Lecture 06:
Neural language modelling

Overview

- Neural language model
    - Recurrent neural networks
    - LSTM
    - GRU

Recap…

- Given a sequence of words $w^{(1)}, w^{(2)}, \dots, w^{(t)}$, a language model compute the probability distribution of the next word $w^{(t+1)}$

$$P\left(w^{(t+1)} \middle| w^{(t)}, \dots, w^{(1)}\right), \qquad \forall w \in V = \{w^1, \dots, w^{|V|}\}$$

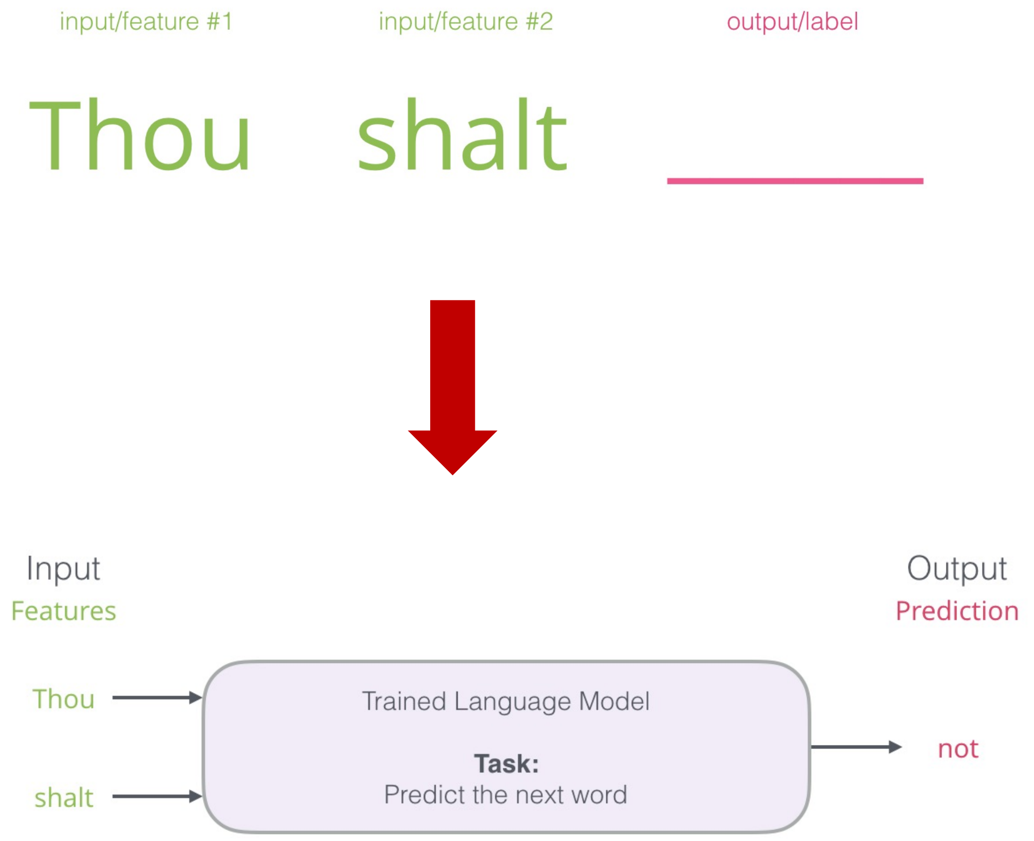
Markov assumptions: $w^{(t+1)}$ depends only on the preceding *n-1* words.

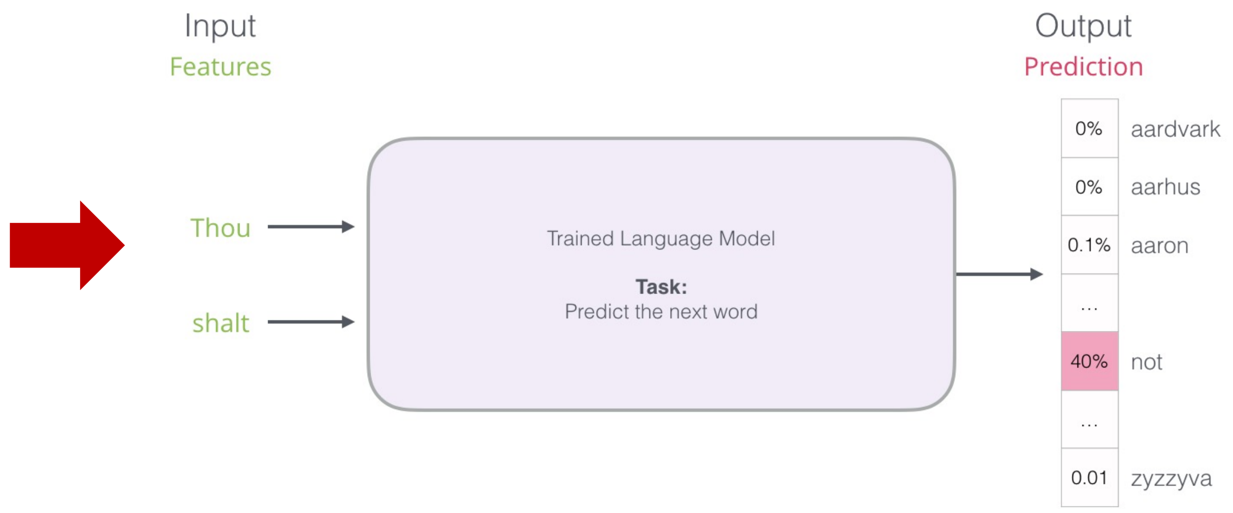
$$P(w^{t+1}|w^t, \dots, w^1) \approx P(w^{t+1}|w^t, \dots, w^{t-n+2})$$

