Deep Learning for Natural Language Processing

Cornelia Caragea

Computer Science
University of Illinois at Chicago

Credits for slides: Manning, Socher, See

Recurrent Neural Networks Variants

Today

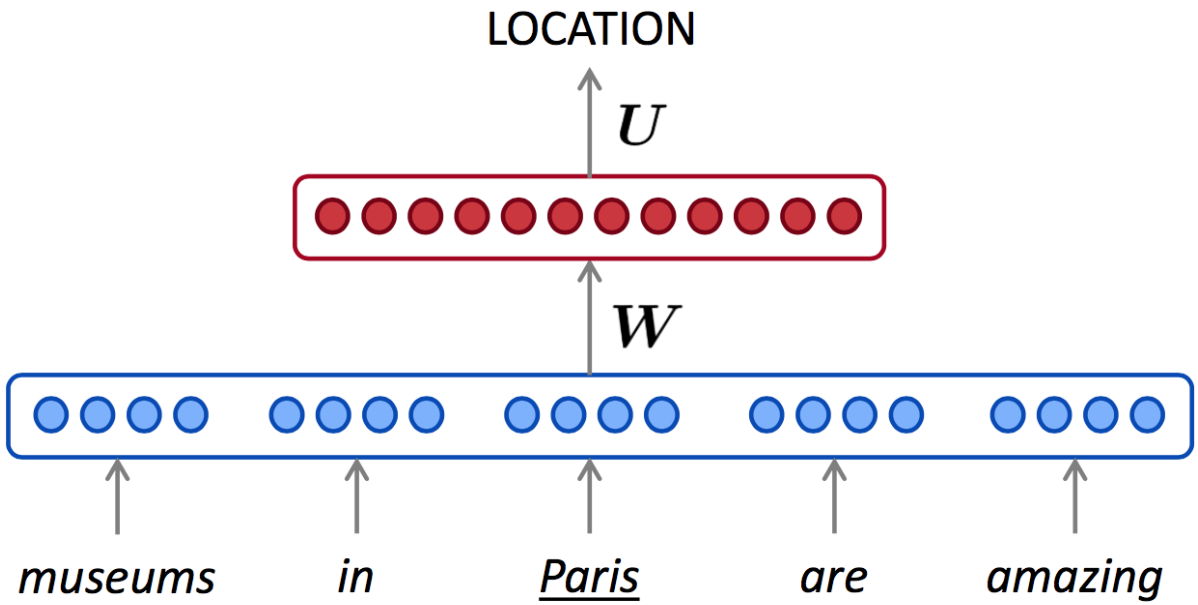
Lecture Structure:

- Recap - neural models (fixed-window neural models and RNNs)
- Recap - problems with RNNs - the vanishing gradient problem
- RNNs variants: LSTM, GRU, bi-directional, stacked

- An application of stacked RNNs: Opinion Mining with Deep Recurrent Nets by Irsoy and Cardie 2014. https://www.cs.cornell.edu/ ∼oirsoy/files/emnlp14drnt.pdf

A window-based neural model

- Applied to Named Entity Recognition:



Language Modeling

- Language Modeling task:
  - Input: sequence of words $x^{(1)}, x^{(2)}, \cdots, x^{(t)}$
  - Output: probability distribution of the next word $x^{(t+1)}$: $P(x^{(t+1)}|x^{(t)}, \cdots, x^{(1)})$

- A Language Model also assigns probability to a piece of text.

- For example, if we have some text $x^{(1)}, x^{(2)}, \cdots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

$$P(x^{(1)}, \cdots, x^{(T)}) = P(x^{(1)}) \times P(x^{(2)}|x^{(1)}) \times \cdots \times P(x^{(T)}|x^{(T-1)}, \cdots, x^{(1)})$$

$$= \prod_{t=1}^{T} \underbrace{P(x^{(t)}|x^{(t-1)}, \cdots, x^{(2)}, x^{(1)})}_{\text{From LM}}$$

A fixed-window neural Language Model

output distribution

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

hidden layer

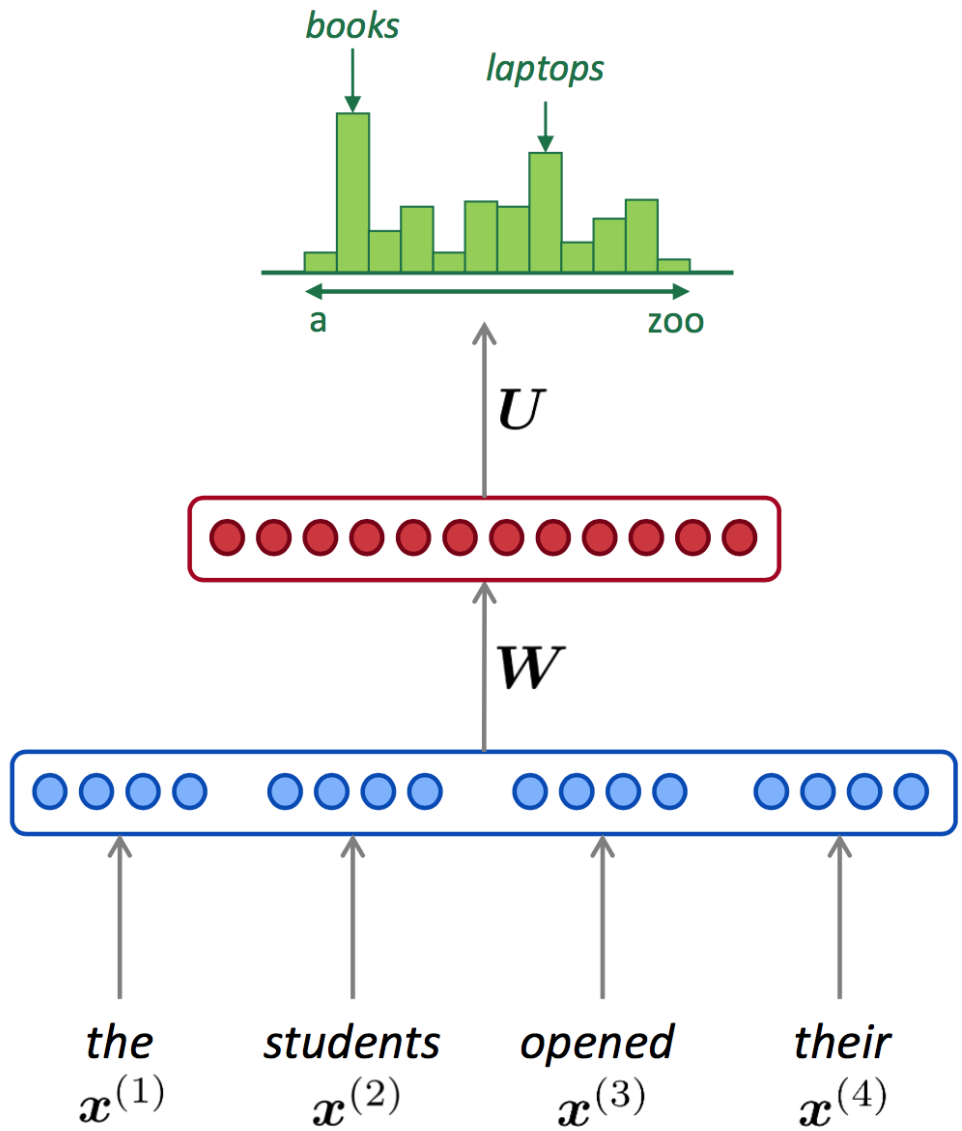
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



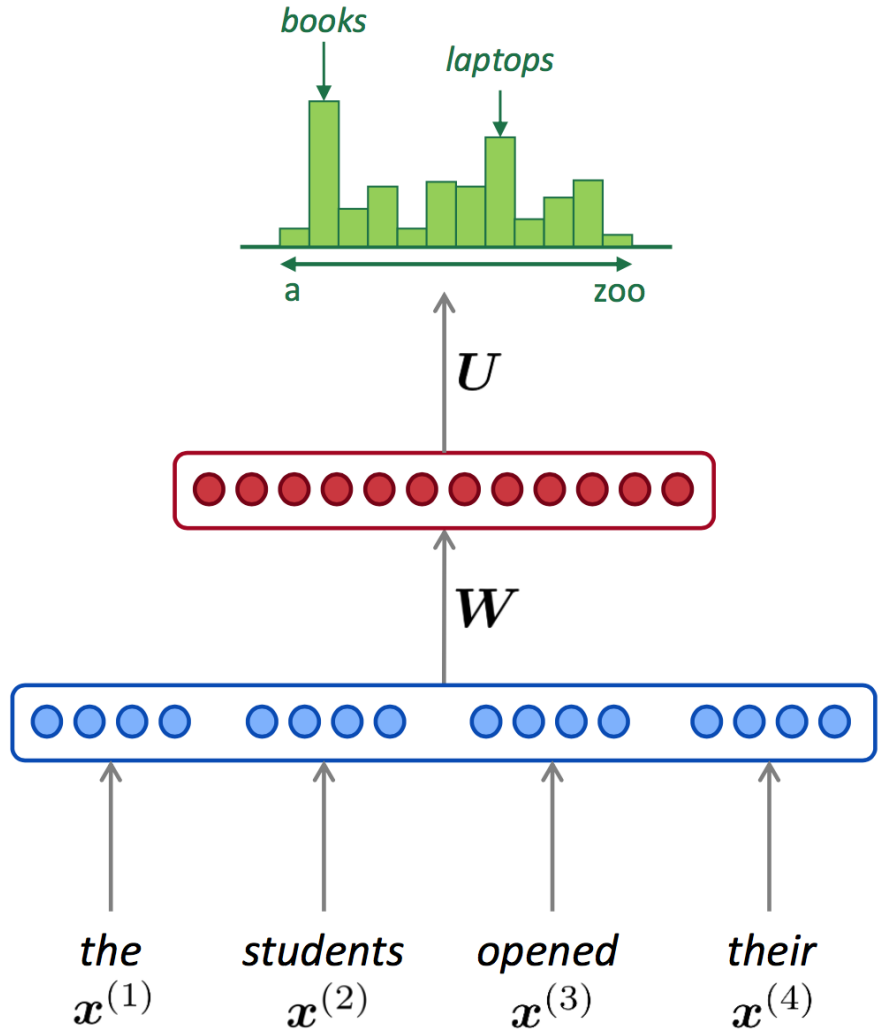
A fixed-window neural Language Model

Improvements over n-gram LM:

- No sparsity problem.
- Do not need to store all observed n-grams.

Remaining problems:

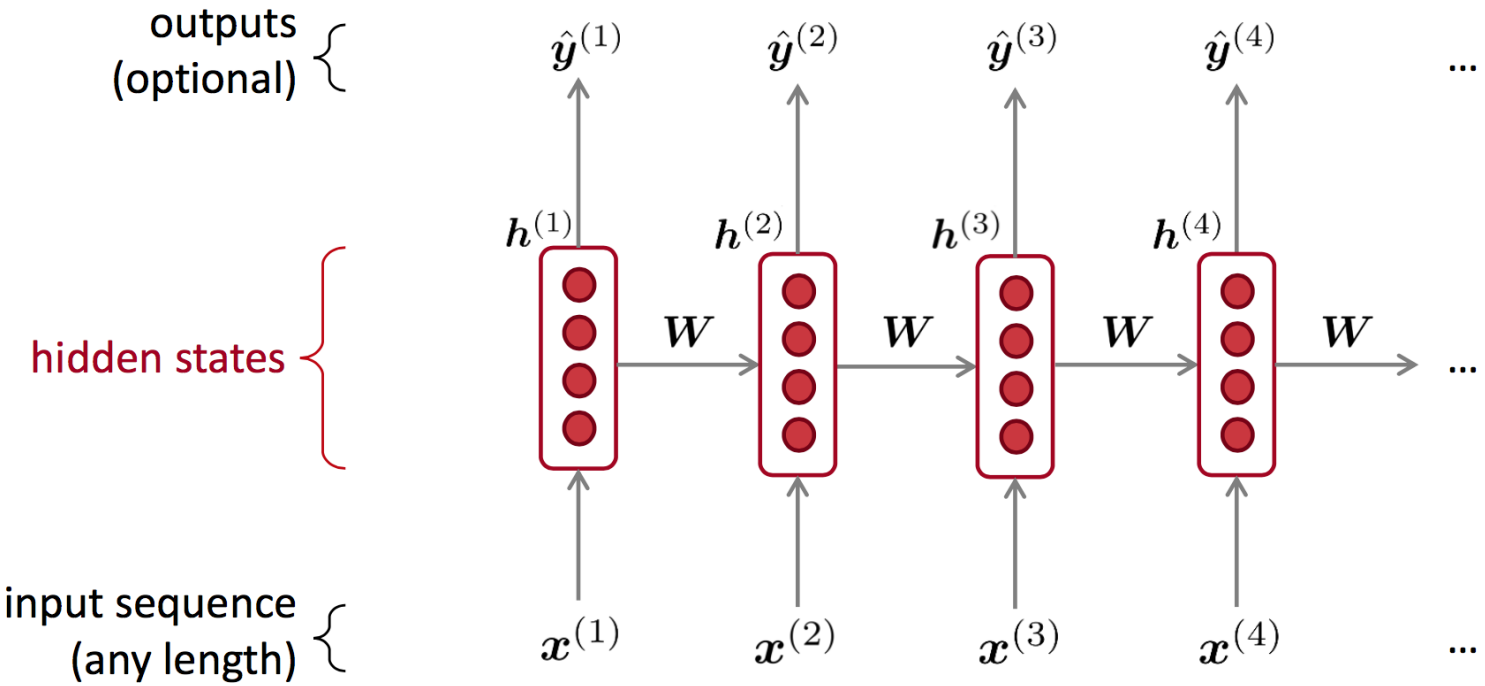
- Fixed window is too small.
- Enlarging window enlarges W.
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W: Inefficient to learn separate weights for different words, when there are commonalities between them.



Recurrent Neural Networks (RNN)

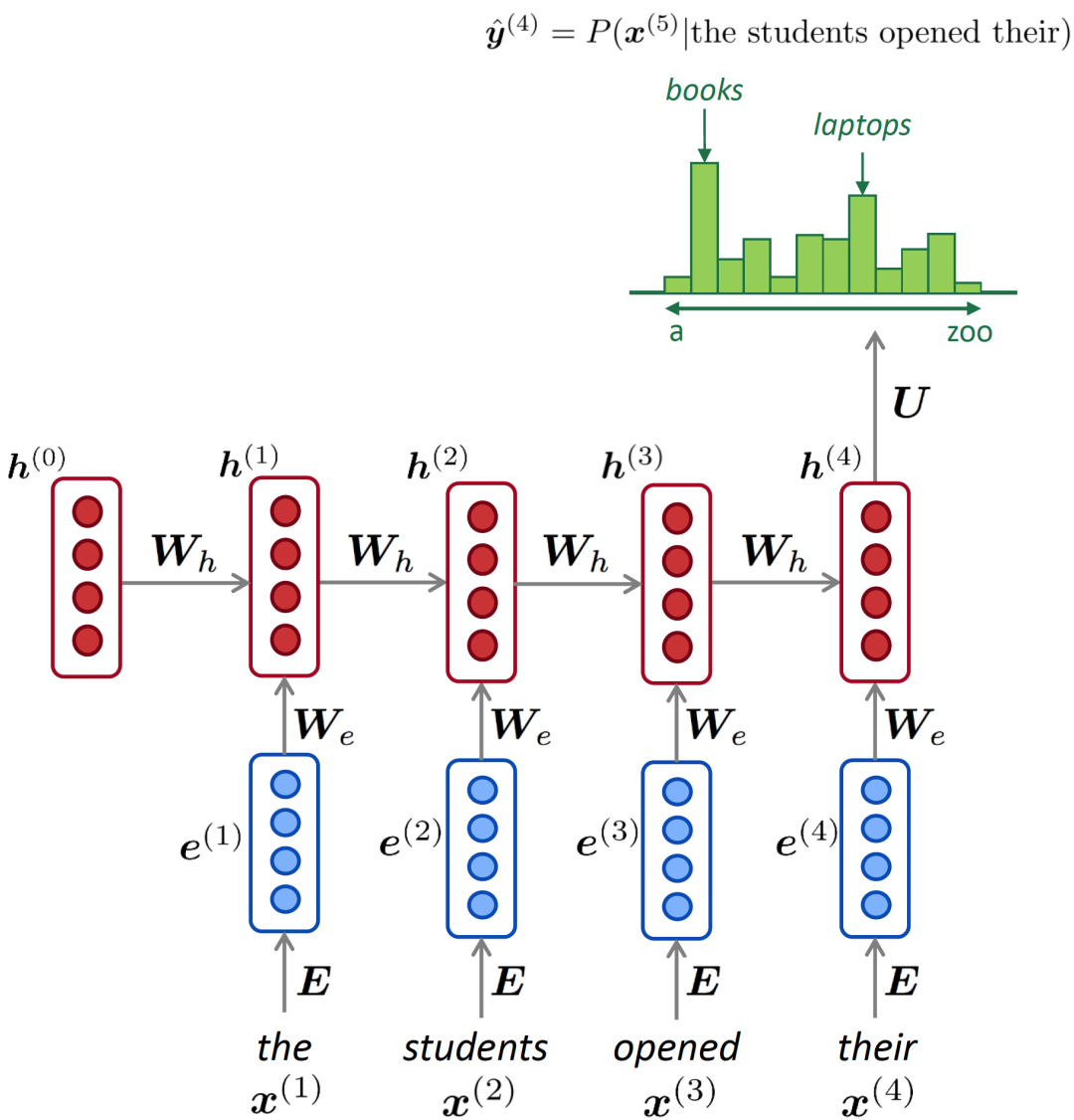
A family of neural architectures

- RNNs have recurrent connections, which allow a form of memory, and are suitable for tasks with arbitrary temporal dimensions.



Core idea: Apply the same weights W *repeatedly*!

An RNN Language Model



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

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

Note: the input sequence could be much longer.

An RNN Language Model

words / one-hot vectors
$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$

the	*students*	*opened*	*their*
$\boldsymbol{x}^{(1)}$	$\boldsymbol{x}^{(2)}$	$\boldsymbol{x}^{(3)}$	$\boldsymbol{x}^{(4)}$

