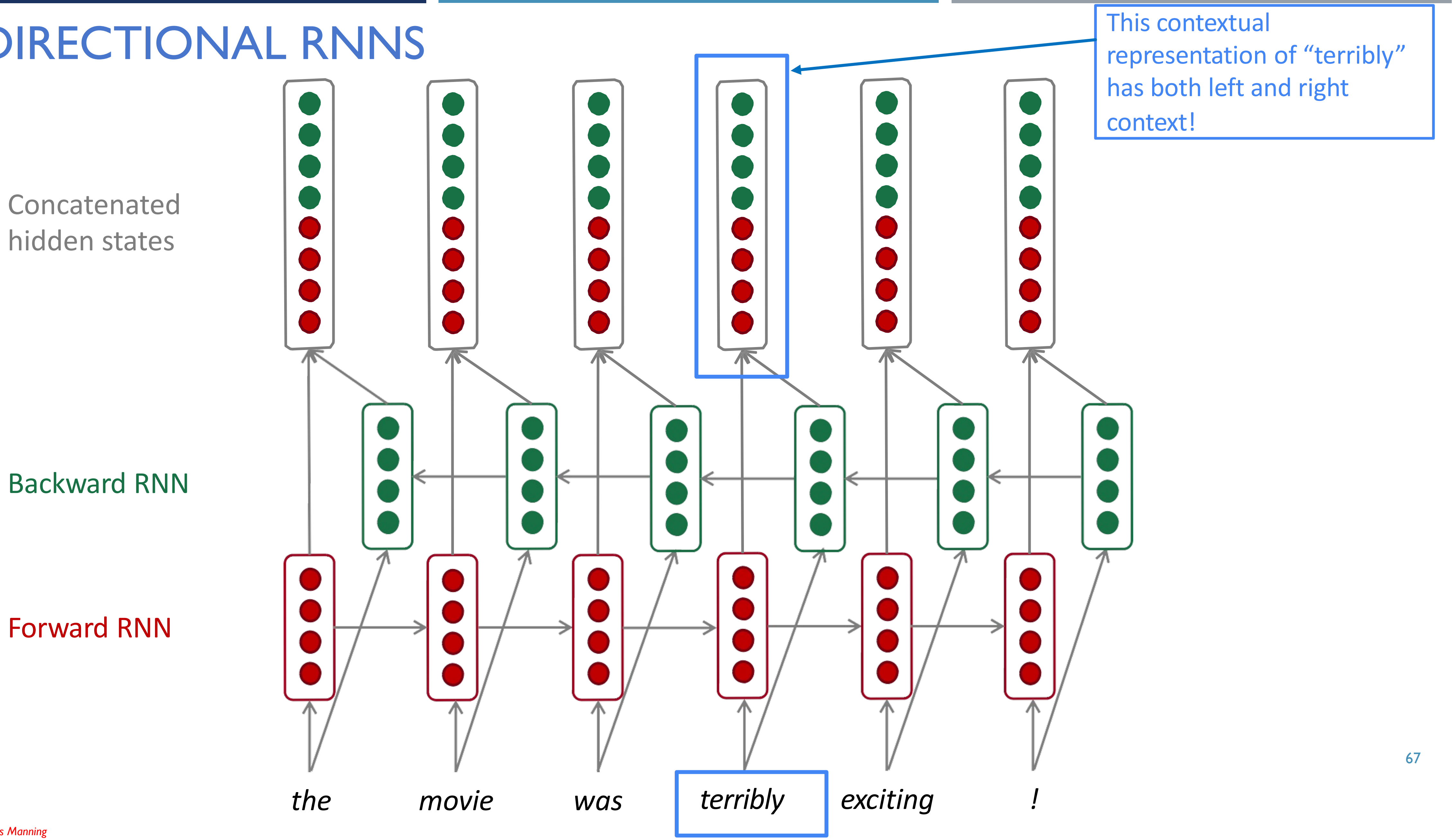
**How does this solve vanishing gradient?** Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)



Credits: Chris Manning

"Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", Cho et al. 2014, https://arxiv.org/pdf/1406.1078v3.pdf

# BIDIRECTIONAL RNNS






Credits: Chris Manning

# BIDIRECTIONAL RNNS

On timestep *t*:

This is a general notation to mean "compute one forward step of the RNN" – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \boxed{\mathrm{RNN}_{\mathrm{FW}}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boxed{\boldsymbol{h}^{(t)}} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.



Credits: Chris Manning

# REFERENCES


- Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies”, Linzen et al, 2016
- Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", Cho et al. 2014
- https://web.stanford.edu/~jurafsky/slp3/
- Foundations of Statistical Natural Language Processing by Christopher D. Manning, Hinrich Schütze