- Model that assigns a prbobality to the sequence of words

$$P(w^{(1)}, w^{(2)}, \dots, w^{(t)})$$

Recap…

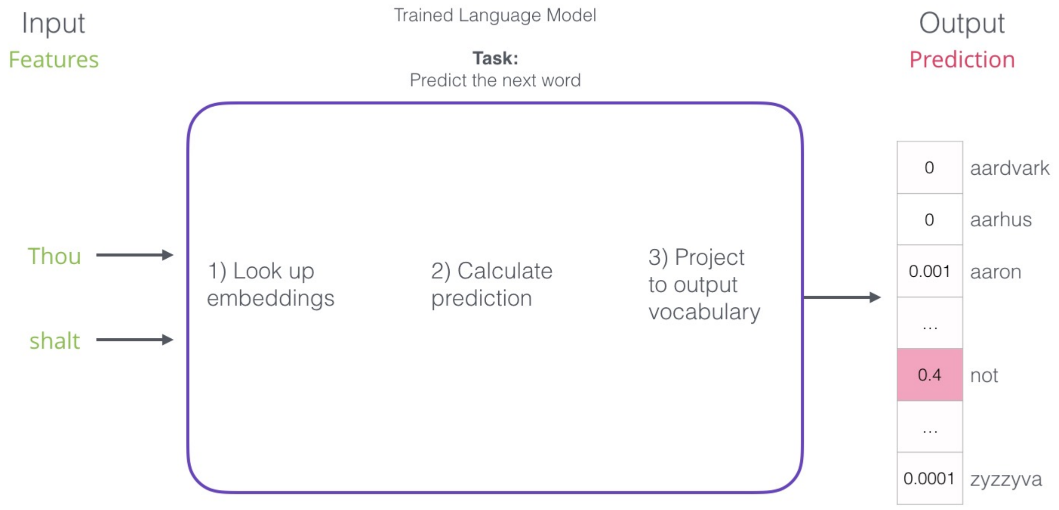


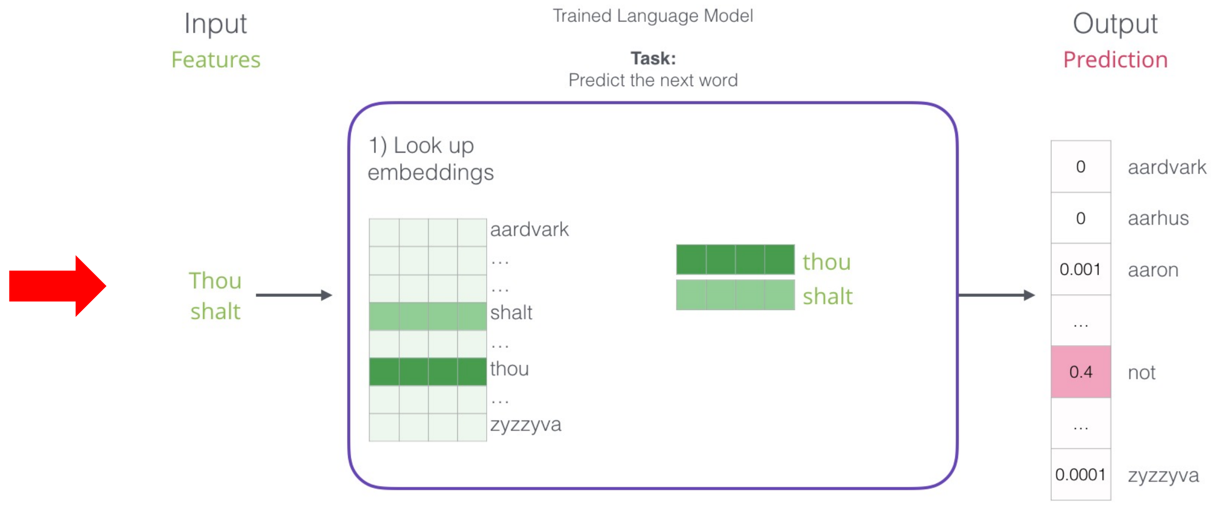


0%	aardvark
0%	aarhus
0.1%	aaron
…	
40%	not
…	
0.01	zyzzyva

Recap...

Three main steps in a neural language models





Neural language model

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

- Neural netword based language model
  - Fixed window-based
    - classify a word in its context window of neighbouring words.

  - RNN based

neural language model

output distribution

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

hidden layer

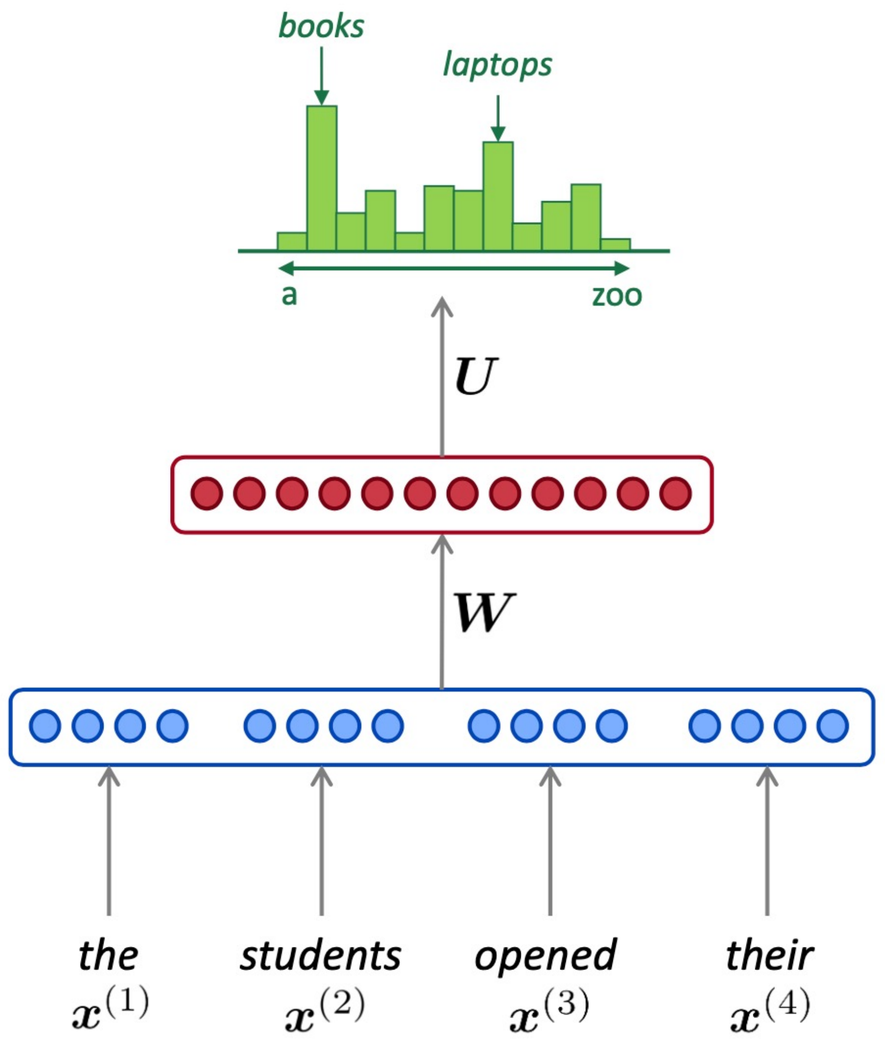
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