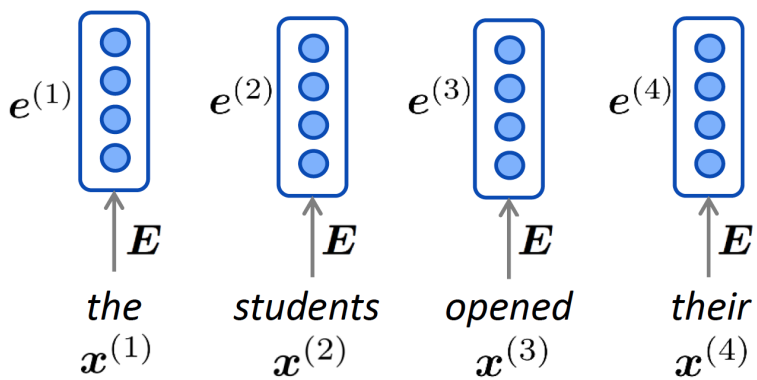
An RNN Language Model

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

hidden states

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

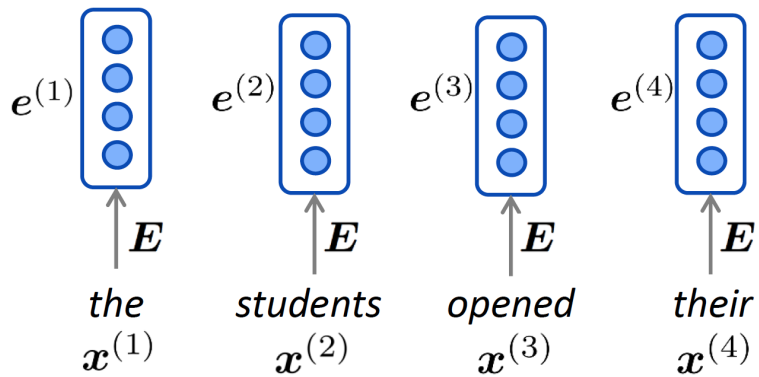
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

hidden states

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

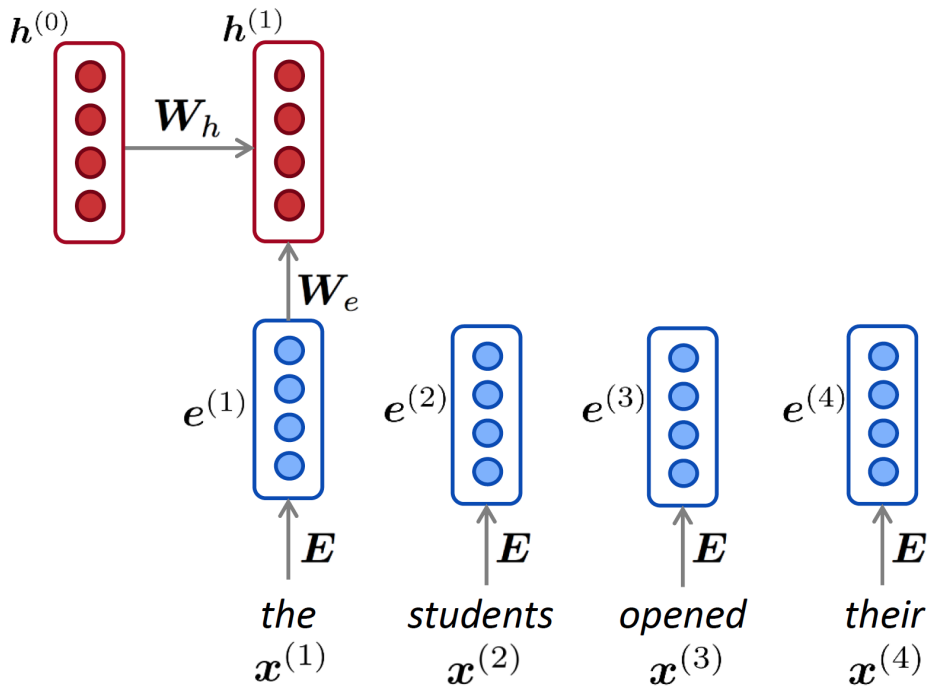
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

hidden states

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

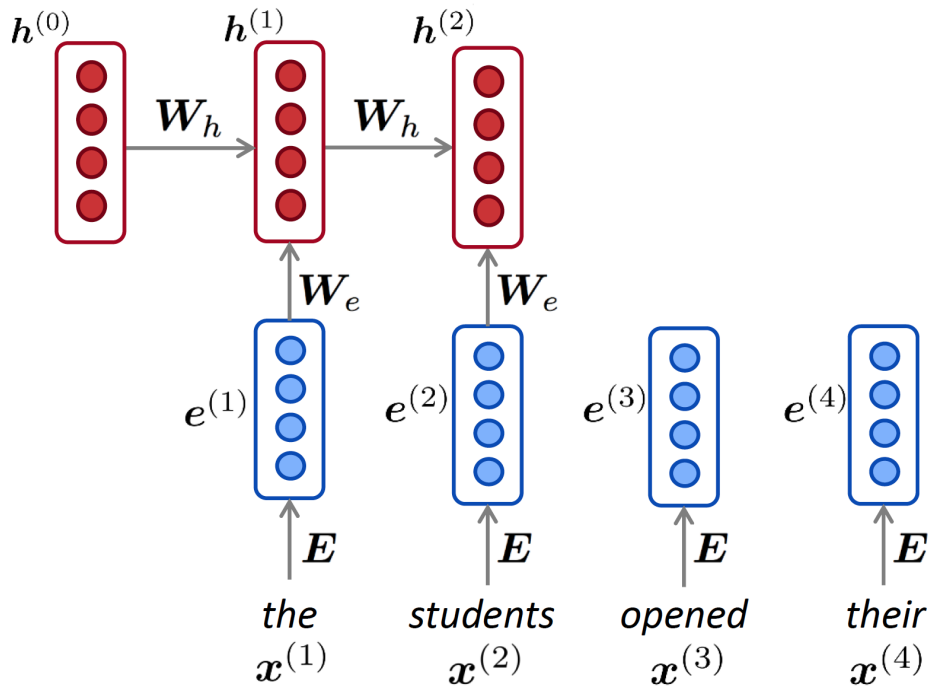
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

hidden states

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

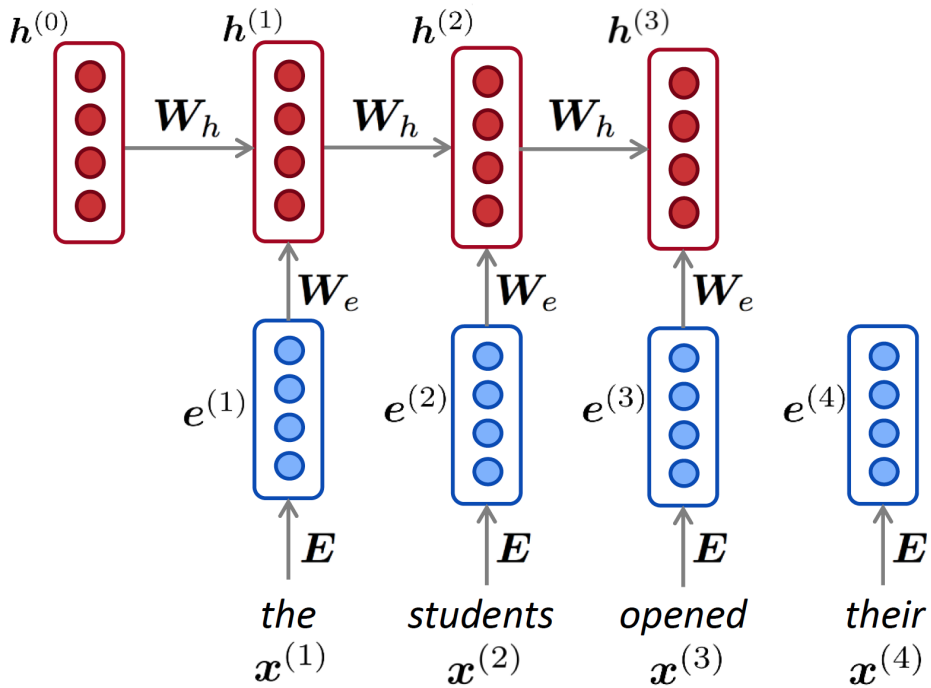
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

hidden states

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

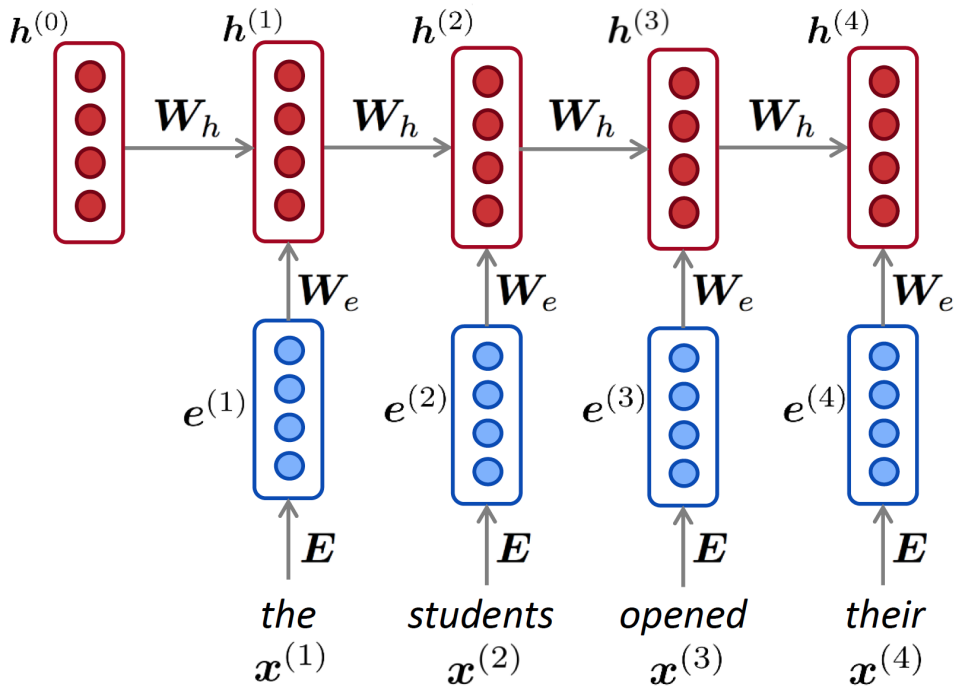
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



An RNN Language Model

output distribution

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

hidden states

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

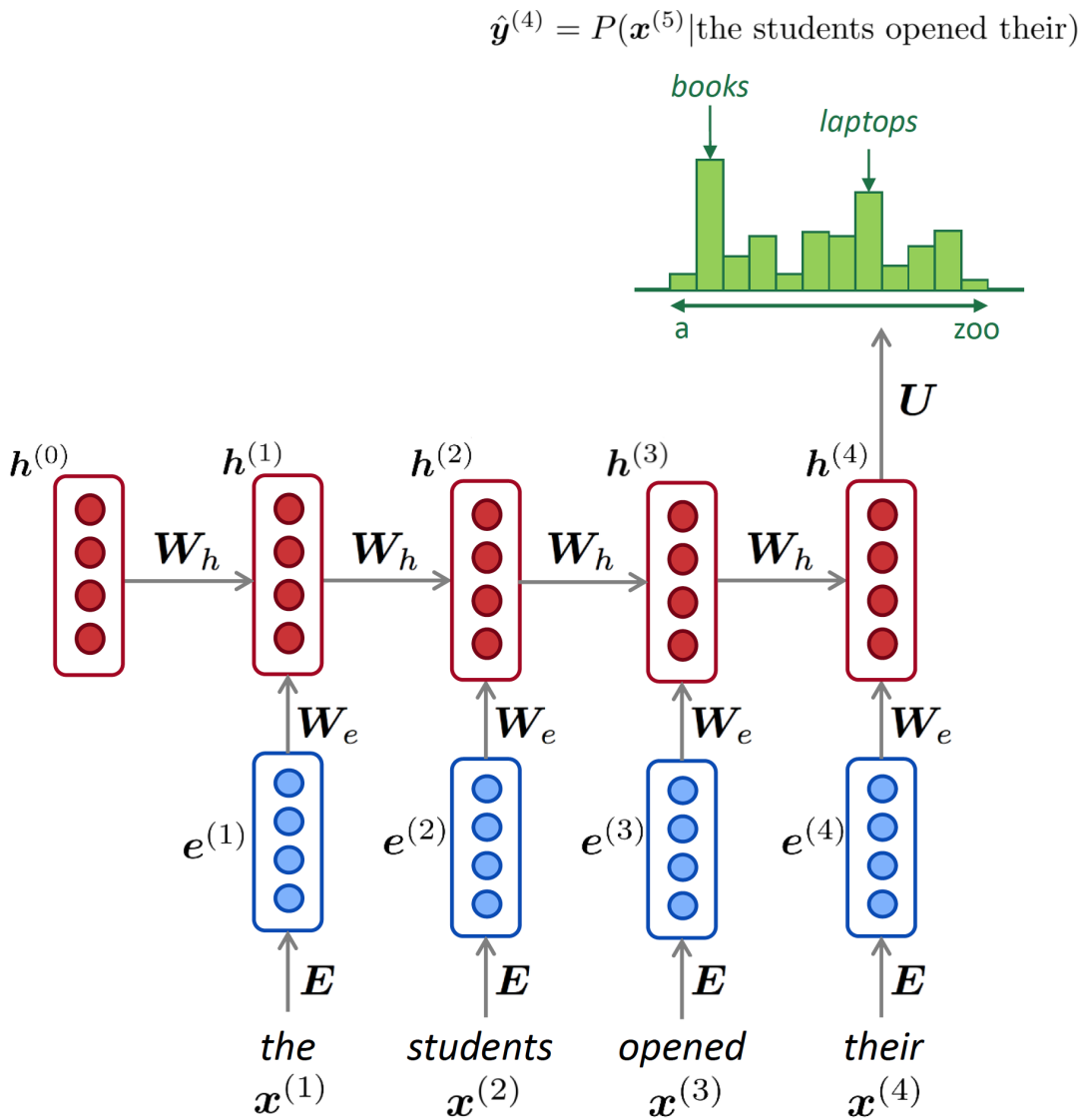
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



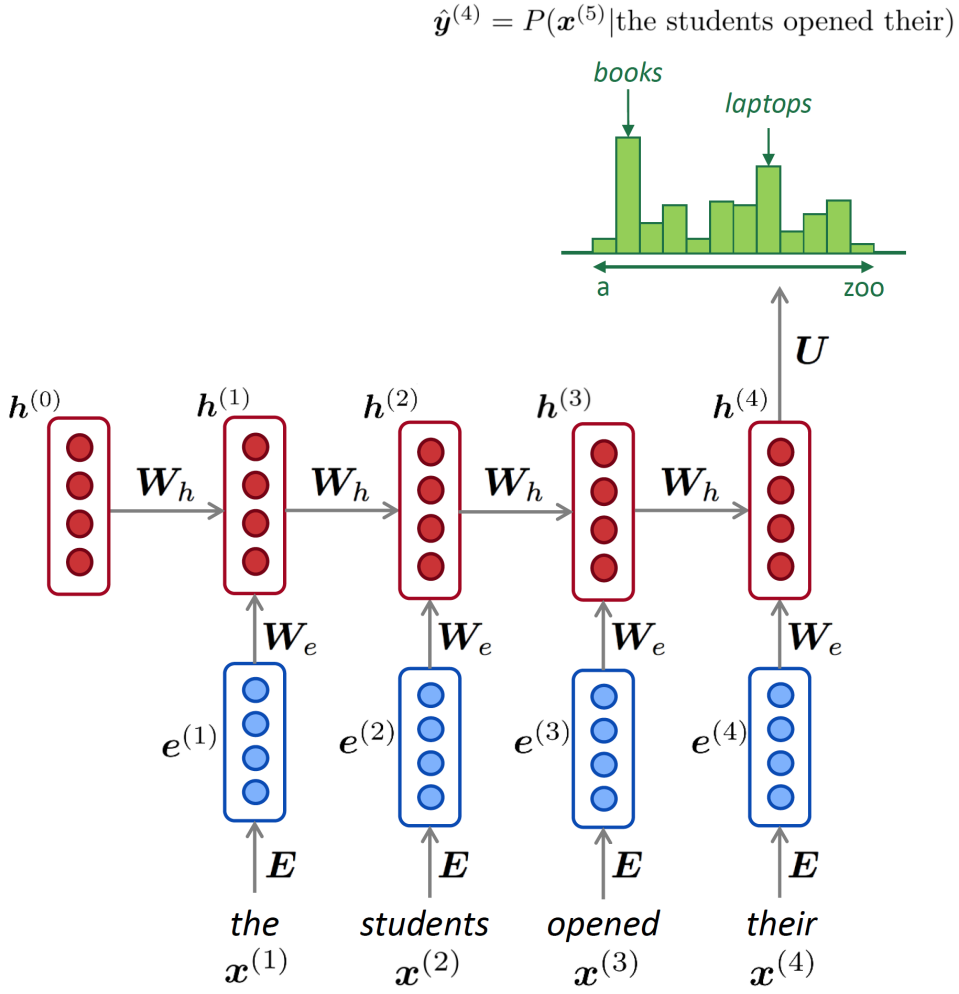
An RNN Language Model

RNN **Advantages:**

- Can process any length input
- Computation for step t can (in theory) use information from many steps back
- Model size does not increase for longer input
- Same weights applied on every time step, so we learn some general functions of the language.

RNN **Disadvantages:**

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



Training an RNN Language Model

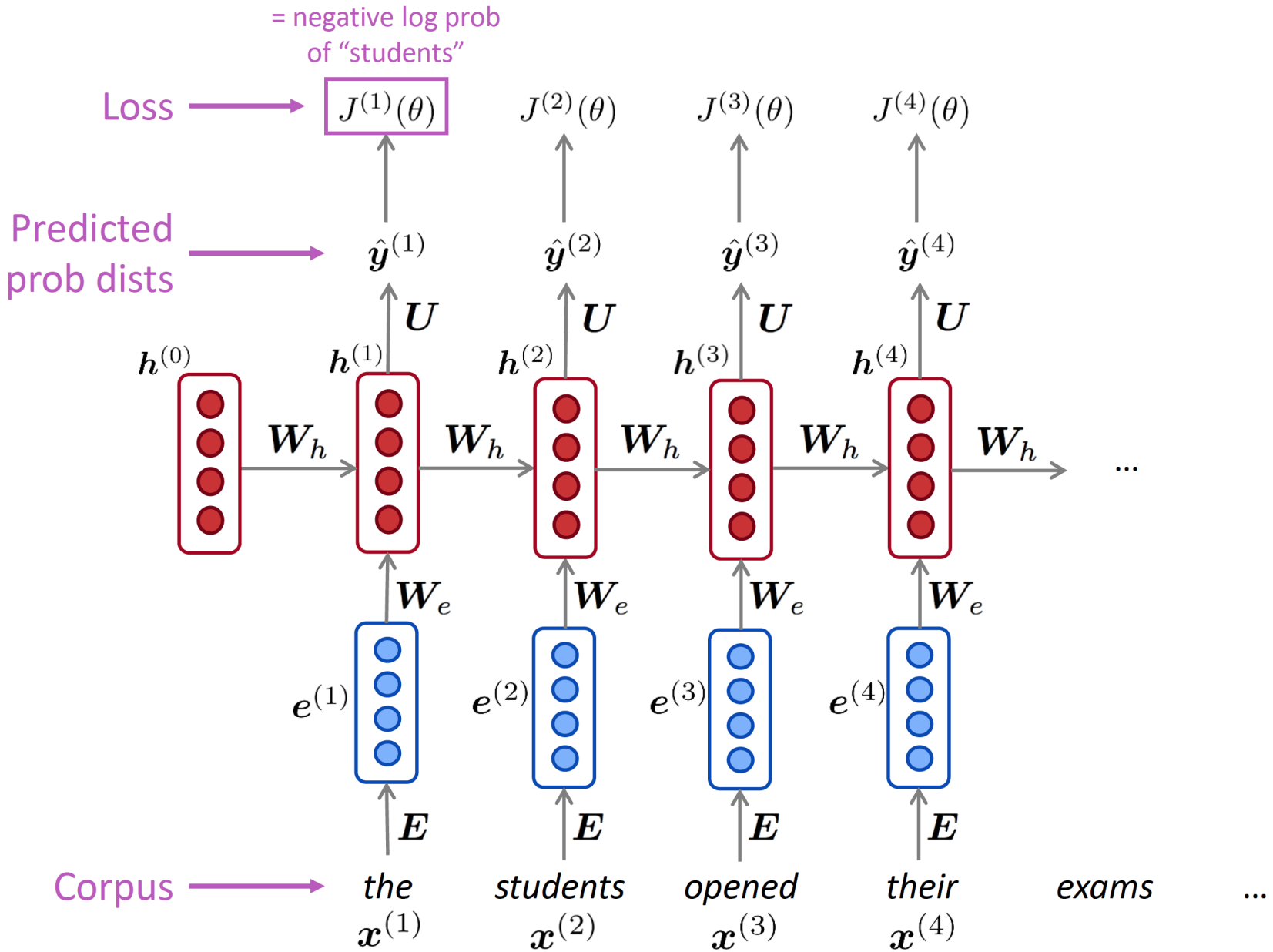
- Get a big corpus of text which is a sequence of words $x_1, ..., x_T$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ for every step t.
  - i.e. predict probability dist of every word, given words so far
- Loss function on step t is cross-entropy between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

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

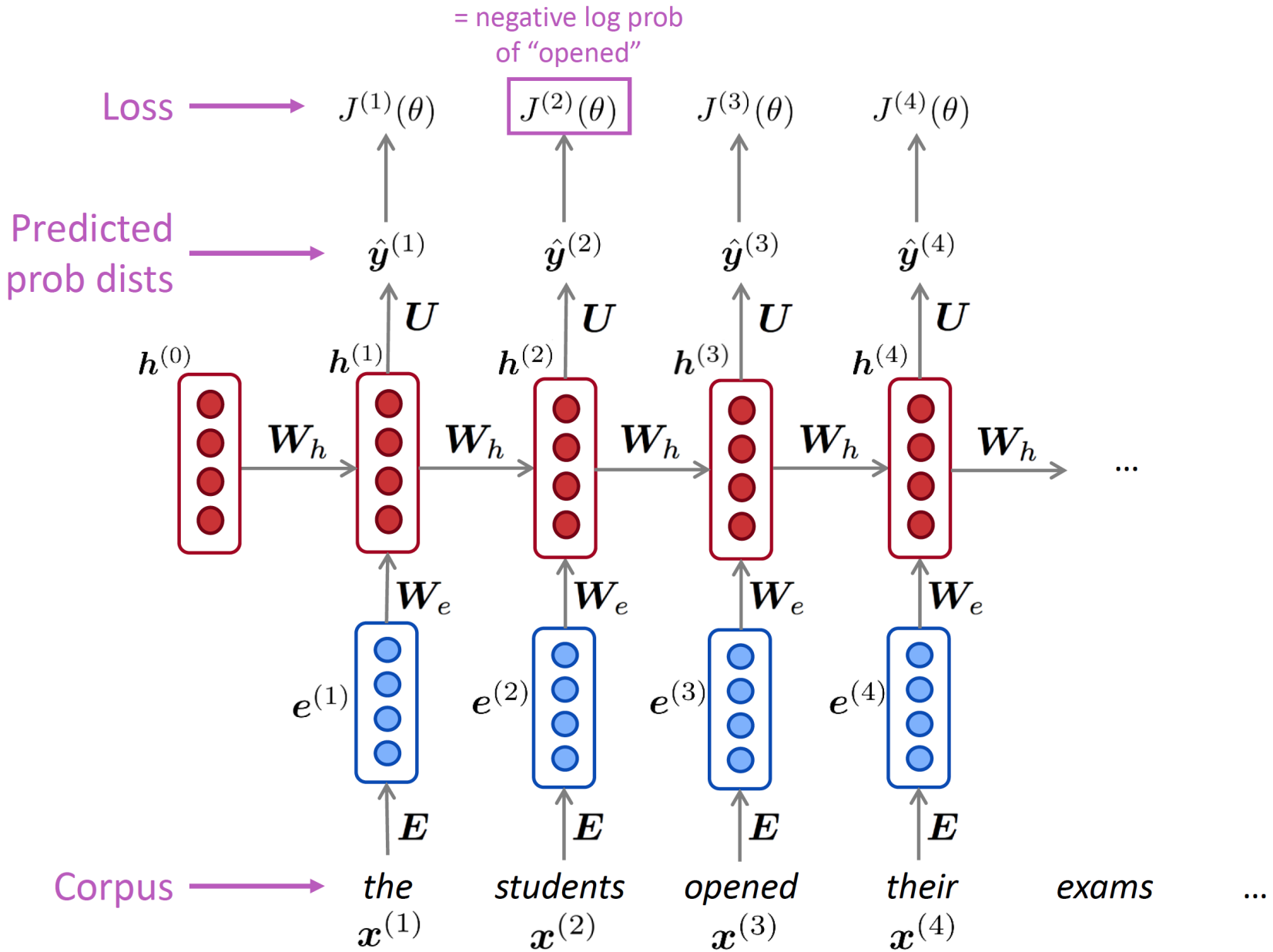
- Average this to get overall loss for entire training set:

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

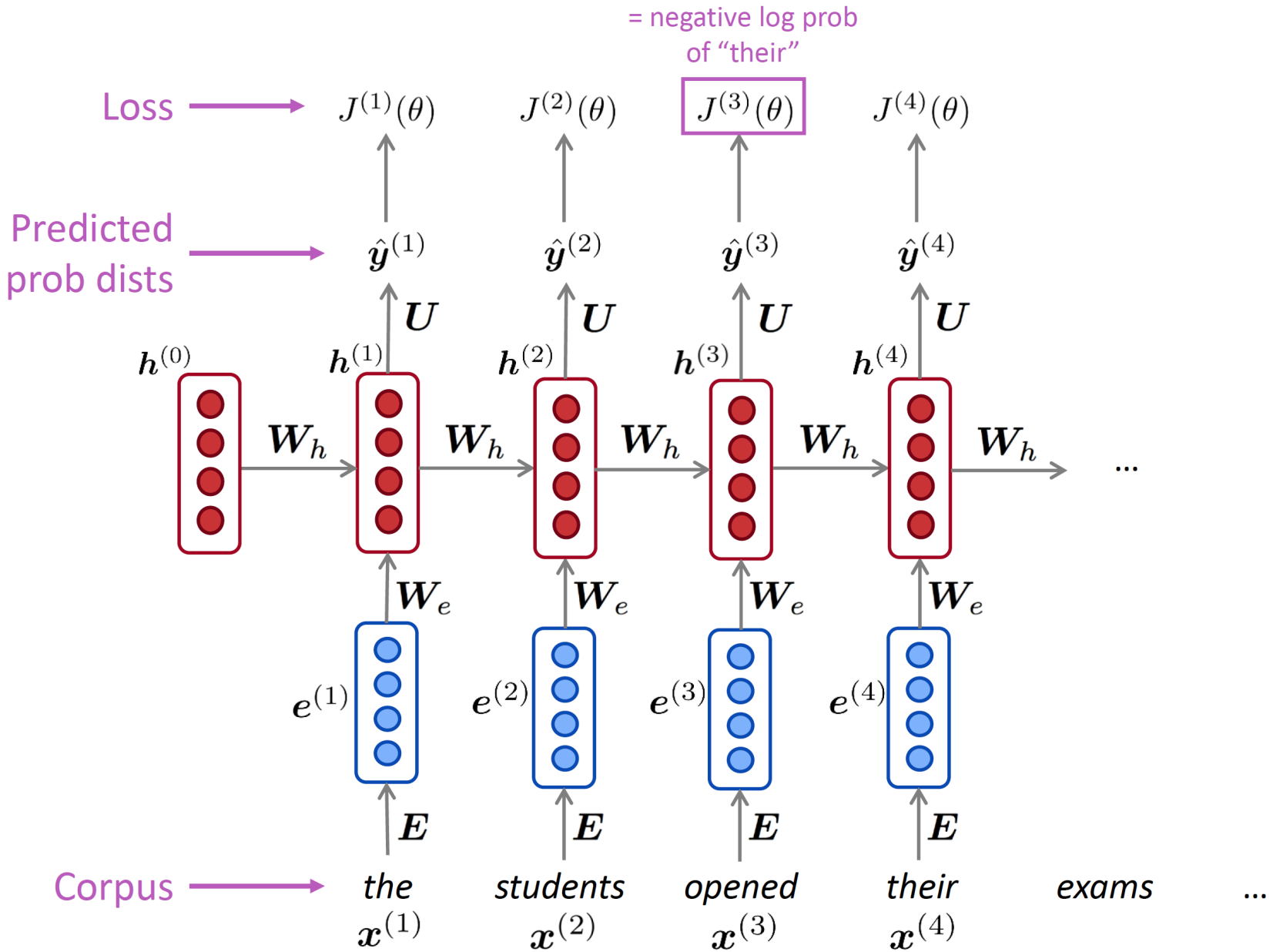
Training an RNN Language Model



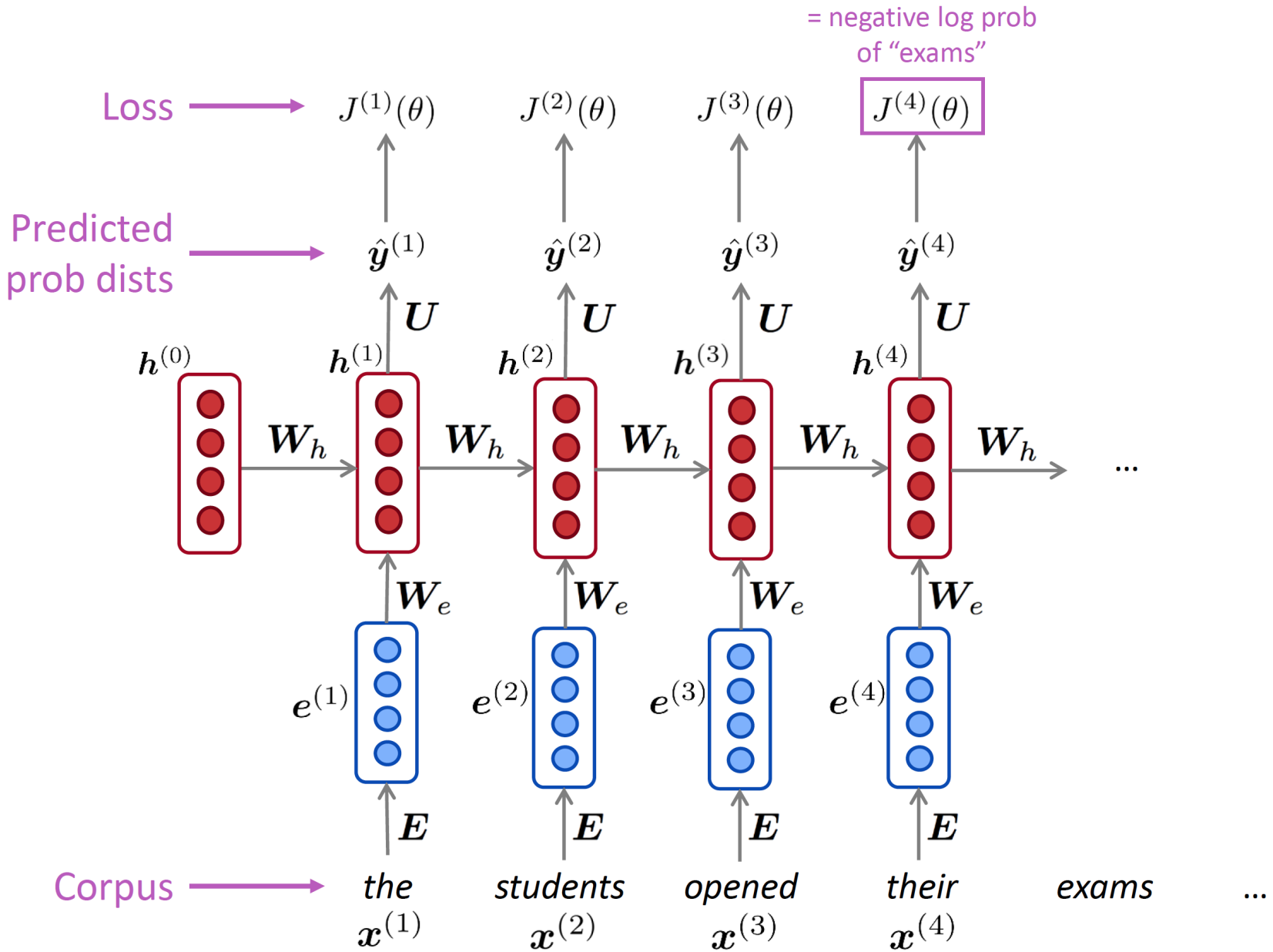
Training an RNN Language Model



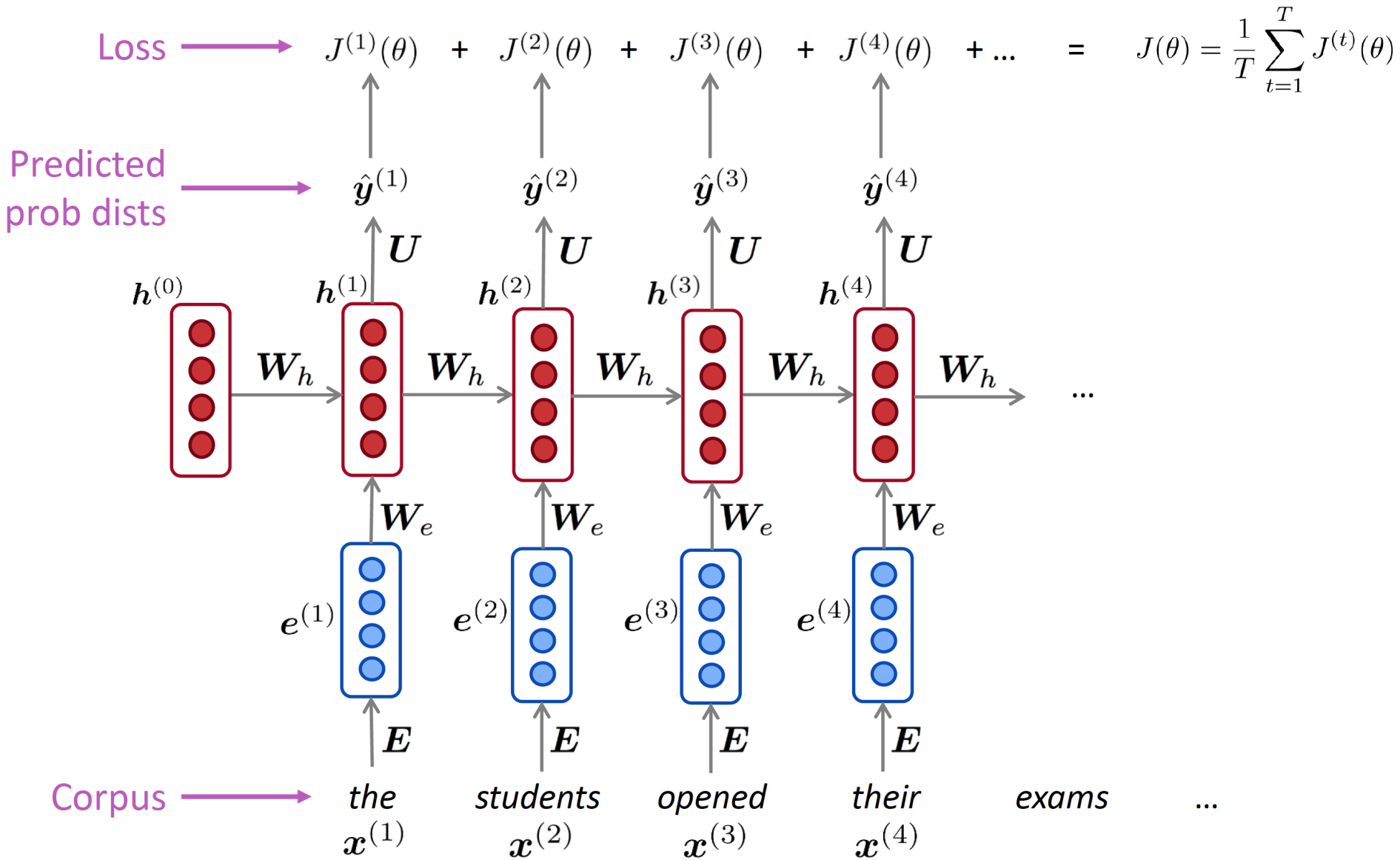
Training an RNN Language Model



Training an RNN Language Model



Training an RNN Language Model



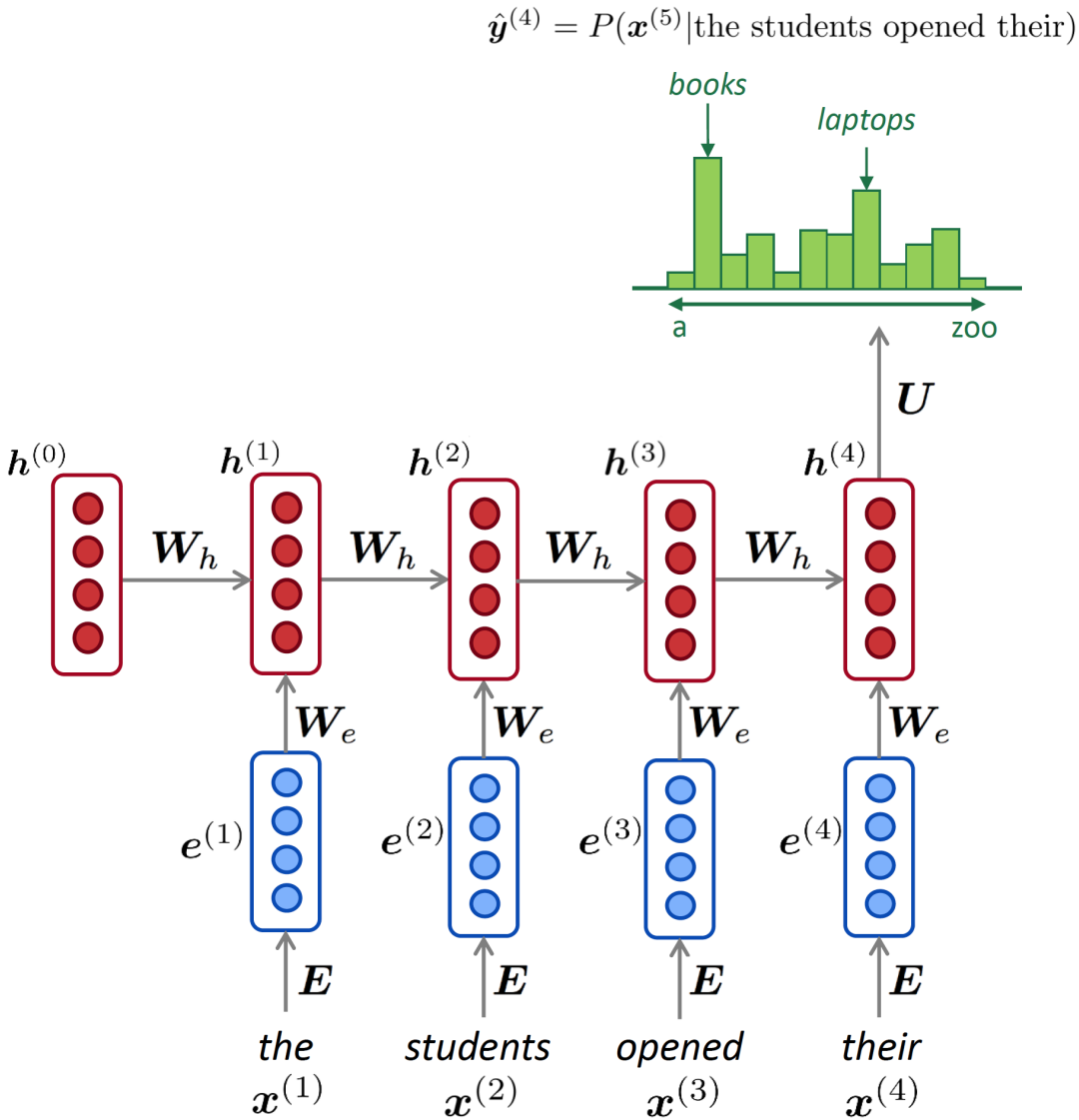
Training an RNN Language Model

- 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 sentence (or a document)
- Recall: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.
- Compute loss $J(\theta)$ for a sentence (actually a batch of sentences), compute gradients and update weights. Repeat.

An RNN Language Model



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

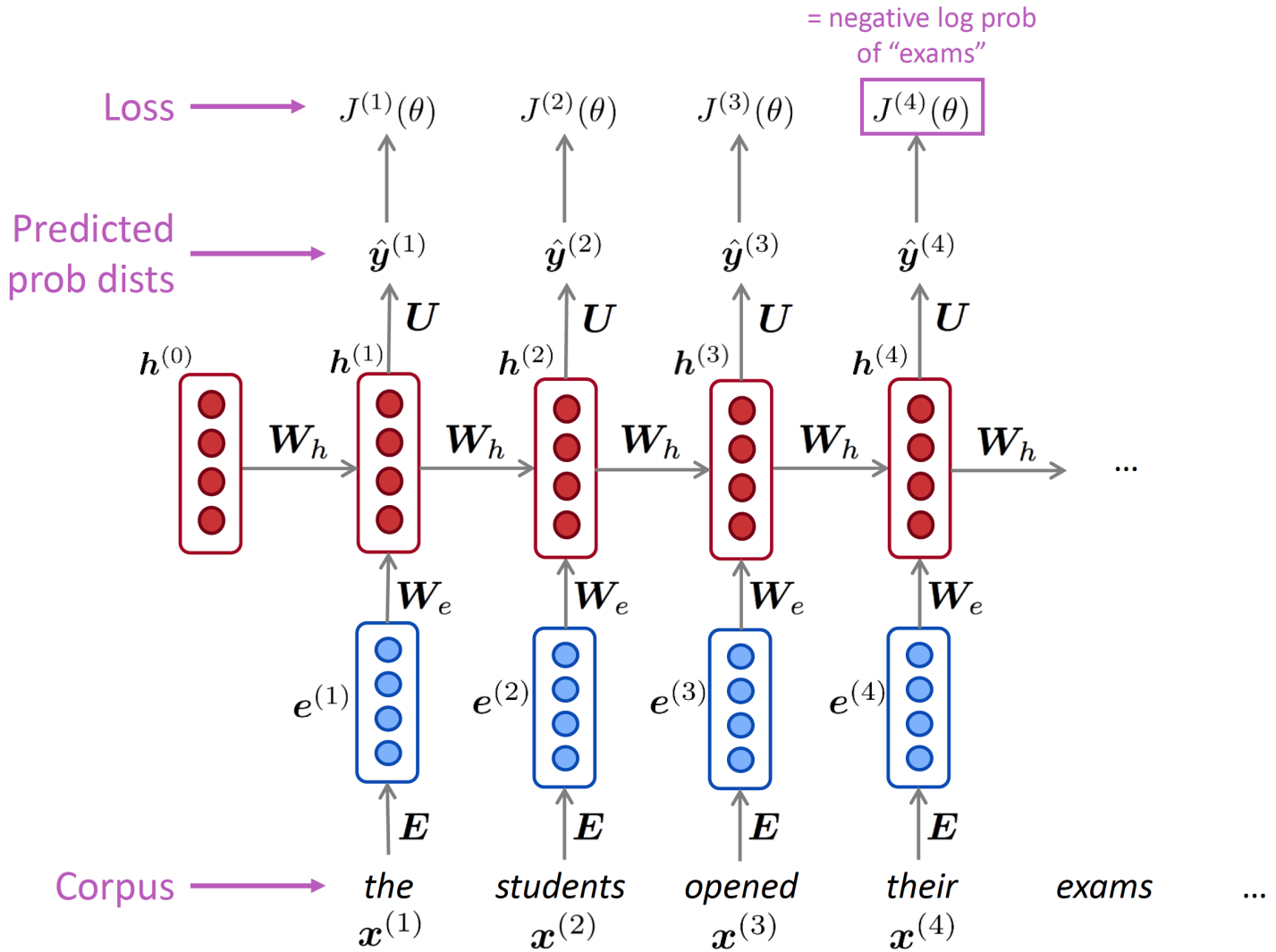
$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

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

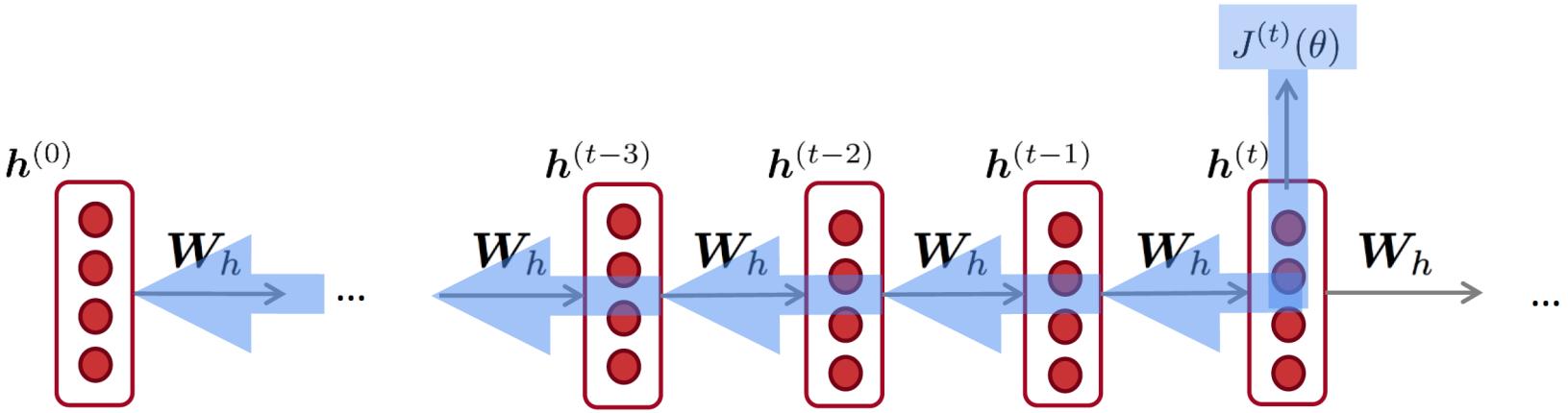
Note: we compute the gradient of the loss wrt all the involved variables: U, b_2, W_h, W_e, b_1, x.