Credit: Chris manning

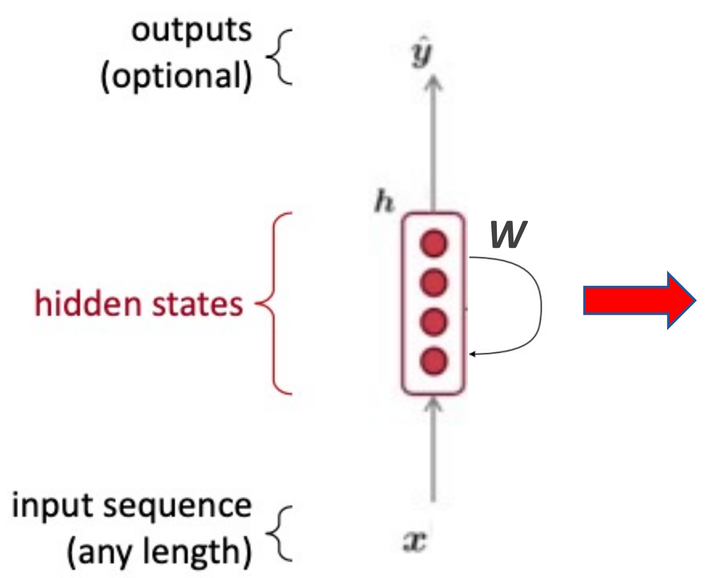
neural language model

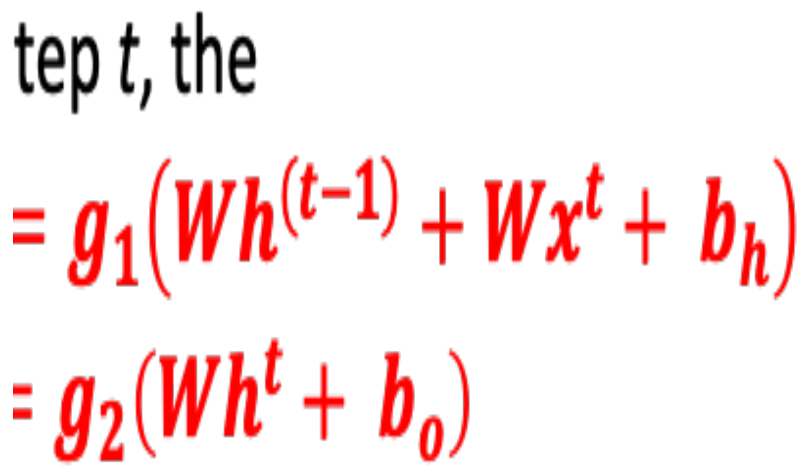
- Fixed-window improvement over n-gram language model:
  - Do not suffer from the sparsity problem
  - Don’t need to store all observed n-grams

- Unresolved problems
  - Fixed window maybe to small to capture appropriate context
  - Enlarging window enlarges the ***W*** matrix
  - Windows might not be large enough
    - Every window size might leave out relevant context information
  - There is no symetry in how the inputs are processed
    - $w^{(1)}$ and $w^{(2)}$ are multiplied by different weigths

Recurrent neural network (RNN)

- RNN architecture: Neural Network that applies the same weight matrix ***W*** repeatedly
  - are a class of neural networks that allow previous outputs to be used as inputs while having hidden states.




tep t, the

$= g_1\left(Wh^{(t-1)} + Wx^t + b_h\right)$

$= g_2\left(Wh^t + b_o\right)$

For each timestep t, the

activation: $\boldsymbol{h^t = g_1\left(Wh^{(t-1)} + Wx^t + b_h\right)}$

output: $\boldsymbol{y^t = g_2(Wh^t + b_o)}$

RNNs for Language model

$\hat{y}^5 = P\left(x^{(5)} \middle| \textit{the students opened their}\right)$

output distribution

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

hidden states

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

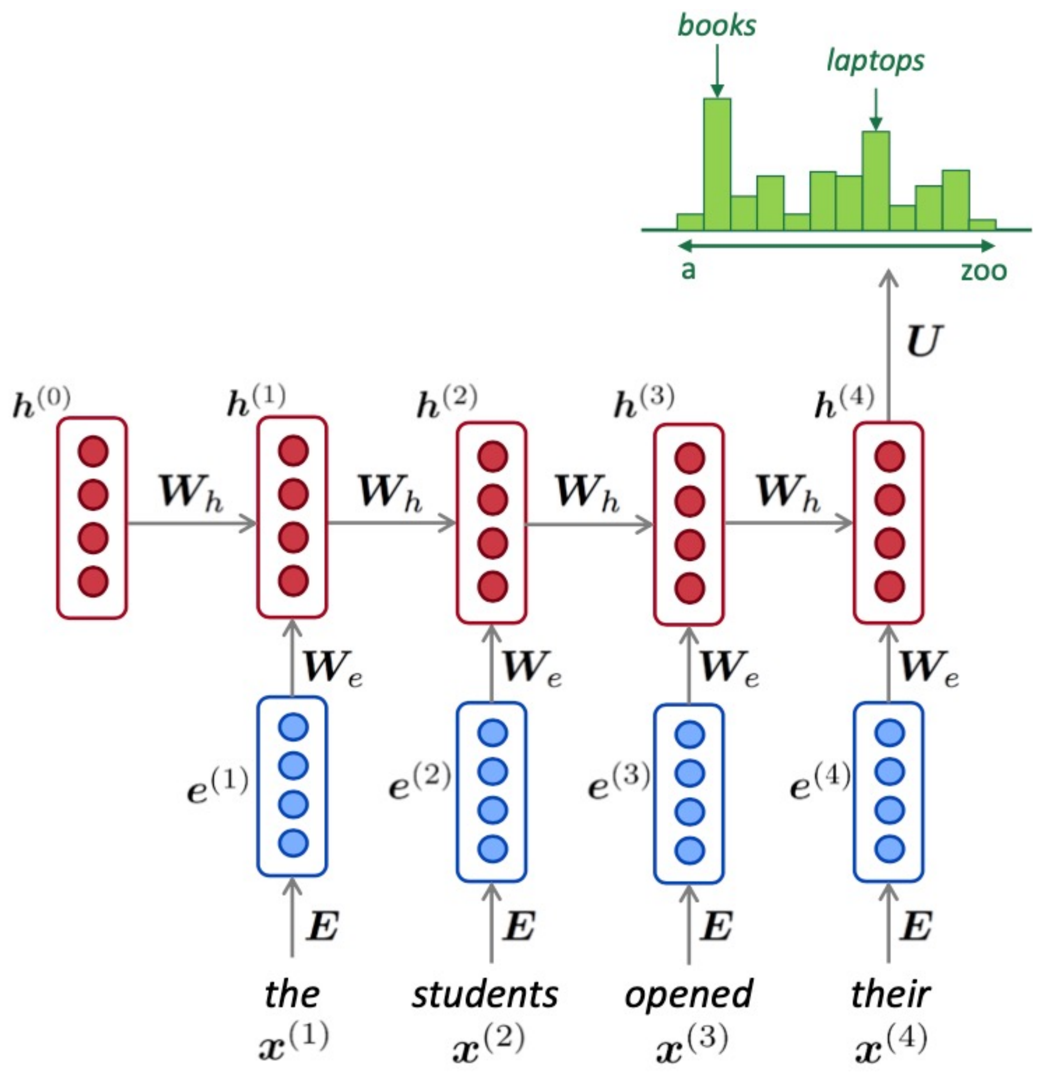
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