An RNN Language Model



Note: we compute the gradient of the loss wrt all the involved variables: U, b_2, W_h, W_e, b_1, x.

Backpropagation for RNNs



$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}$$

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i*=*t*,…,0, summing gradients as you go.
This algorithm is called
"backpropagation through time"

Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.

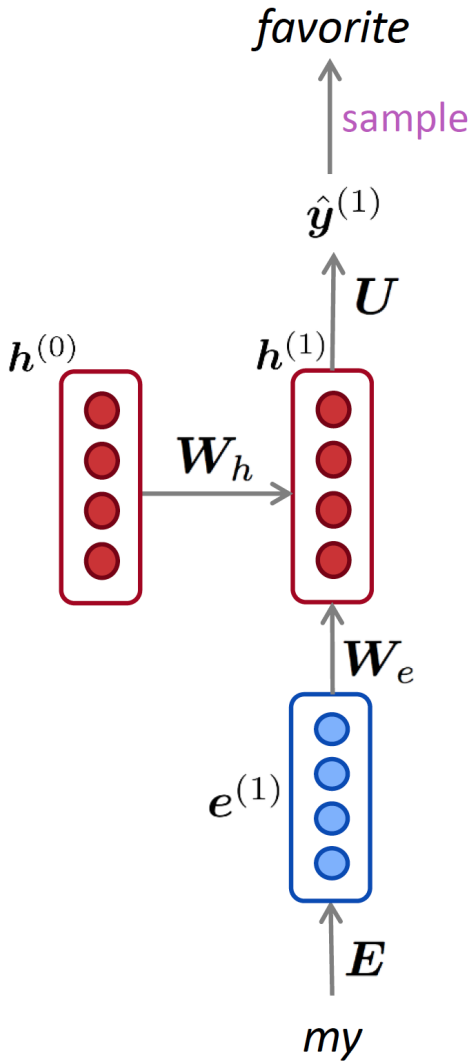
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.

my

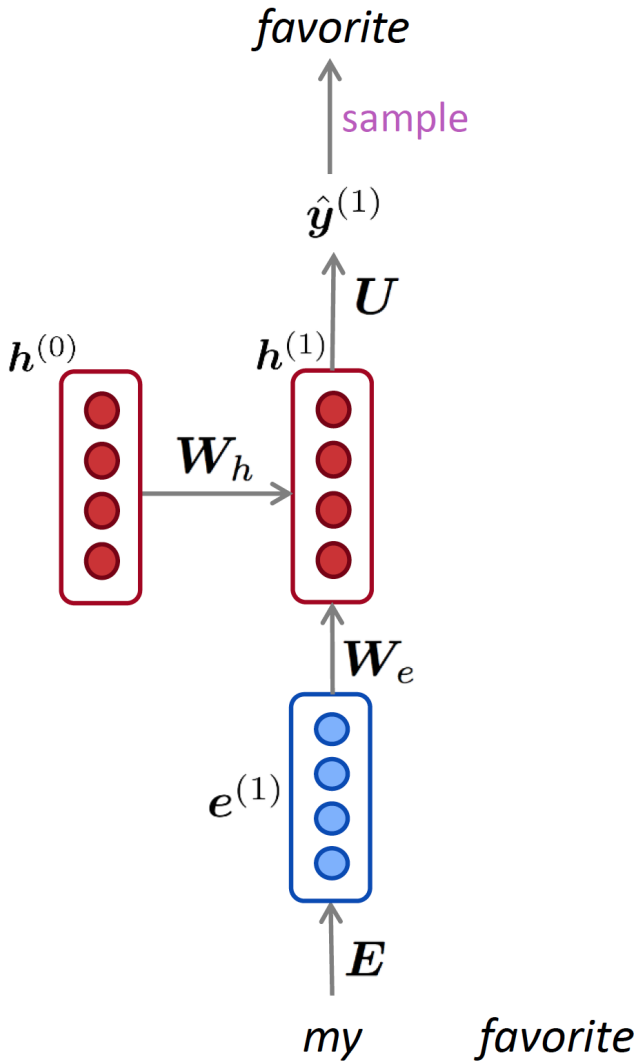
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



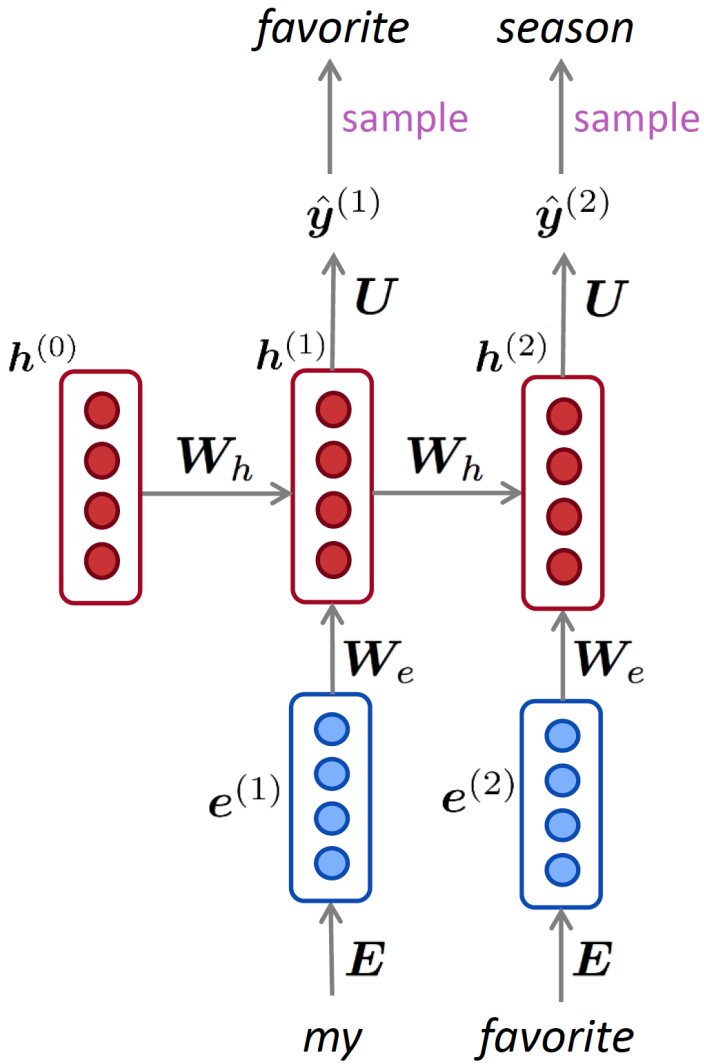
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



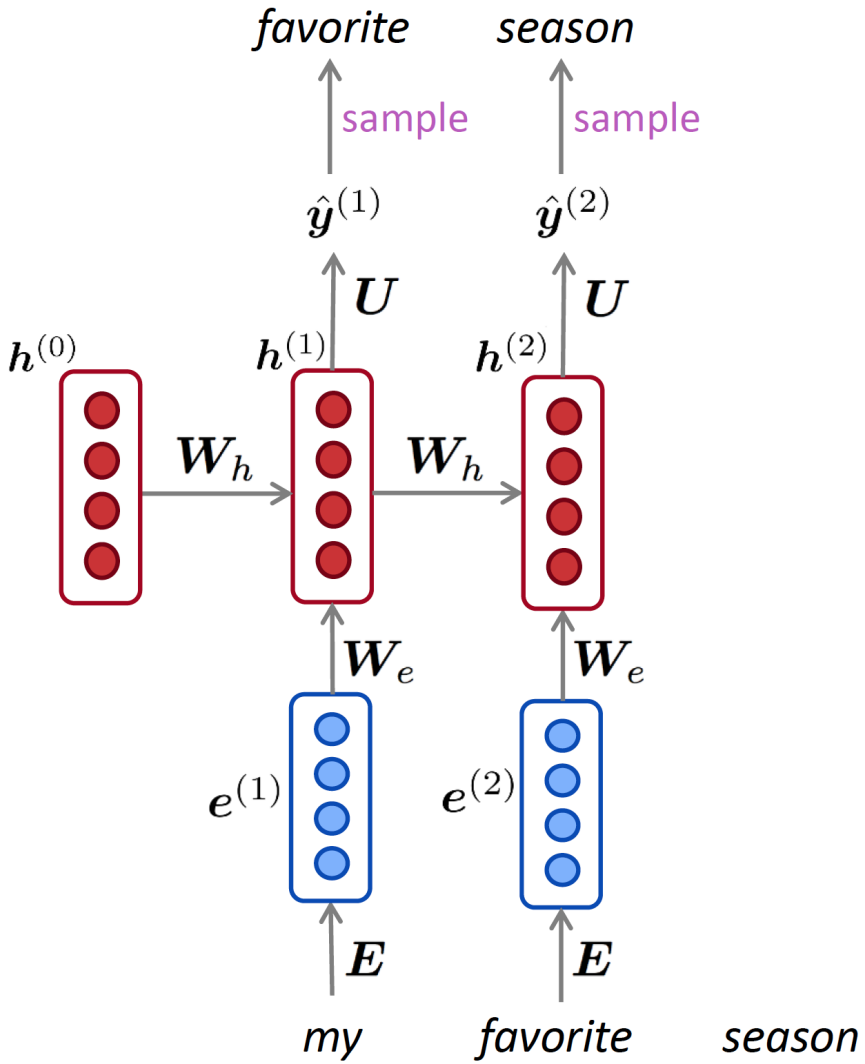
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



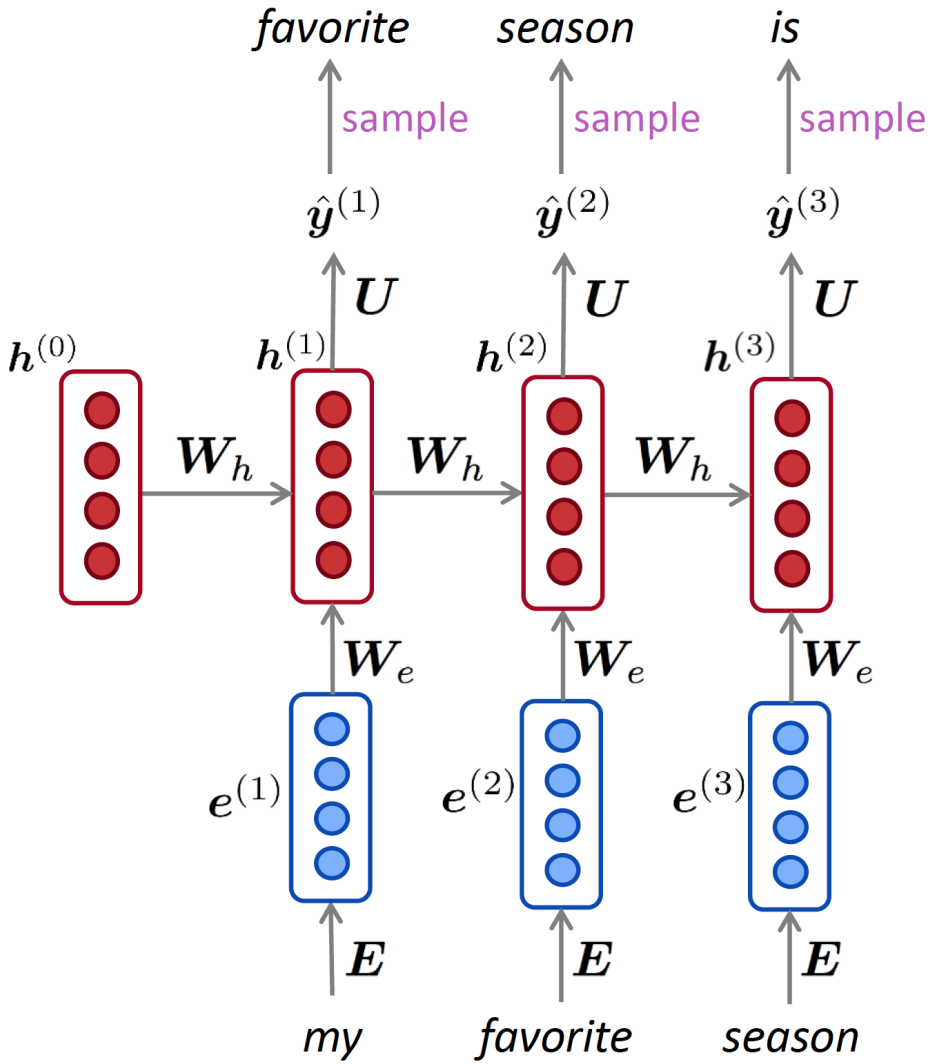
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



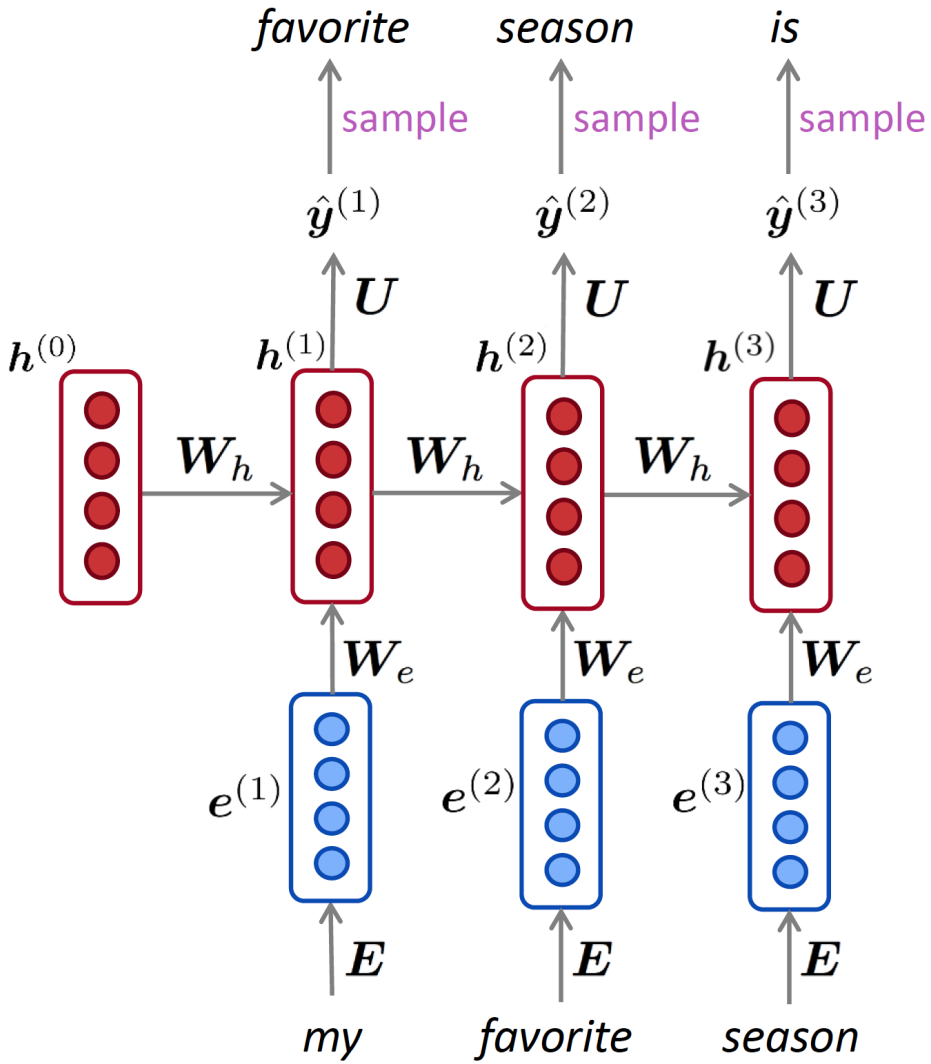
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



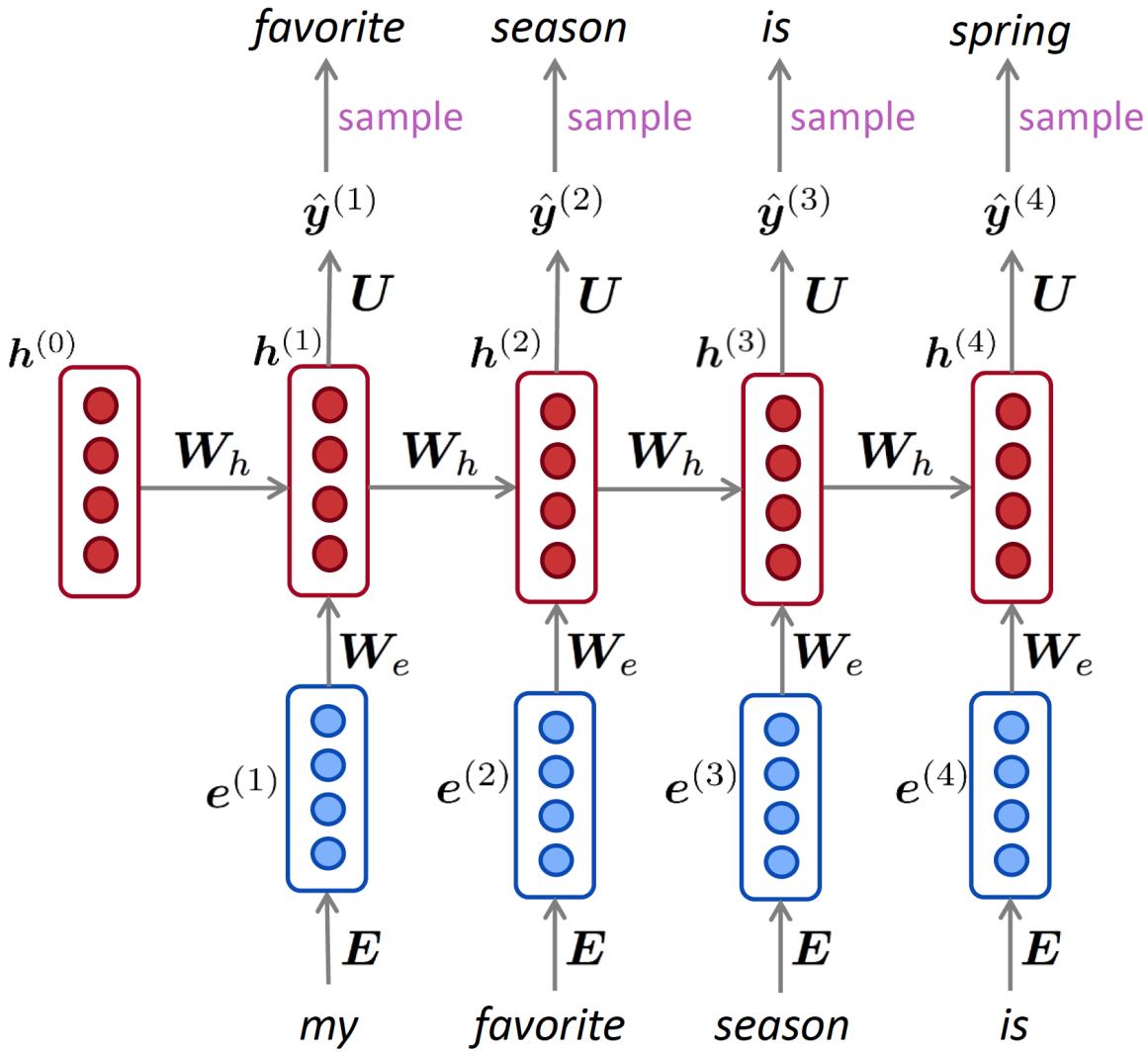
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