Credit: Chris Manning

Advantages of RNNs

RNN advantages:

- Can process any length input
- Computation for step *t* can use information from many steps back
- Model size doesn’t increase with input sentence
- There is symmetry in how inputs are processed
  - The same weight matrix is applied on every time step (word in the sequence)

RNN disadvantages:

- Training is computationally intensive
- It is difficult to access information from many steps back

Training a RNN Language model

- Feed the sequence $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ into and RNN
- Estimate probability distribution of *every word* $\hat{y}^{(t)}$ in the vocabulary
  - Predict probability of every word given the history

- *Coss-entropy loss* between predicted probabiliyty distribution $\hat{y}^{(t)}$ and the true next word $y^{(t)}$ (one hot vector for $x^{(t+1)}$).

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

- Overall loss of the entire training set is the average of the individaul loss function.

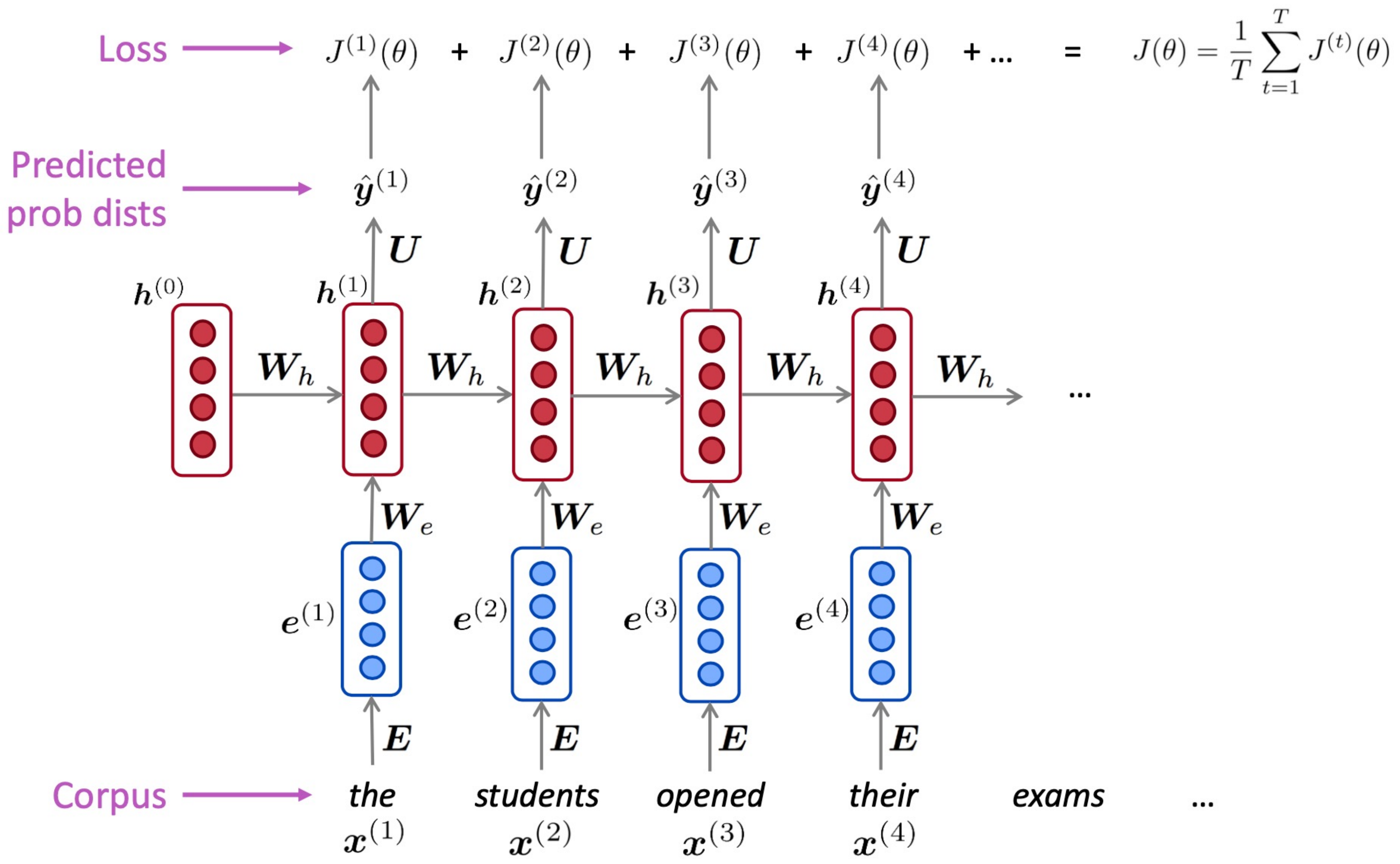
$$J(\theta) = \frac{1}{T}\sum\nolimits_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -log\hat{y}_{x_{t+1}}^{(t)}$$

Training RNNs Language model

- RNN is computationally intensive
  - computing loss and gradient across entire corpus $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ is expensive.
- Stochastic gradient descend allows you to compute gradient and update weights for small chunks of data.
  - Compute loss $J(\theta)$ for a batch of sentences
- Find the parameters θ^* that assigns the maximum probability $x^{(1)}, \ldots, x^{(T)}$

$$\theta^* = \underset{\theta}{\operatorname{argmax}} P_\theta(x^{(1)}, \ldots, x^{(T)}) = \underset{\theta}{\operatorname{argmax}} \prod_{t=1}^{T} P_\theta(x^{(t)} | x^{(1)} \ldots, x^{(t-1)})$$

Training a RNN Language model



Loss
$J^{(1)}(\theta)$ + $J^{(2)}(\theta)$ + $J^{(3)}(\theta)$ + $J^{(4)}(\theta)$ + … = $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$
Predicted prob dists
$\hat{\boldsymbol{y}}^{(1)}$ $\hat{\boldsymbol{y}}^{(2)}$ $\hat{\boldsymbol{y}}^{(3)}$ $\hat{\boldsymbol{y}}^{(4)}$
$\boldsymbol{U}$
$\boldsymbol{h}^{(0)}$ $\boldsymbol{h}^{(1)}$ $\boldsymbol{h}^{(2)}$ $\boldsymbol{h}^{(3)}$ $\boldsymbol{h}^{(4)}$
$\boldsymbol{W}_h$
…
$\boldsymbol{W}_e$
$\boldsymbol{e}^{(1)}$ $\boldsymbol{e}^{(2)}$ $\boldsymbol{e}^{(3)}$ $\boldsymbol{e}^{(4)}$
$\boldsymbol{E}$
Corpus
the students opened their exams …
$\boldsymbol{x}^{(1)}$ $\boldsymbol{x}^{(2)}$ $\boldsymbol{x}^{(3)}$ $\boldsymbol{x}^{(4)}$

Credit: Chris Manning

Limitations of RNNs

- Apply nonlinear activation function g on $h^{(t-1)}$ and $x^{(t)}$ to estimate $h^{(t)}$

$$h^{(t)} = g(Uh^{(t-1)} + Wx^{(t)} + \ b_h)$$

  - where $g \in \{tanh, ReLU\}$

- Difficulties training on effectively on long sequences.
  - *g = tanh:*
    - RNNs often suffer from vanishing gradient
  - *g = ReLU:*
    - RNNs often suffer from exploding gradient

Exploding gradient problem

- When the gradient becomes too large then the SGD update step becomes too big:

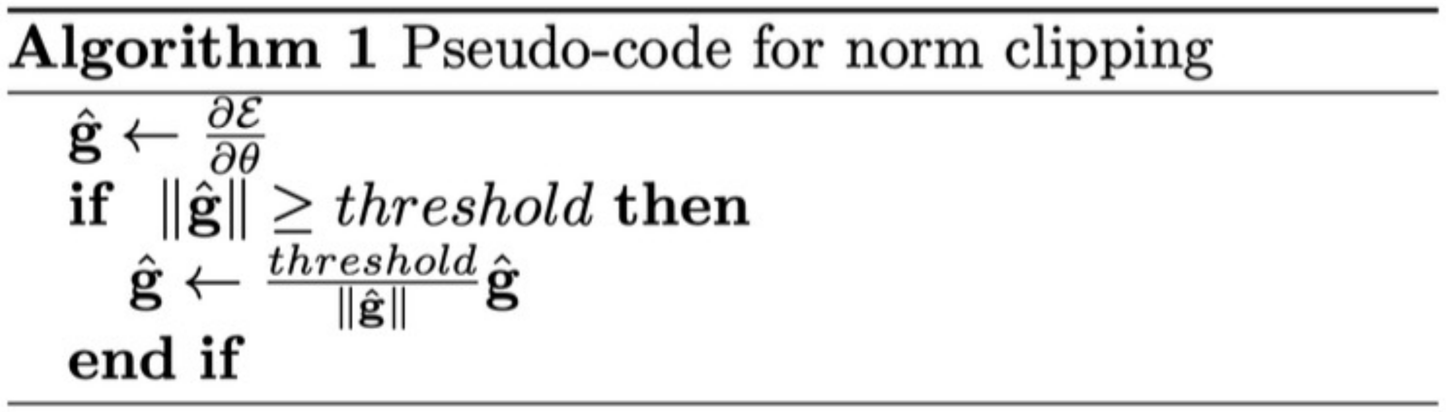
$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$

α is the learning rate

- Exploding gradients can result to bad updates:
  - if we take too large a step (i.e., large α) and reach a bad parameter configuration (with large loss)
  - Often as a result of not selecting an appropriate α
- Exploding gradients can result to *inf* and *NaN* in your network

Solution for exploding gradient

- **Gradient clipping:**
  - Define a gradient threshold
  - If current gradient is greater than threshold scale it down before applying stochastic gradient descent