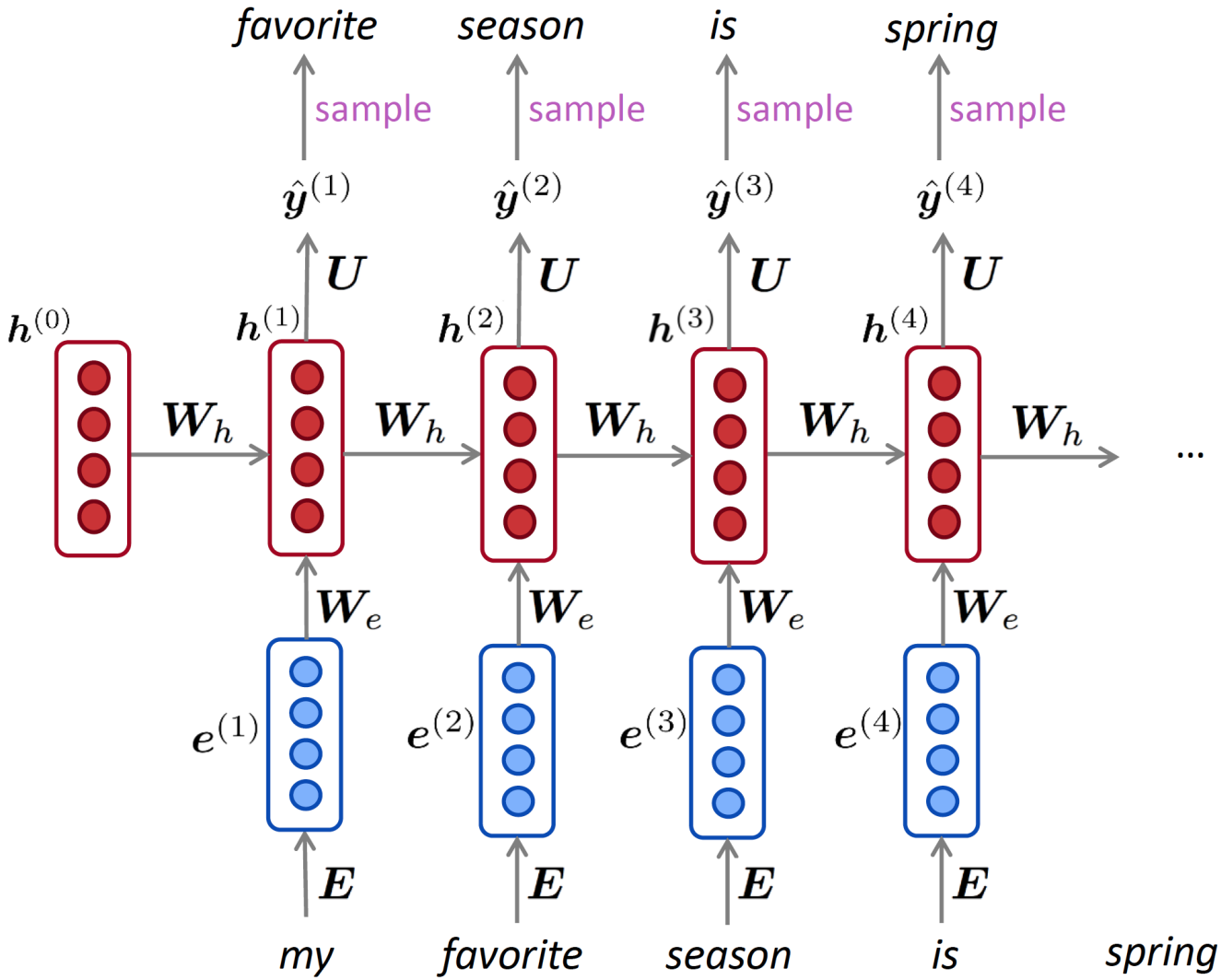
Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



Generating text with an RNN LM

Just like an n-gram LM, we can use an RNN LM to generate text by repeated sampling. Sampled output is next step's input.



Problems with RNNs

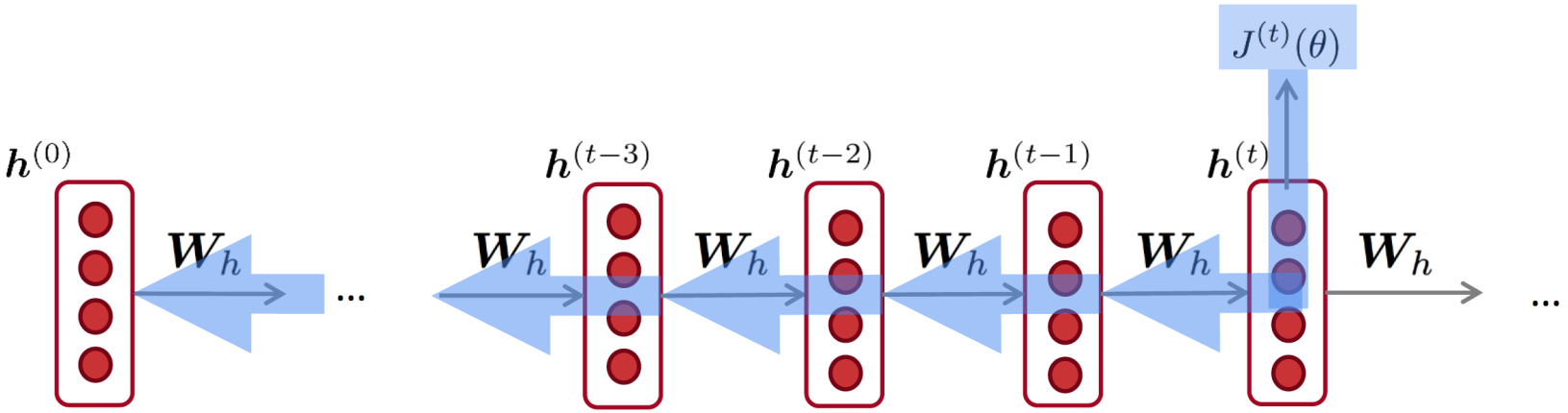
- Vanishing gradient problem
- Remember that we have:

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

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

- Note: we compute the gradient of the loss wrt all the involved variables: U, b_2, W_h, W_e, b_1, x.

Backpropagation for RNNs

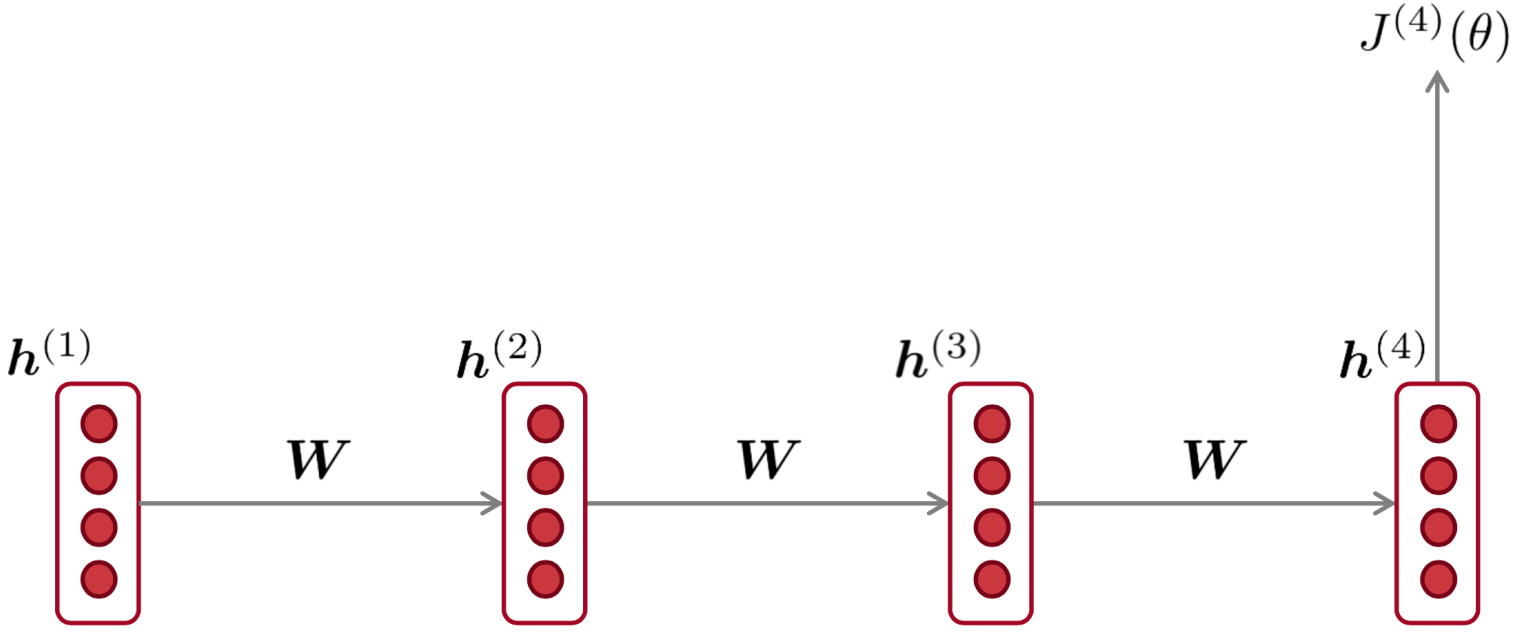


$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}$$

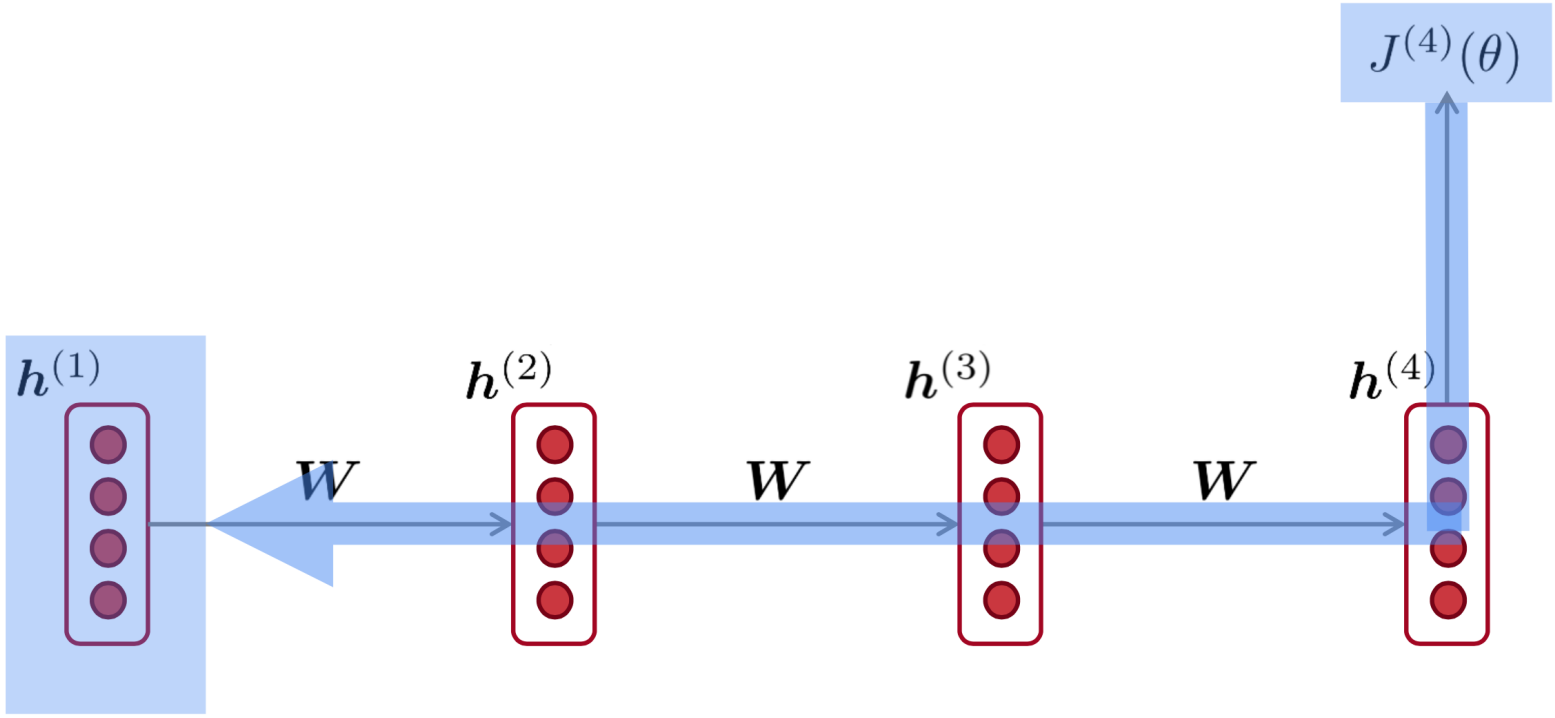
Question: How do we calculate this?

Answer: Backpropagate over timesteps *i*=*t*,…,0, summing gradients as you go.
This algorithm is called
"backpropagation through time"

Vanishing gradient intuition

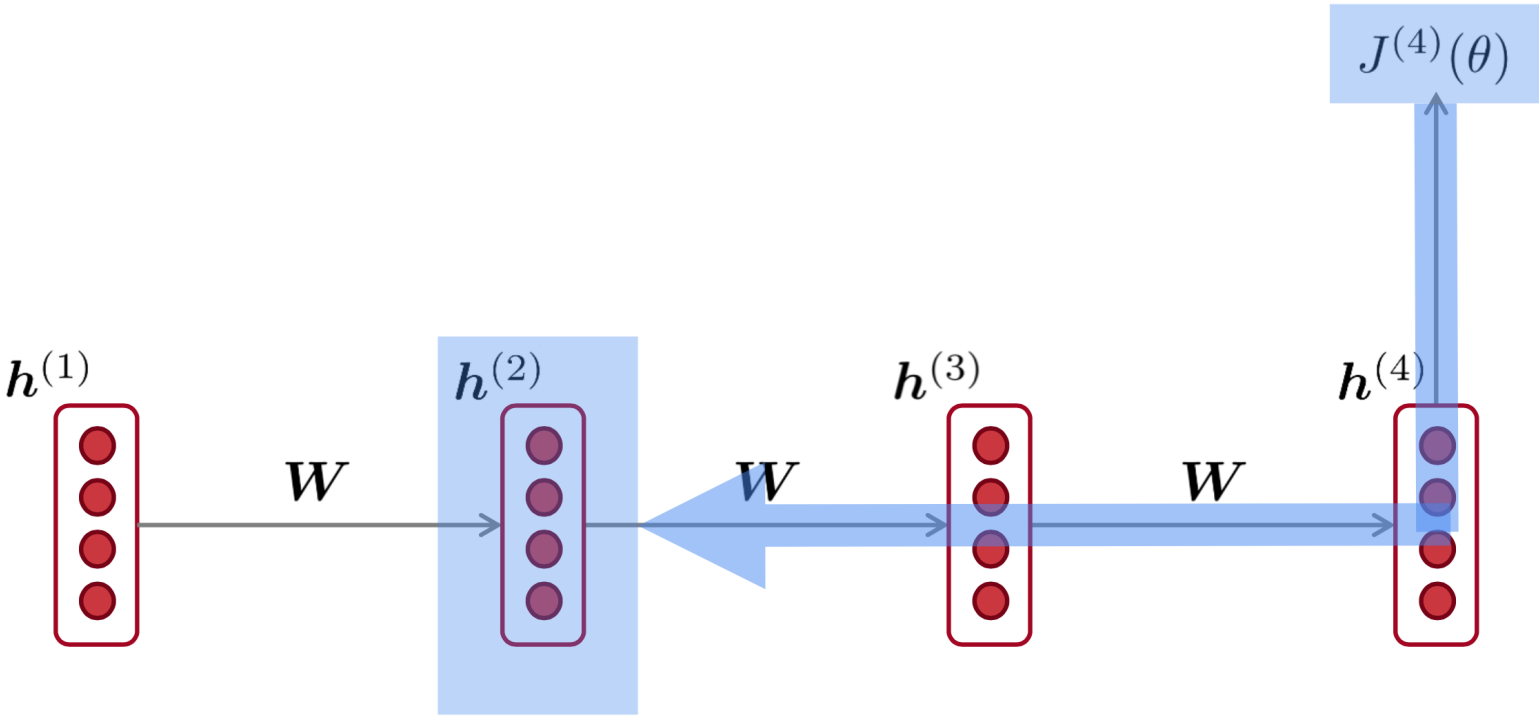


Vanishing gradient intuition



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

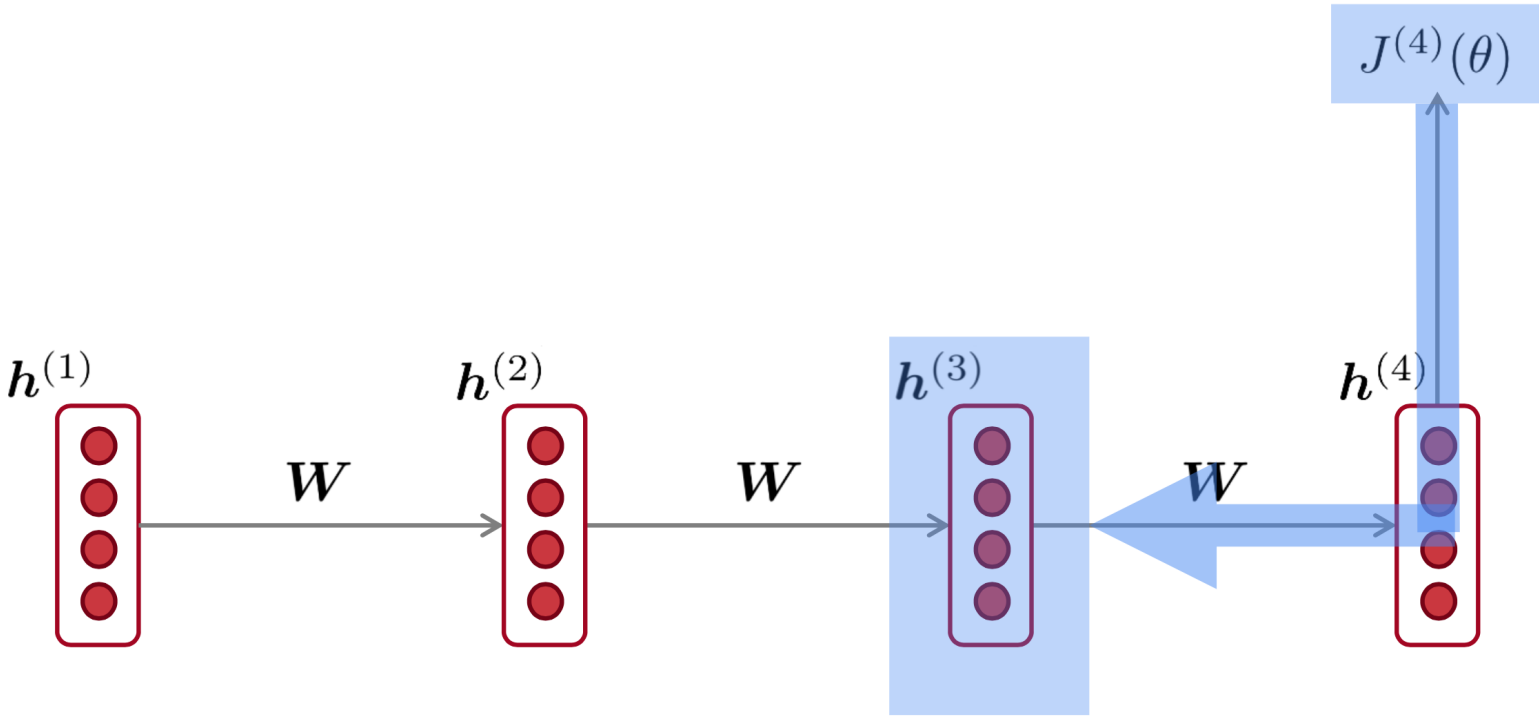
Vanishing gradient intuition



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

chain rule!

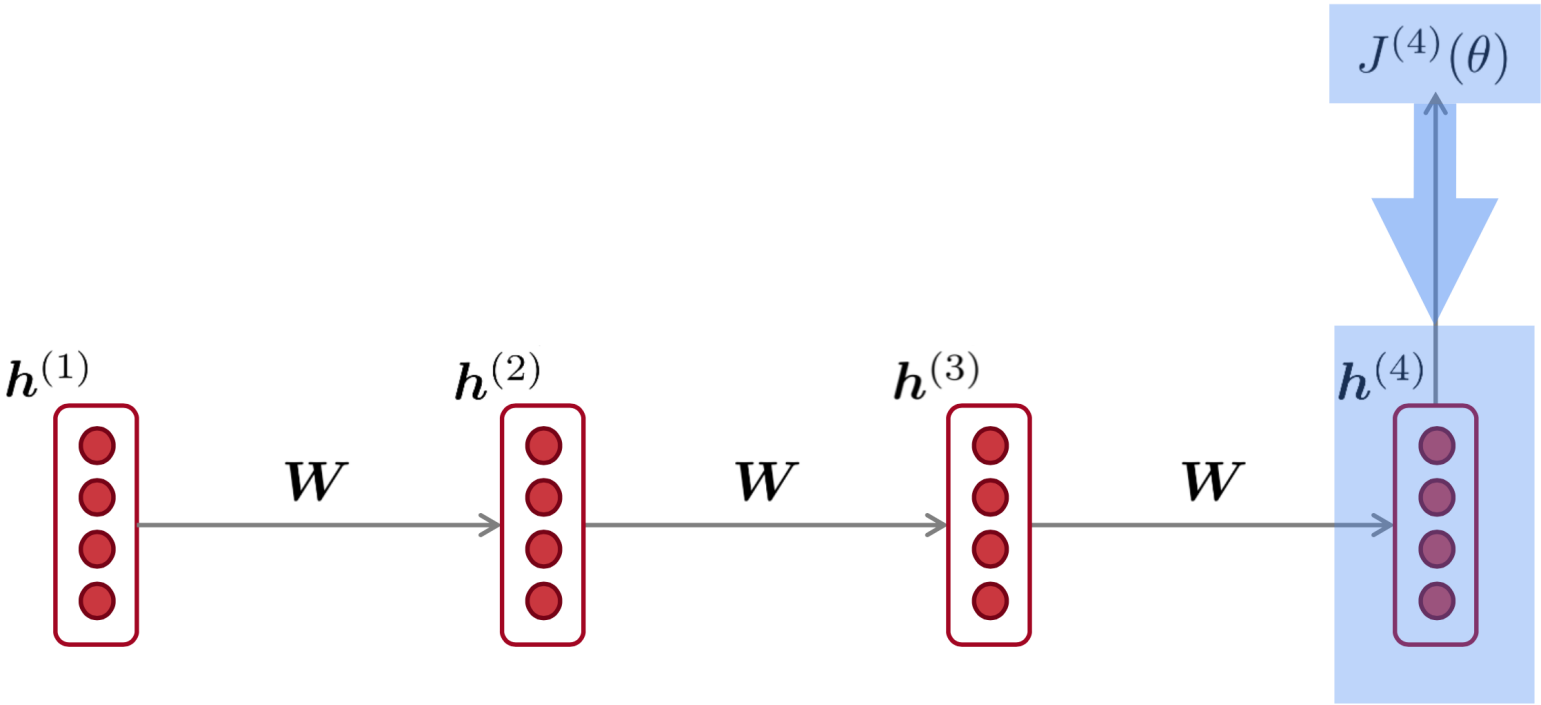
Vanishing gradient intuition



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

chain rule!