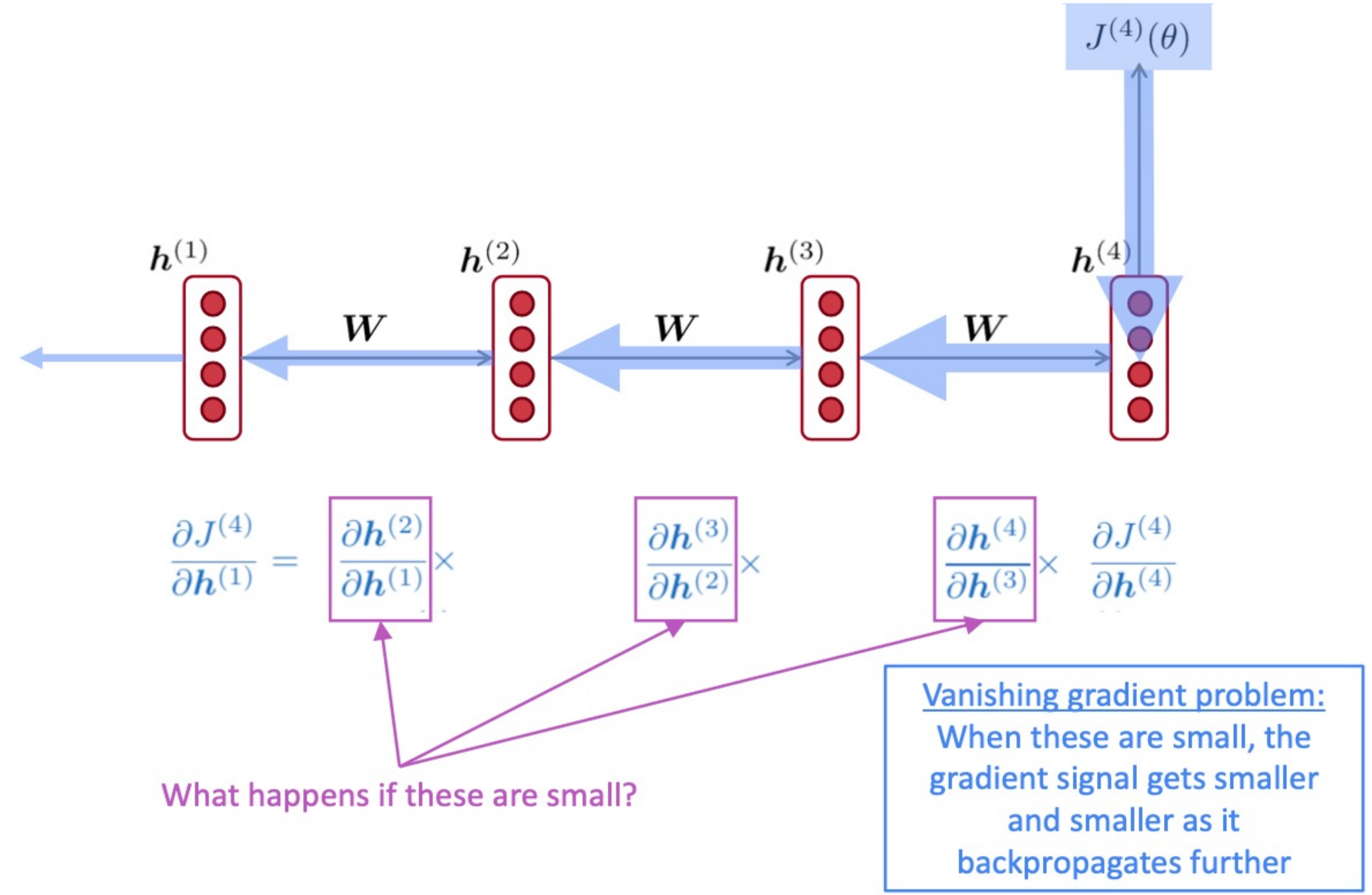


Algorithm 1 Pseudo-code for norm clipping

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

- **Intuition:** this allows the model to take a smaller step in the same direction

Vanishing gradient problem



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

What happens if these are small?

<u>Vanishing gradient problem:</u>
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Credit: Chris Manning

Vanishing gradient problem

- Gradient signal from faraway is lost becuase it’s much smaller than the gradient signal from close by.

- Model weights are updated only with respect to near effects not long-term effects.

- Gradients 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 step $t + n$, then we can’t tell whether
    1. There is no dependency between step t and $t + n$ in the data
    2. We have wring parameters to capture the true dependency between t and $t + n$

Effect of vanishing gradient on RNN-LM

Consider the language model task

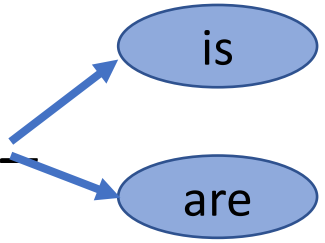
When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her ________

*Predicted word: **ticket***
*Correct word: **tickets***

- 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
  - So the model is unable to predict similar long-distance dependencies at test time

Effect of vanishing gradient on RNN-LM

- **Sentence:** *The writer of the books is planning a sequel.*
- **LM task:** *The writer of the books___*



- ***Correct*** ⇔ ***Syntactic recency****: The <u>writer</u> of the books <u>is</u>*
- ***Incorrect*** ⇔ ***Sequential recency****: The writer of the <u>books</u> <u>are</u>*

- Due to vanishing gradient, RNN-LM are better at learning from sequential recency than syntactic recency.

Solution for vanishing/exploding gradient

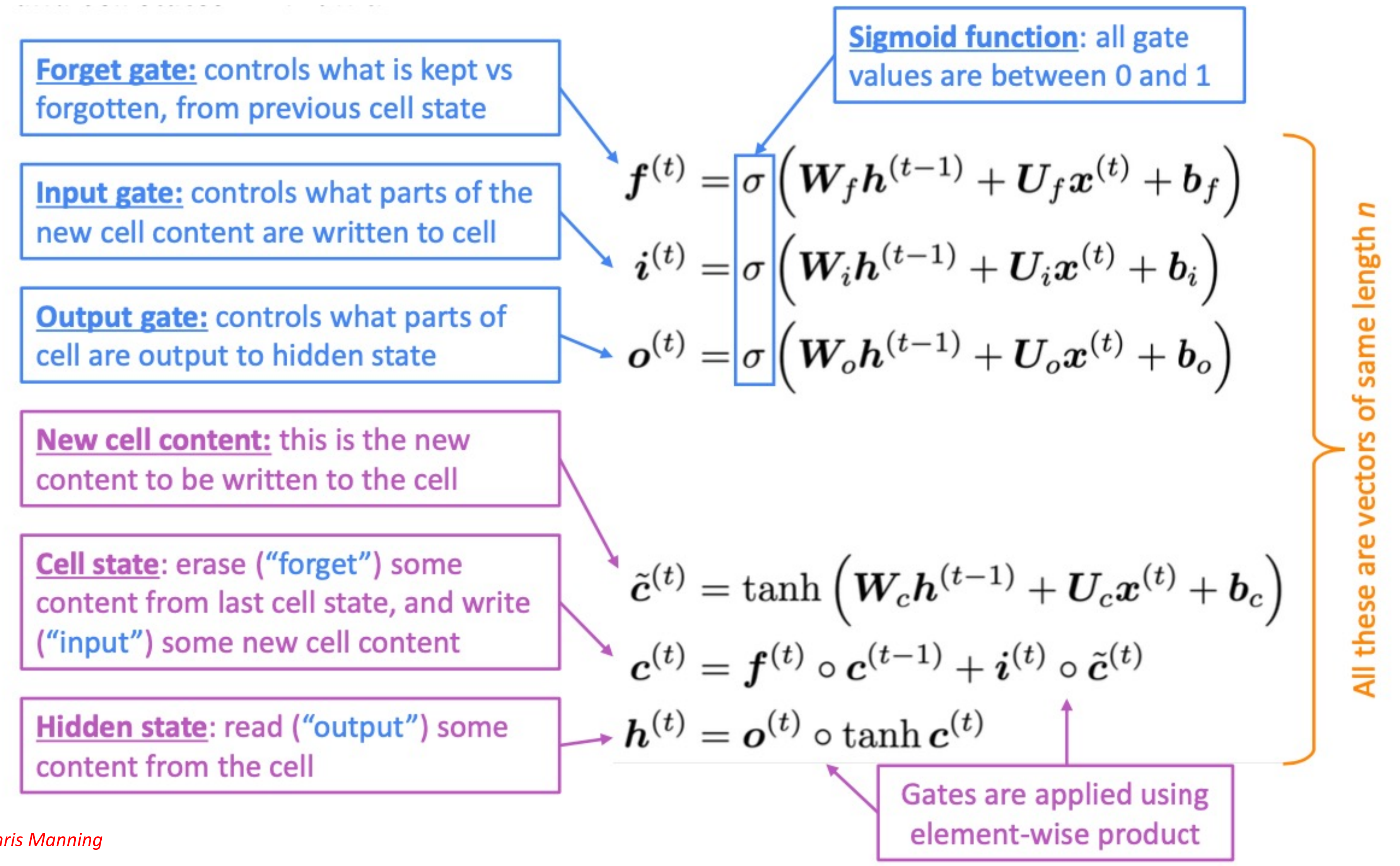
- Traditional RNN the hidden state is constantly being rewritten