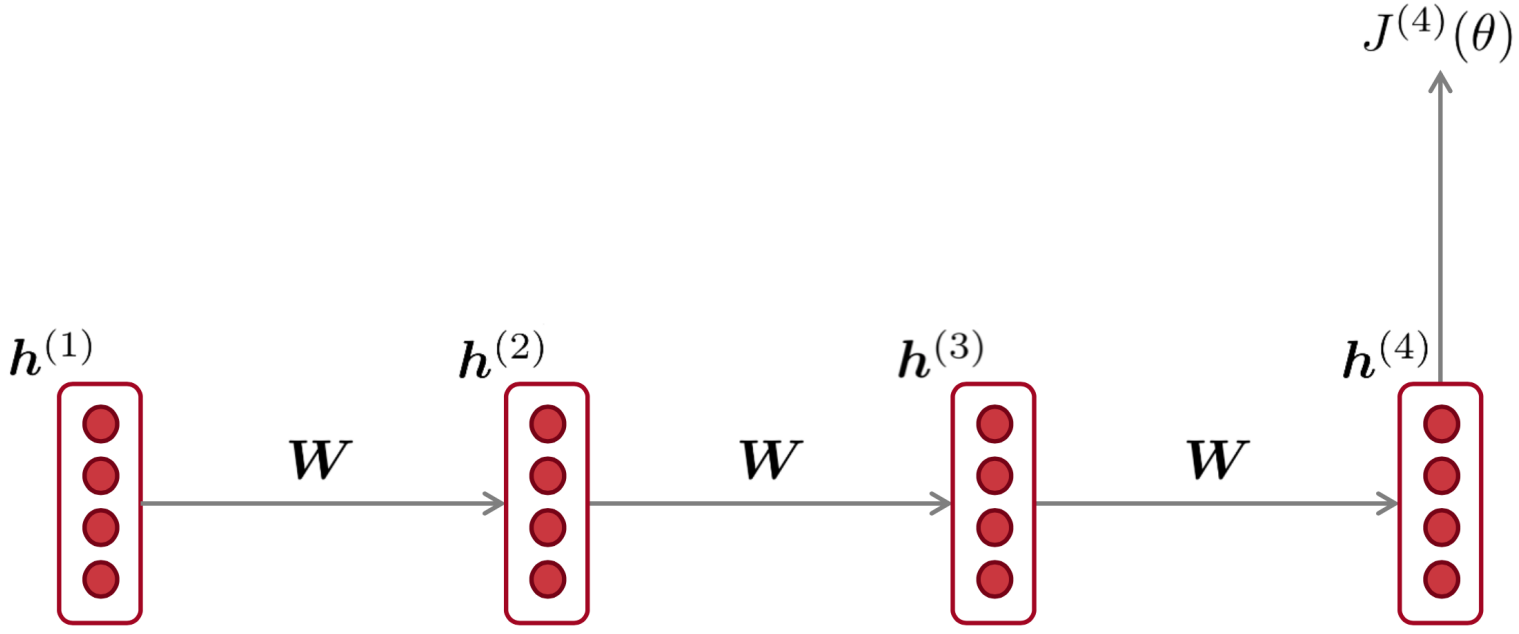
Vanishing gradient intuition



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

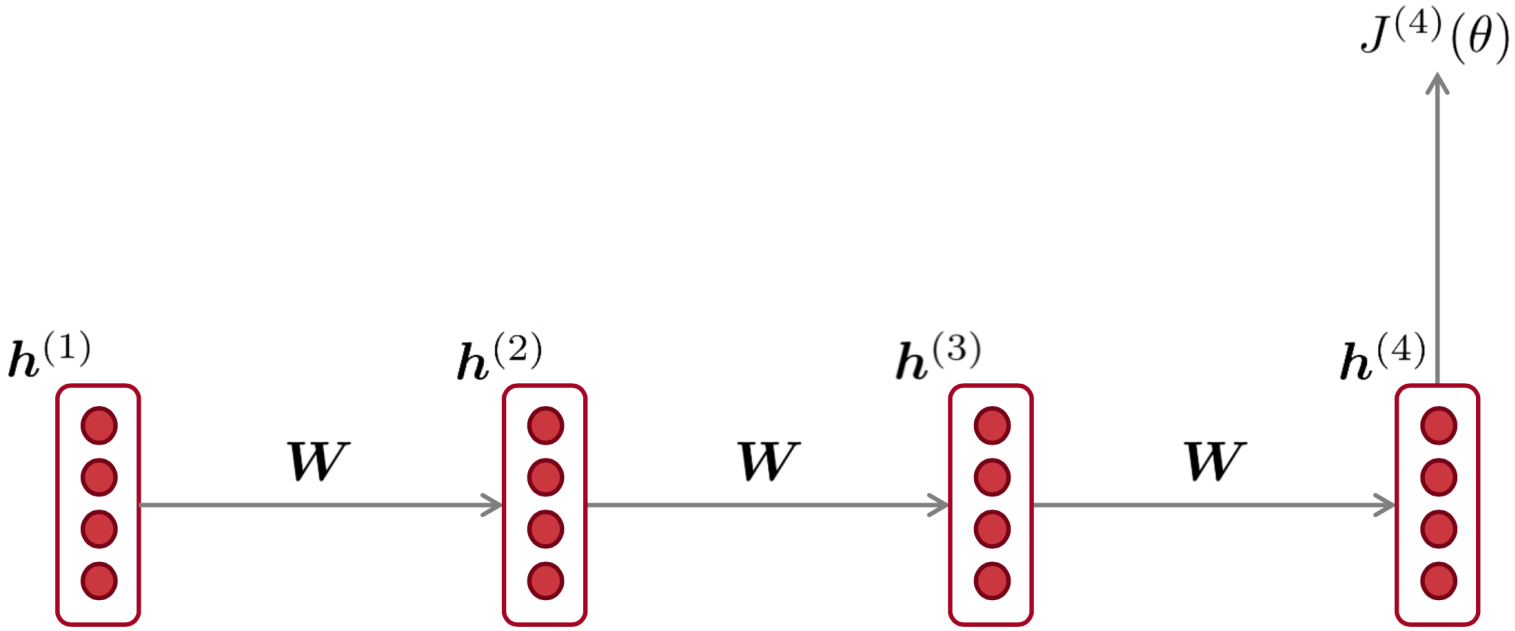
chain rule!

Vanishing gradient intuition



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

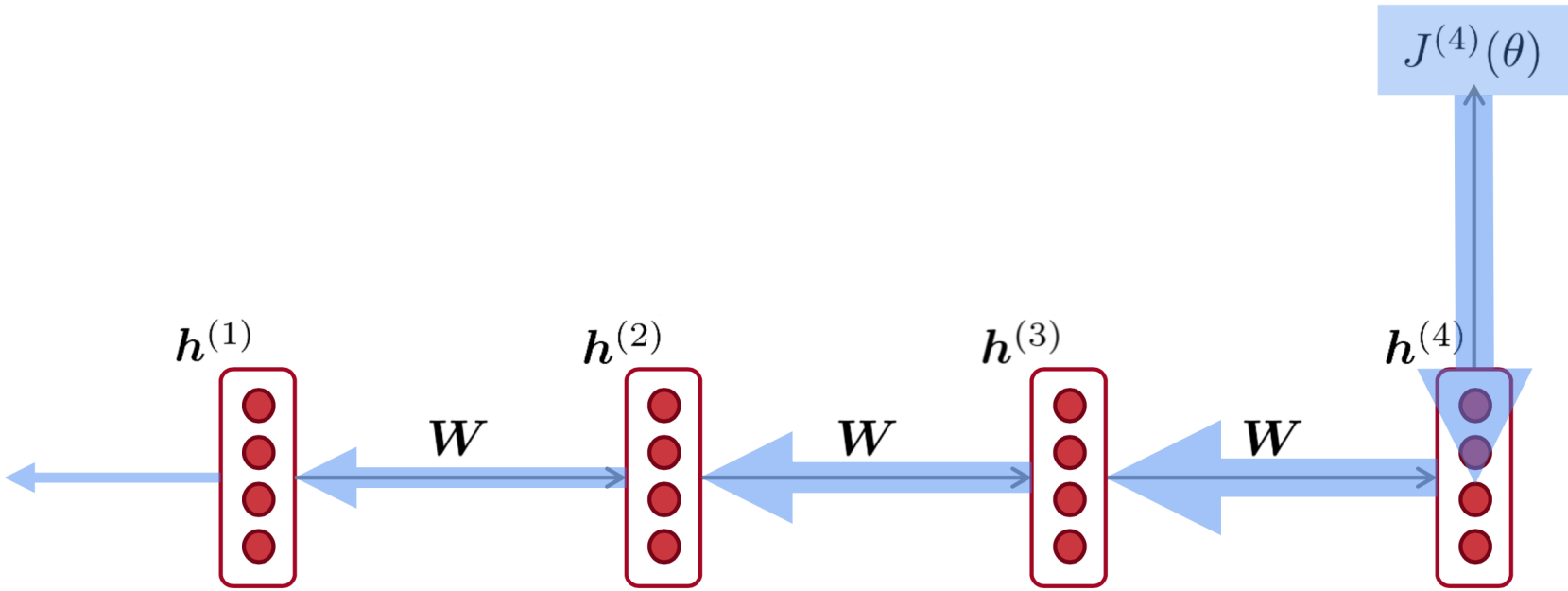
Vanishing gradient intuition



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

What happens if these are small?

Vanishing gradient intuition



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

What happens if these are small?

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

Vanishing gradient proof sketch

- Recall: $$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$
- Therefore: $$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h \qquad \text{(chain rule)}$$
- Consider the gradient of the loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $h^{(j)}$ on some previous step j.

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \qquad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \qquad \text{(value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \text{)}$$

If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013.

http://proceedings.mlr.press/v28/pascanu13.pdf

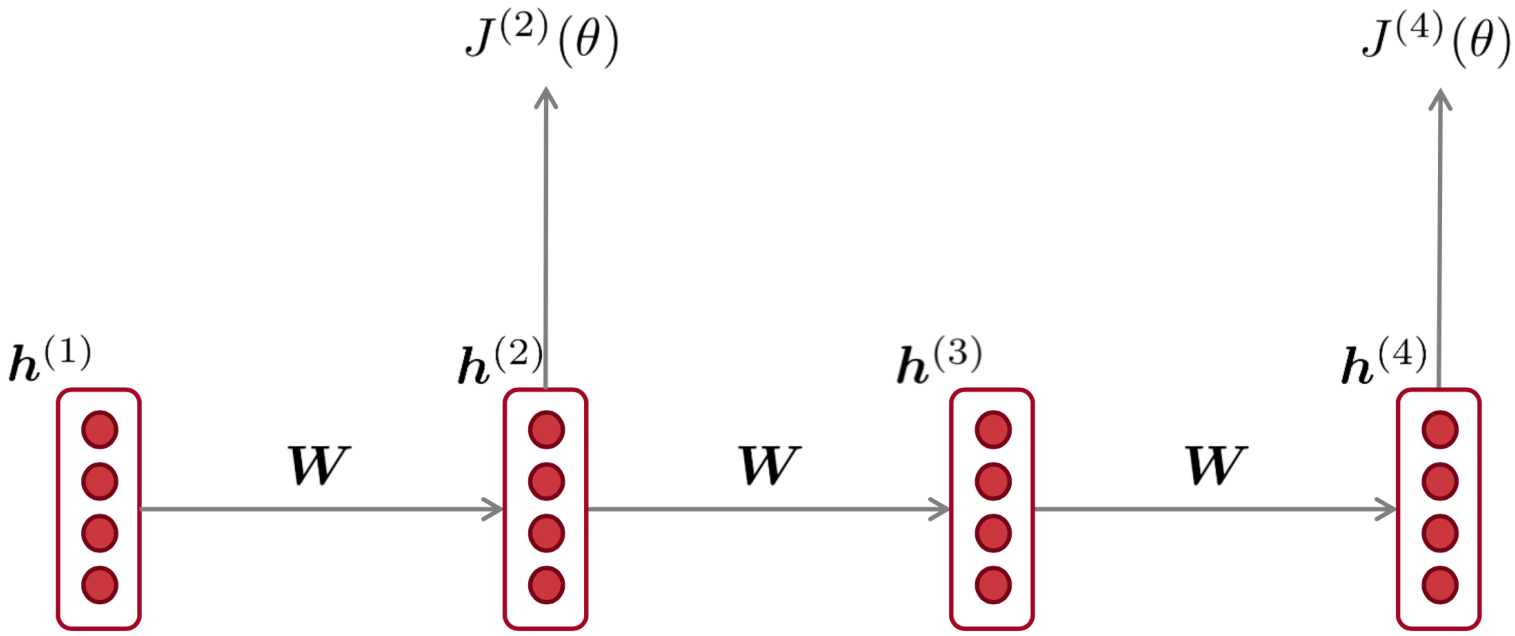
Vanishing gradient proof sketch

- Pascanu et al showed that if the largest eigenvalue of W_h is less than 1, then the gradient will shrink exponentially
  - Here the bound is 1 because we have sigmoid nonlinearity

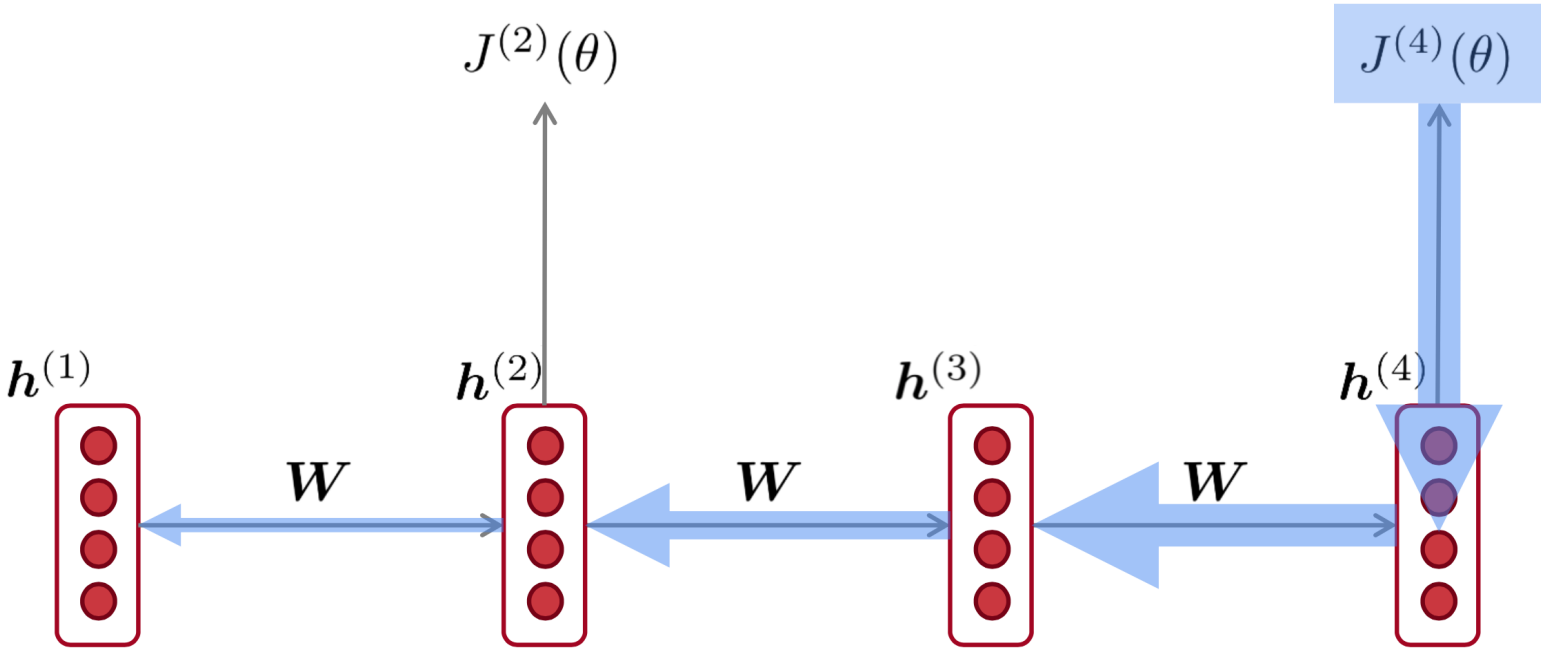
- There is a similar proof relating a largest eigenvalue >1 to exploding gradients

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

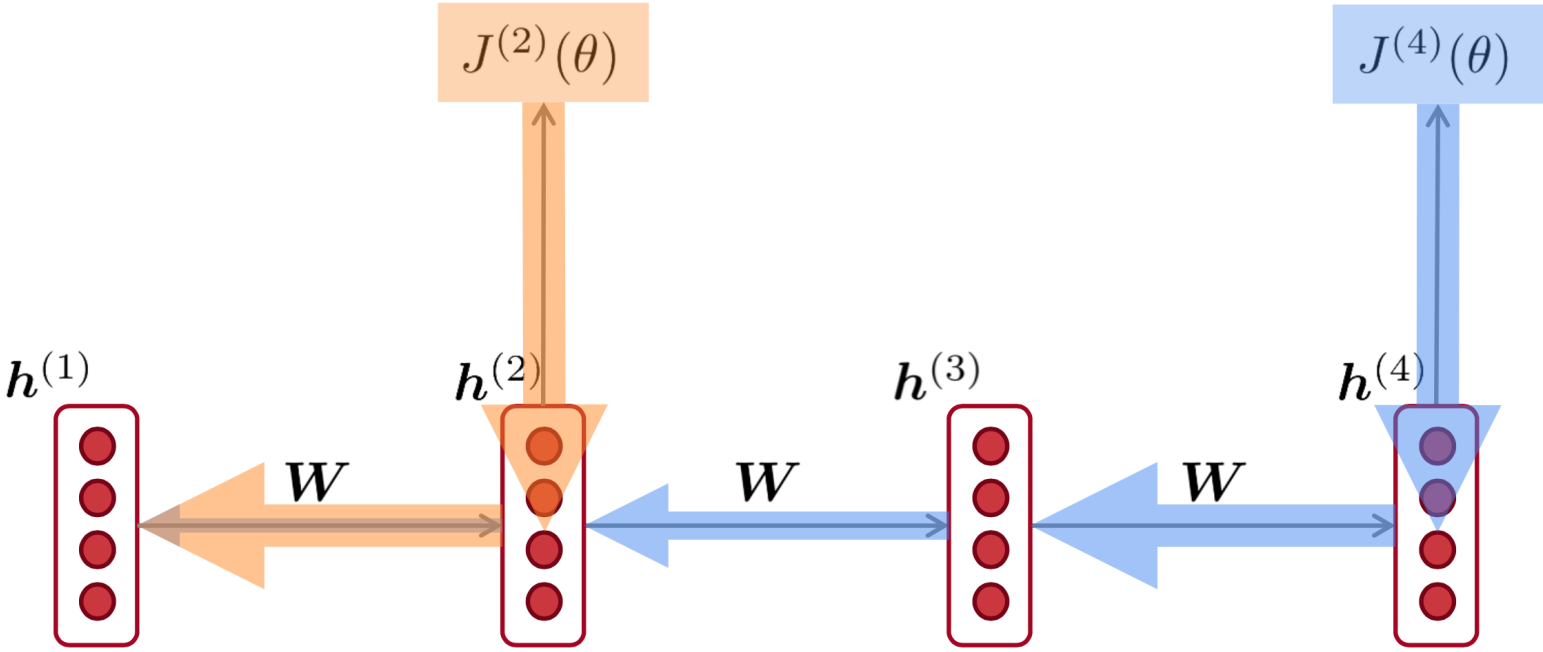
Why is vanishing gradient a problem?



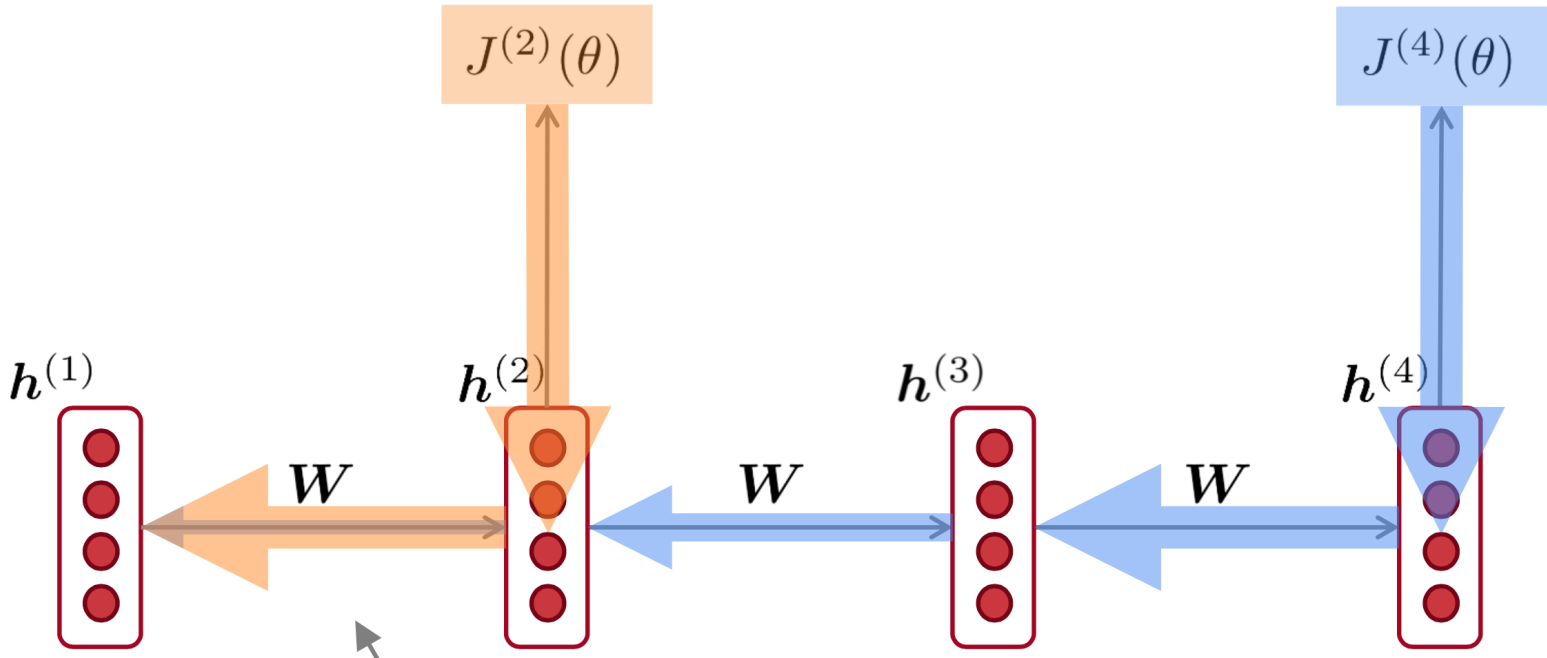
Why is vanishing gradient a problem?



Why is vanishing gradient a problem?



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.

Why is vanishing gradient a problem?

- 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 step $t+n$), then we cannot tell whether:
  - There is no dependency between step t and $t+n$ in the data.
  - We have wrong parameters to capture the true dependency between t and $t+n$.

How to fix vanishing gradient problem?

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

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

- How about an RNN with separate memory?

Long Short-Term Memory (LSTM)

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

- On step t, there is a hidden state $h^{(t)}$ and a cell state $c^{(t)}$
  - Both are vectors of 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 of 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", Hochreiter and Schmidhuber, 1997.

https://www.bioinf.jku.at/publications/older/2604.pdf

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:

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

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

Sigmoid function: all gate values are between 0 and 1

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

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

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

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

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

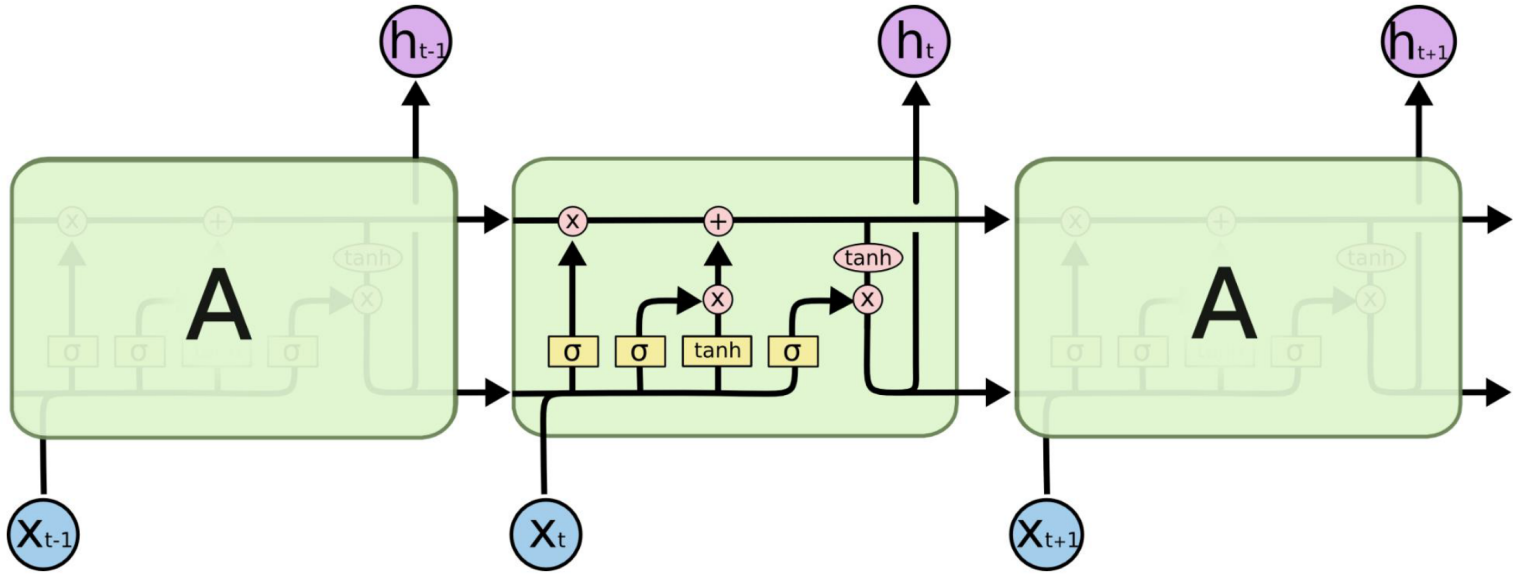
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

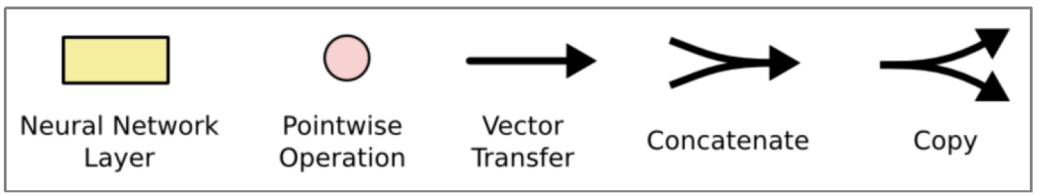
All these are vectors of same length n

Gates are applied using element-wise product

Long Short-Term Memory (LSTM)

LSTM equations visually:

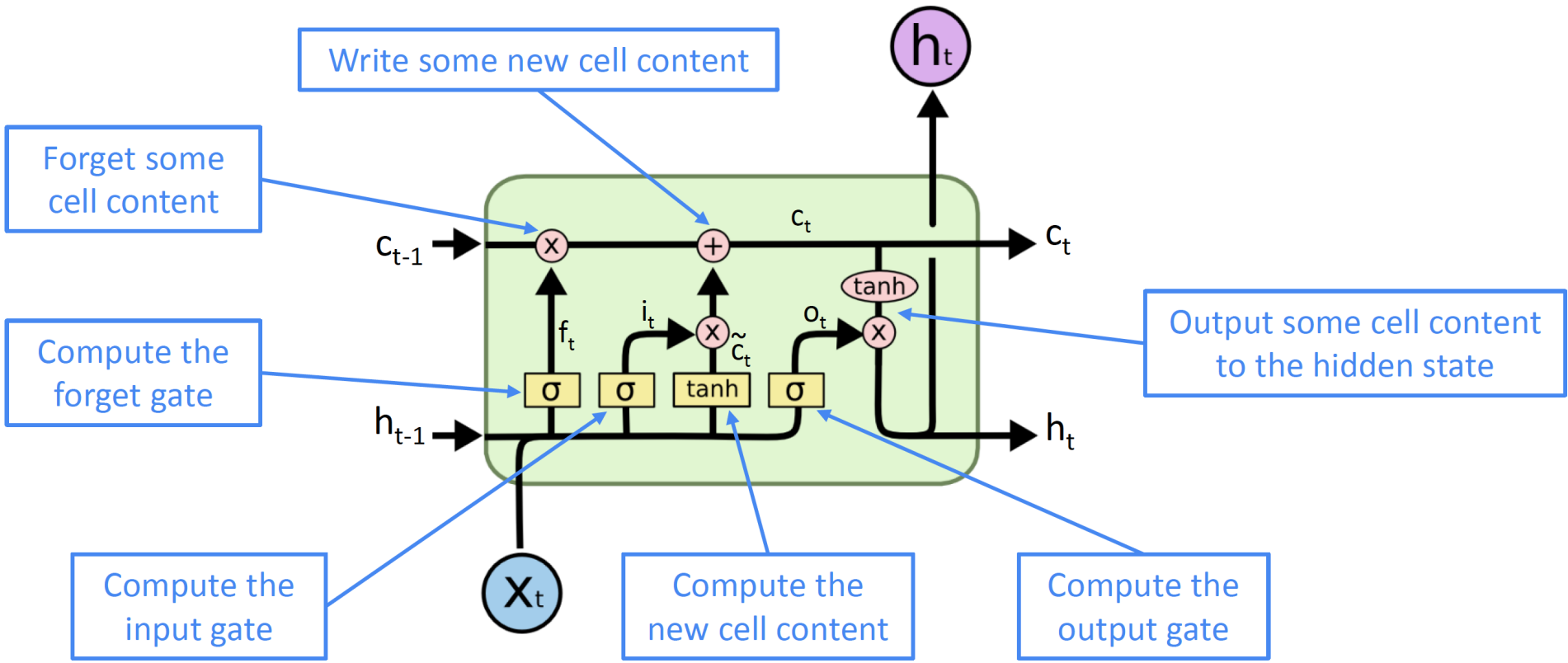




Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory (LSTM)

LSTM equations visually:



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

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 info in the cell is preserved indefinitely
  - By contrast, it is 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

LSTMs success

- In 2013-2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include: handwriting recognition, speech recognition, machine translation, parsing, image captioning
  - LSTM became the dominant approach
- Now (2019), other approaches (e.g. Transformers) have become more dominant for certain tasks.

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

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

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

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

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

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

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)

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

LSTM vs GRU

- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- The biggest difference is that GRU is quicker to compute and has fewer parameters
- There is no conclusive evidence that one consistently performs better than the other
- LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)
- Rule of thumb: start with LSTM, but switch to GRU if you want something more efficient

Is vanishing/exploding gradient just an RNN problem?

- No! It can be a problem for all neural architectures (including feed-forward and convolutional), especially deep ones.
  - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
  - Thus lower layers are learnt very slowly (hard to train)
  - Solution: lots of new deep feedforward/convolutional architectures that add more direct connections (thus allowing the gradient to flow)
- Conclusion: Though vanishing/exploding gradients are a general problem, RNNs are particularly unstable due to the repeated multiplication by the same weight matrix [Bengio et al, 1994]

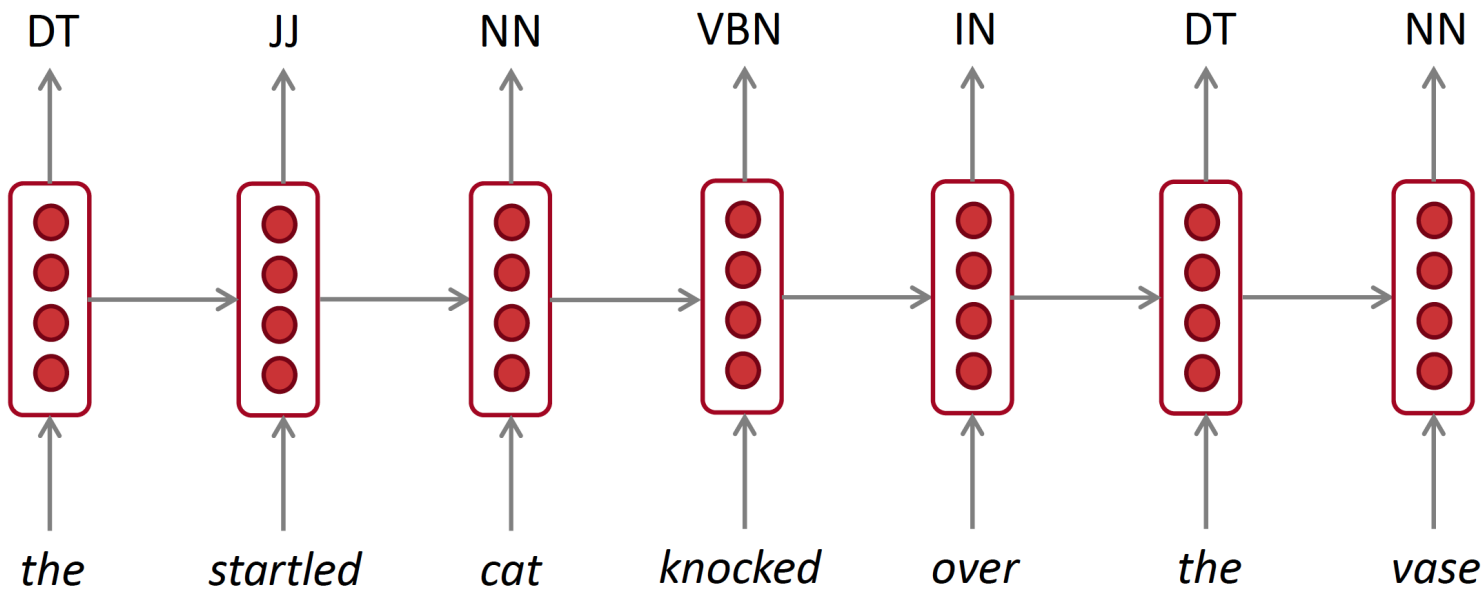
"Learning Long-Term Dependencies with Gradient Descent is Difficult", Bengio et al. 1994, http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf

Recap

- Language Model: A system that predicts the next word
- Recurrent Neural Network: A family of neural networks that:
  - Take sequential input of any length
  - Apply the same weights on each step
  - Can optionally produce output on each step
- Recurrent Neural Network $\neq$ Language Model
- We have shown that RNNs are a great way to build a LM.
- But RNNs are useful for much more!

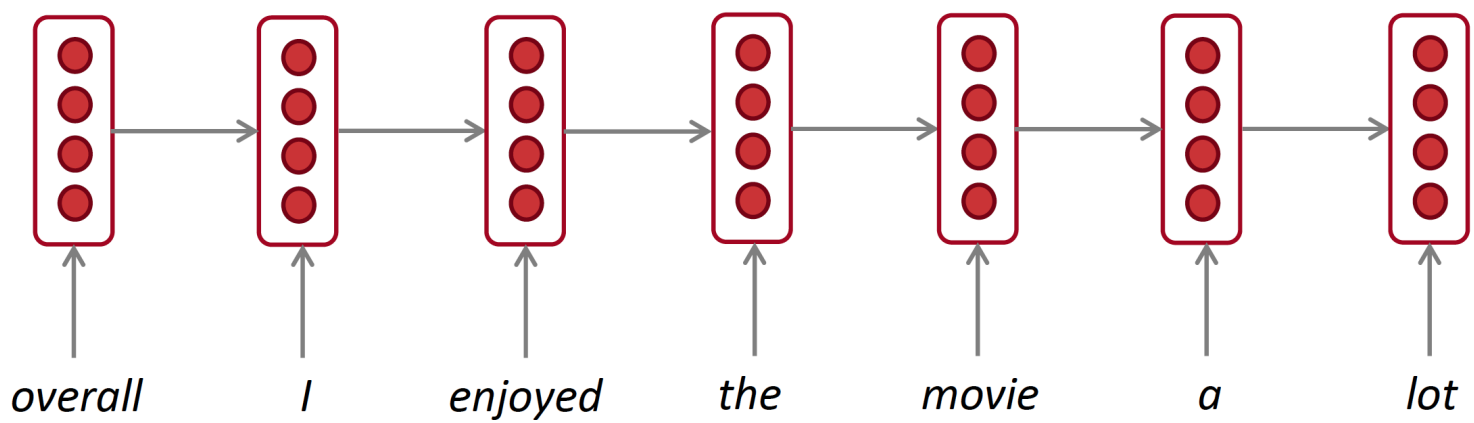
RNNs can be used for tagging

e.g. part-of-speech tagging, named entity recognition



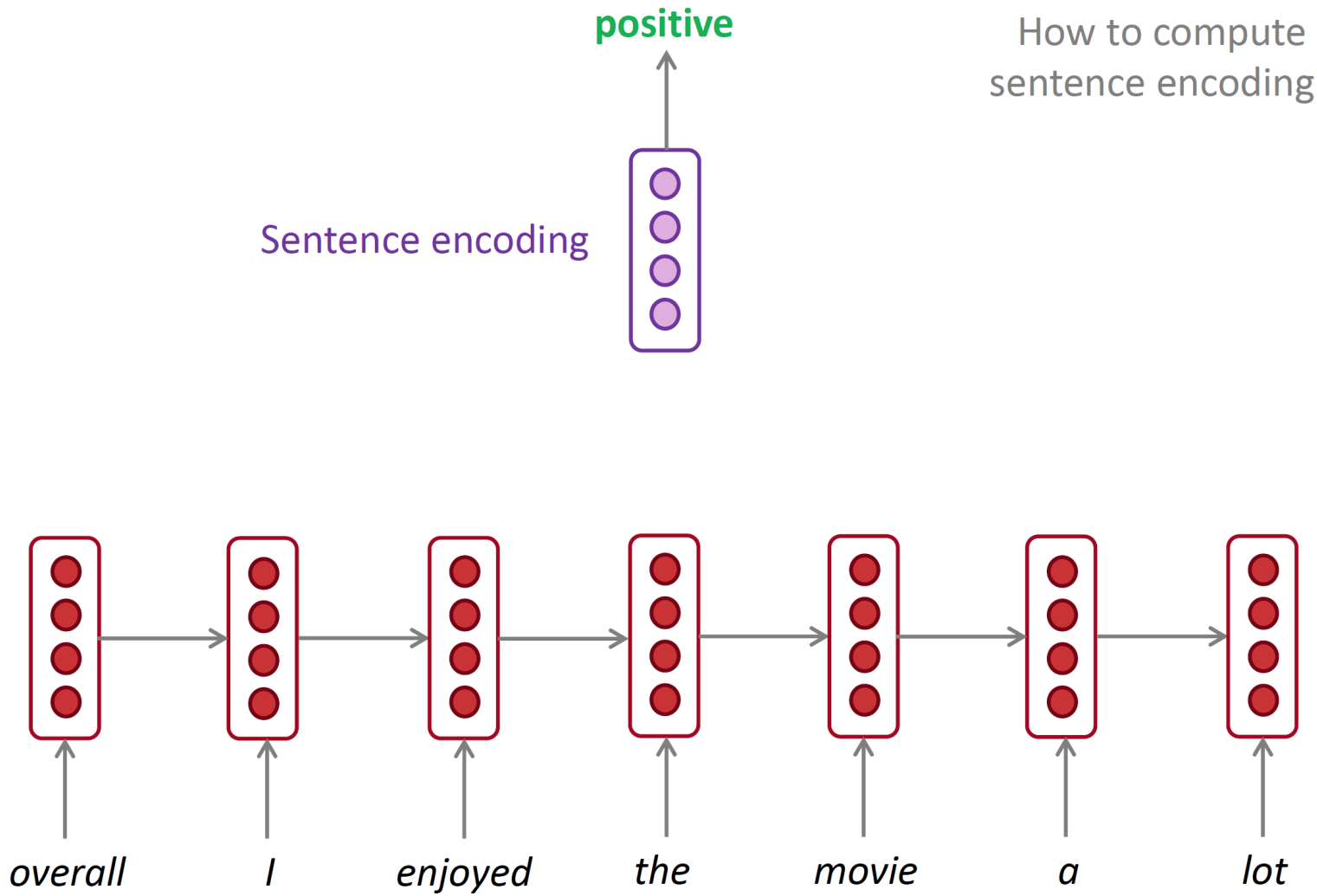
RNNs can be used for sentence classification