$$h^t = g_1\left(W^h h^{(t-1)} + W^x x^t + \ b_h\right)$$

  - Constant updating hidden states may lead to lose of relevant information

- How about a RNN with separate memory?
  - Two RNN flavours to handle vanishing gradient problem
    - Long Short-term Memory (LSTM)
    - Gated Recurrent Units (GRU)

Long Short-Term Memory (LSTM)



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

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

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

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

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

Credit: Chris Manning

How does LSTM solve the vanishing gradient?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - if the forget gate is set to remember everything on every timestep, then the info 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 the 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

Gated Recurrent Units (GRU)

- On each timestep t has input $x^{(t)}$ and hidden state $h^{(t)}$ but no cell state.

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

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

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

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

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

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

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

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

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)

Credit: Chris Manning

Filling the gap

- Consider the following sentences:
  - I am ___.
  - I am ___ hungry.
  - I am ___ hungry, and I can eat half a pig.

- Depending on the amount of information we can fill the blanks with very different words such as *happy, not* and *very*

- A language model that is incapable of taking advantage of all information will perform poorly on such task.

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.

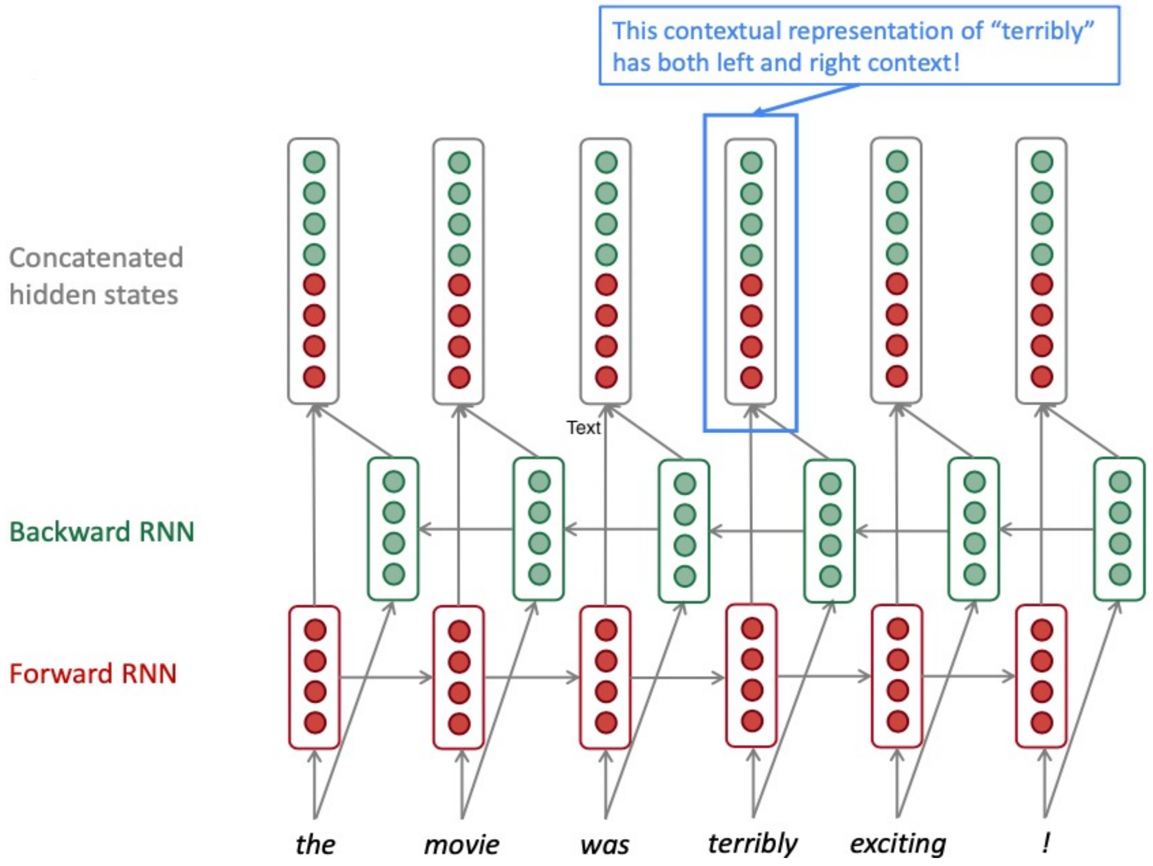
$$\left.\begin{aligned}
&\text{Forward RNN} && \overrightarrow{\boldsymbol{h}}^{(t)} = \boxed{\mathrm{RNN}_{\mathrm{FW}}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)}) \\
&\text{Backward RNN} && \overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})
\end{aligned}\right\} \text{Generally, these two RNNs have separate weights}$$

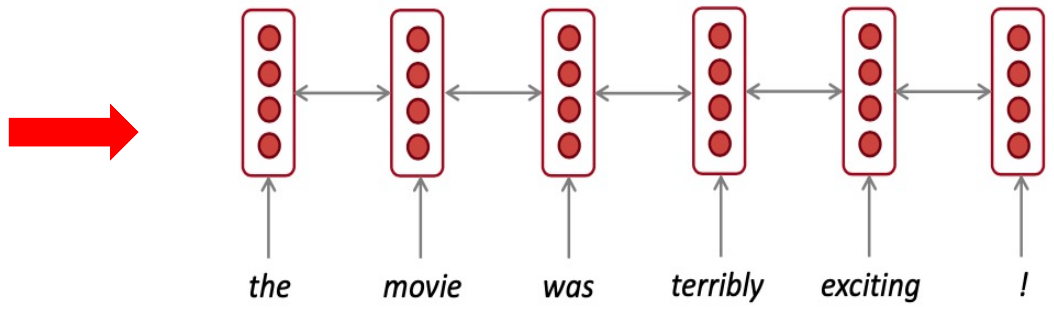
$$\text{Concatenated hidden states} \quad \boxed{\boldsymbol{h}^{(t)}} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$$

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

Credit: Chris Manning

Bidirectional RNNs


This contextual representation of "terribly" has both left and right context!
Concatenated hidden states
Text
Backward RNN
Forward RNN
the
movie
was
terribly
exciting
!



Credit: Chris Manning

Bidirectional RNNs

- Applicable if you have access to the entire input sequence.
  - **Not** applicable to Language Modelling, because in LM you only have left context available.
  - bidirectionality is more efficient if you have entire sequence

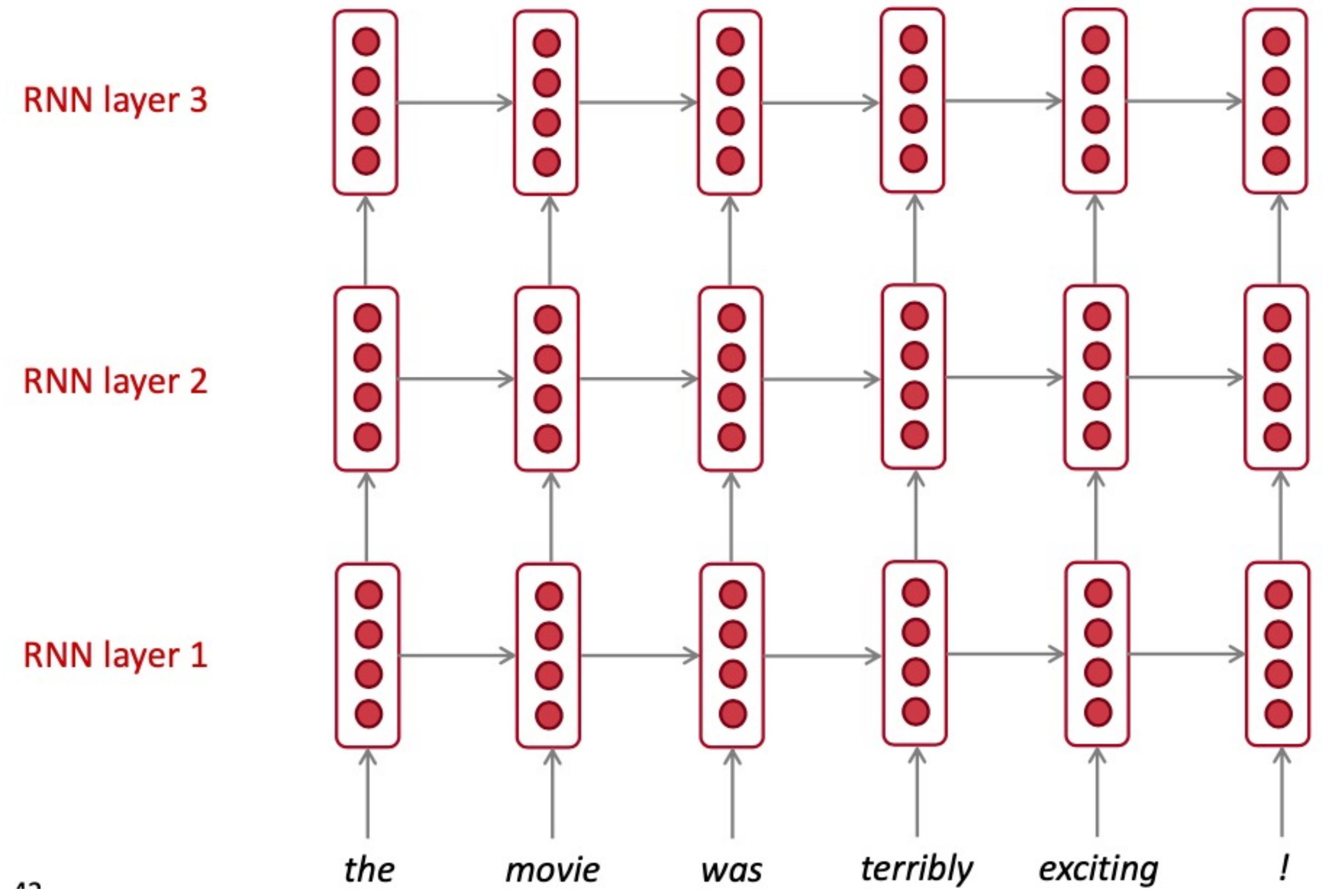
- Examples:
  - BERT (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

Multi-layer or Stacked RNNs

- Bidirectional RNNs are a two layer RNN stacked together in reverse direction.
- When unroll over many timesteps RNNs are deep in one direction
- Stacked RNNs are deep in two dimension (horizontally and vertically).
    - i.e. stacking an RNN layers vertically
    - The hidden state is a concatenation of all hidden states from each stacked RNNs
- Stacking RNN layers allows the network to compute more complex representations
    - The lower RNNs should compute lower-level features and
    - the higher RNNs should compute higher-level features.

Multi-layer or Stacked RNNs

- Hidden states of layer *i* are the inputs to RNN layer i+1



Assingment:

- Study the LSTM and GRU architecture (deep learning course notes) and compare and contrast their applications in NLP
  - Identify the advantages and disadvantages of LSTM and GRU
  - Can you determine when LSTM is preferable over GRU and vice versa