e.g. sentiment classification



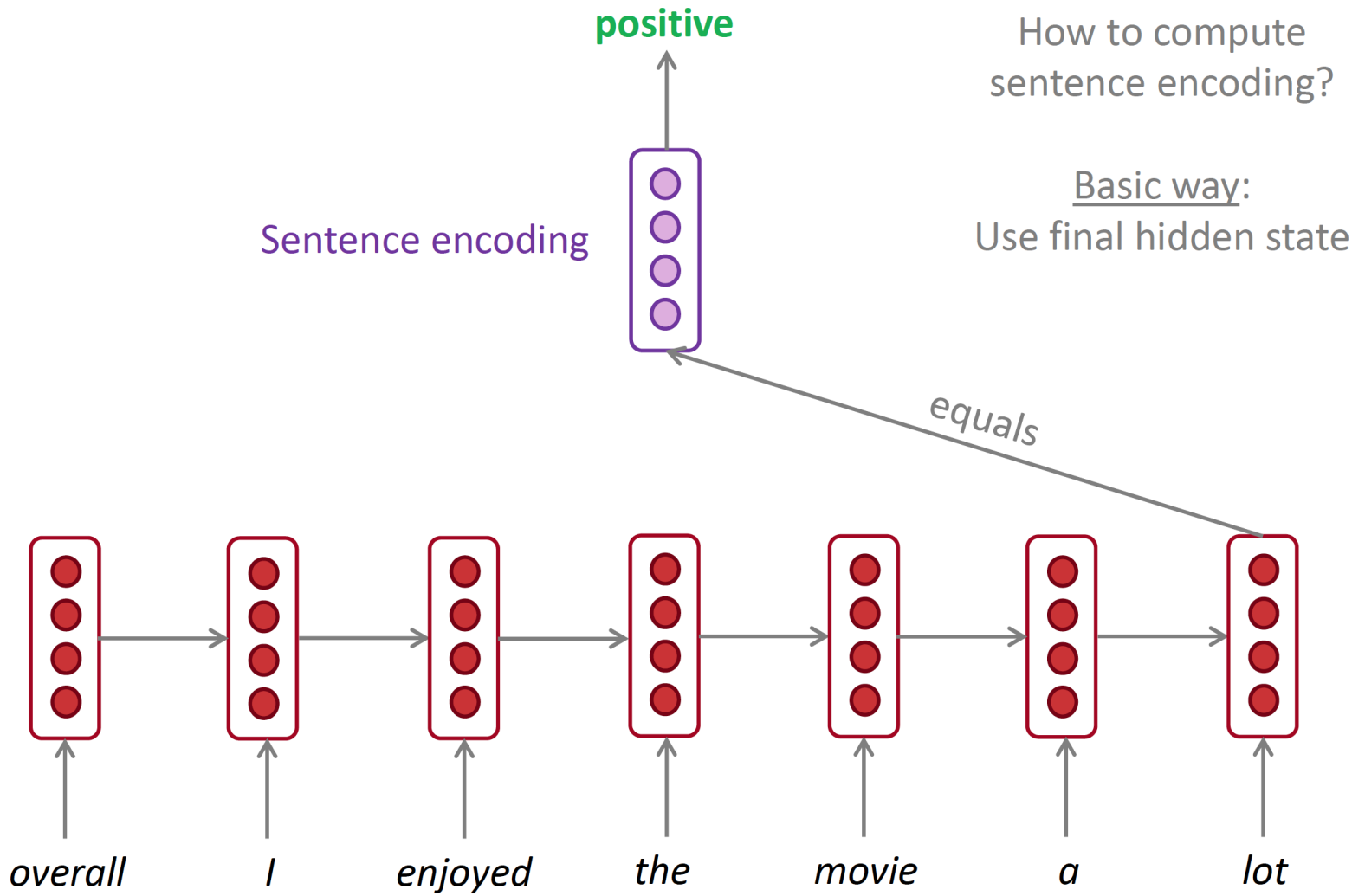
RNNs can be used for sentence classification

e.g. sentiment classification



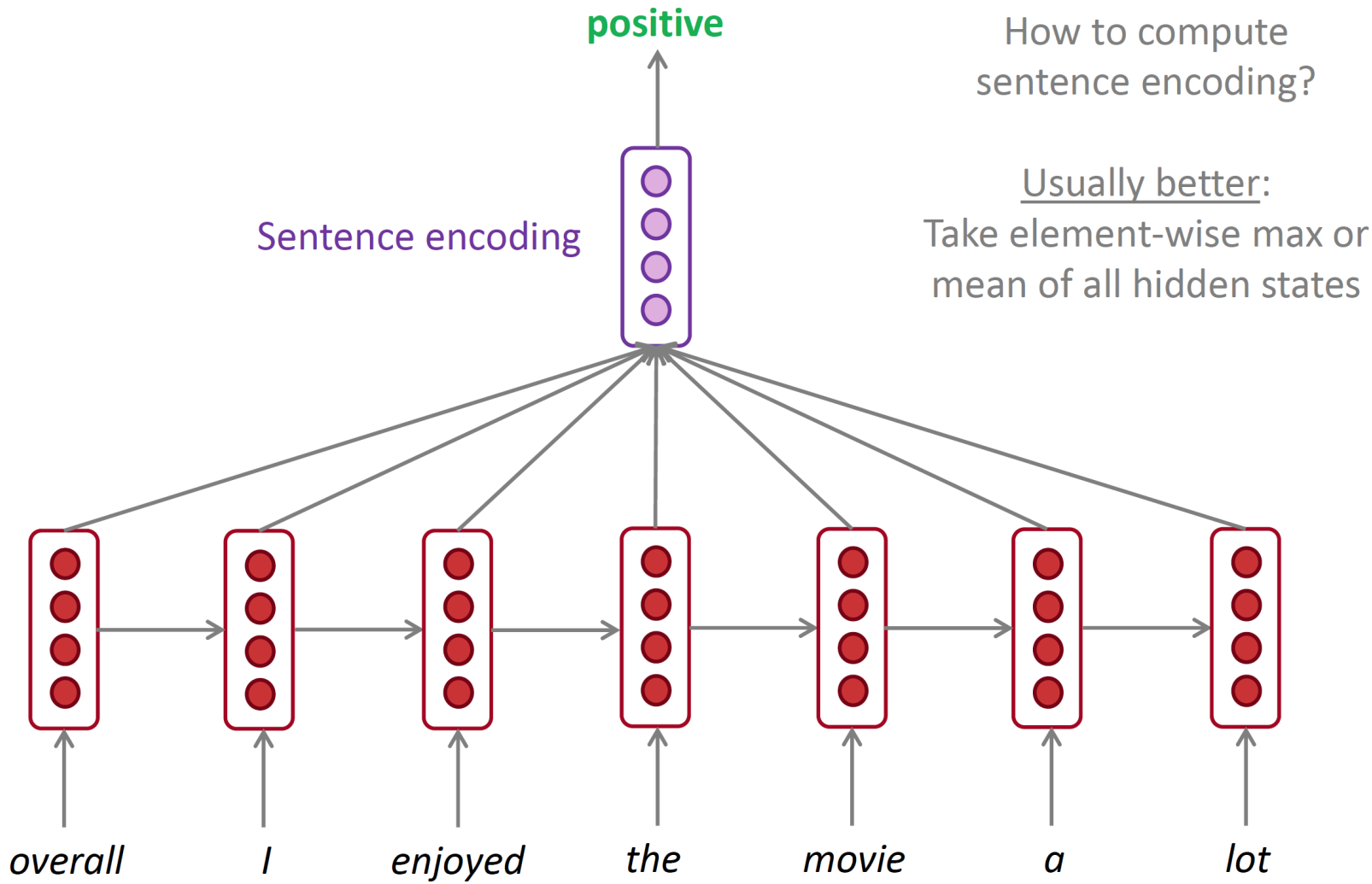
RNNs can be used for sentence classification

e.g. sentiment classification



RNNs can be used for sentence classification

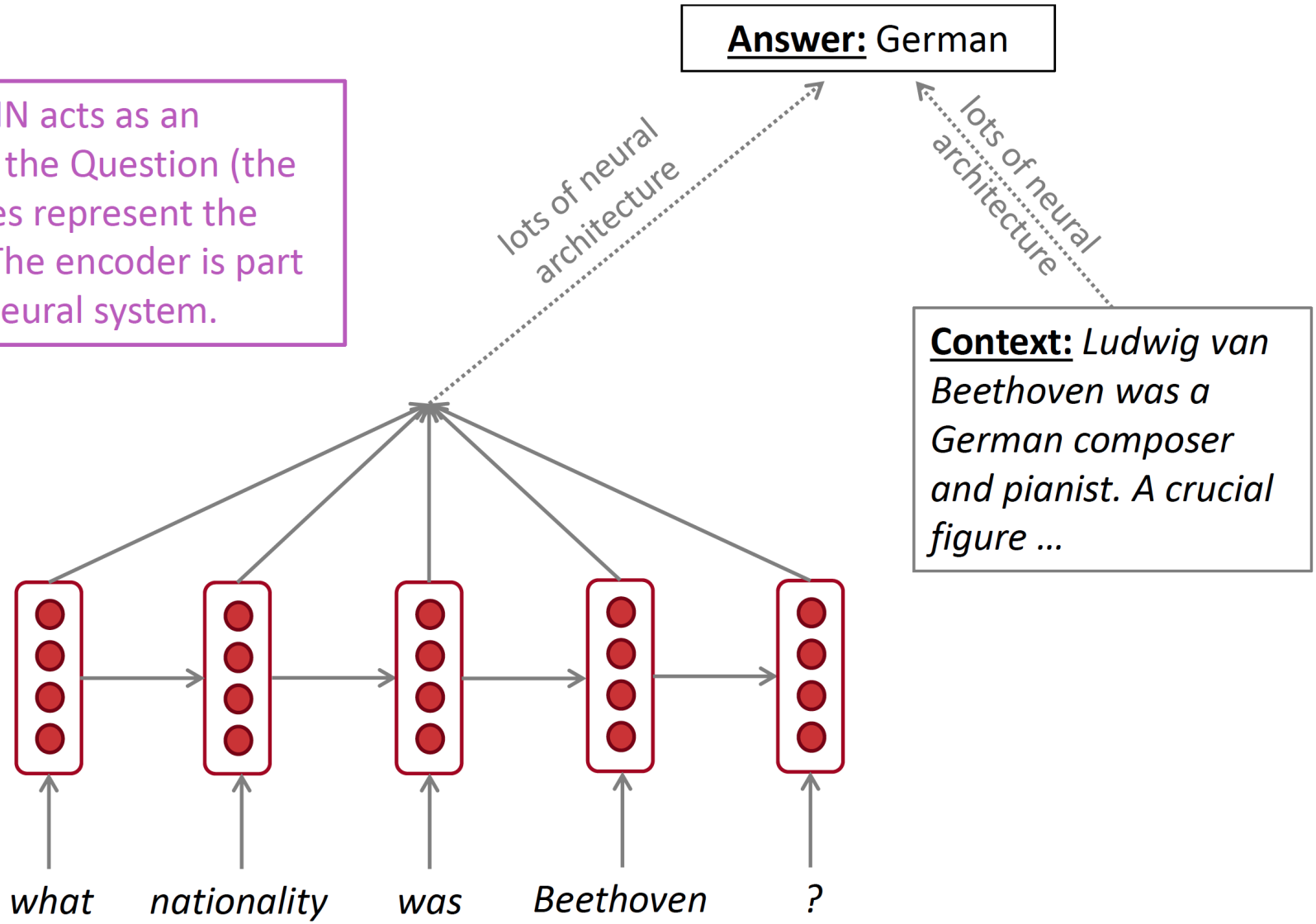
e.g. sentiment classification



RNNs can be used as an encoder module

e.g. question answering, machine translation, many other tasks!

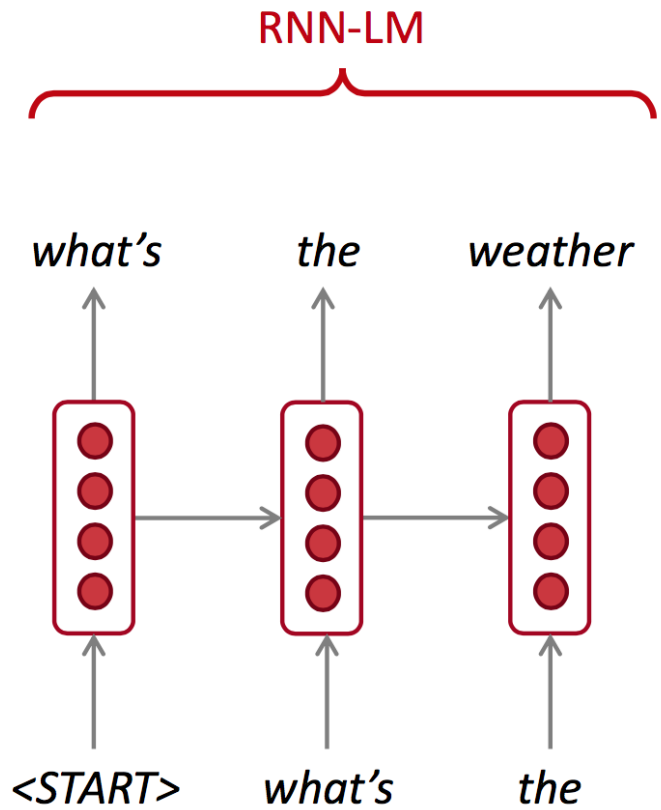
Here the RNN acts as an **encoder** for the Question (the hidden states represent the Question). The encoder is part of a larger neural system.



RNN-LMs can be used to generate text

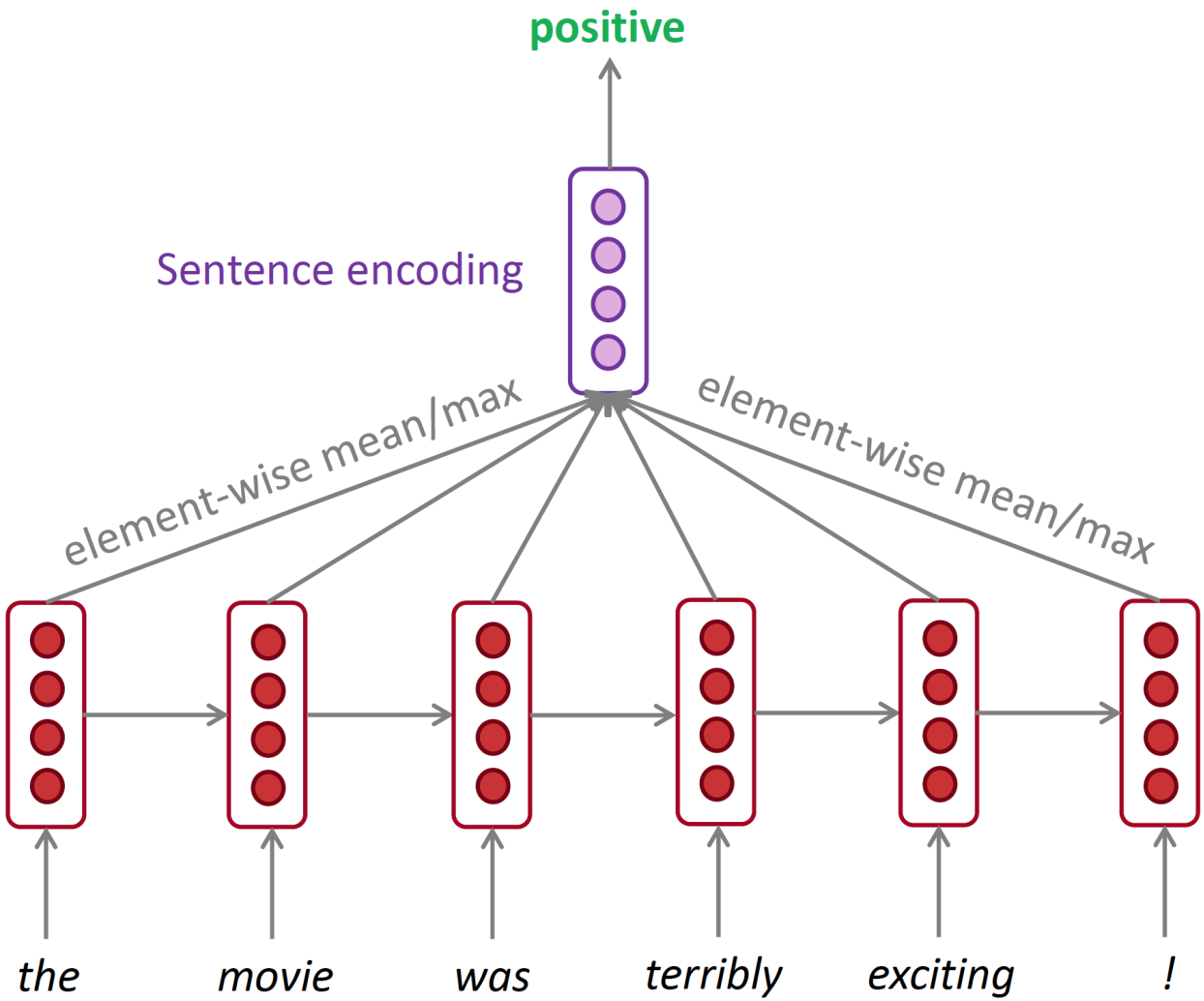
e.g. speech recognition, machine translation, summarization

Input (audio)



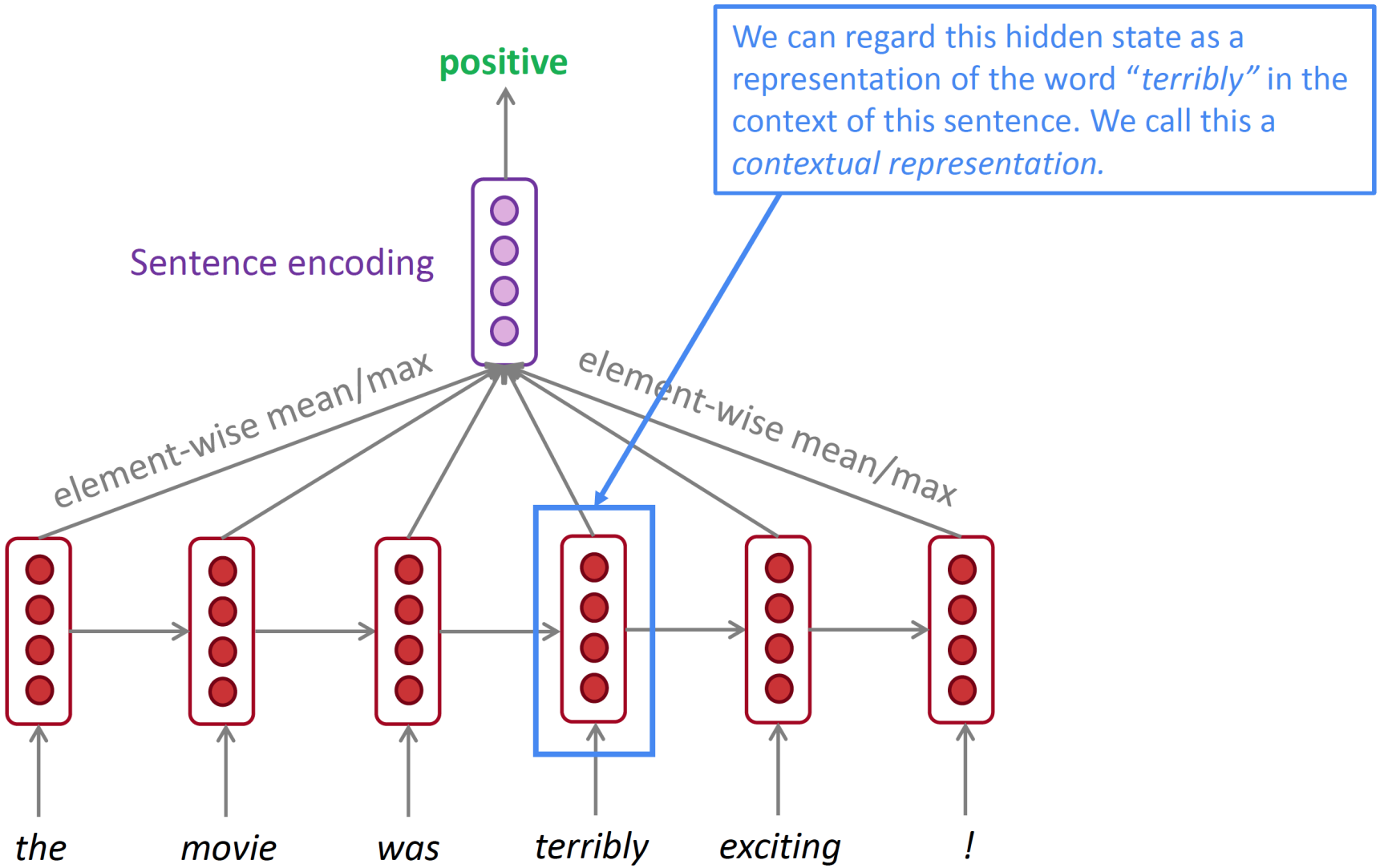
Limitation of Unidirectional RNNs

Task: Sentiment Classification



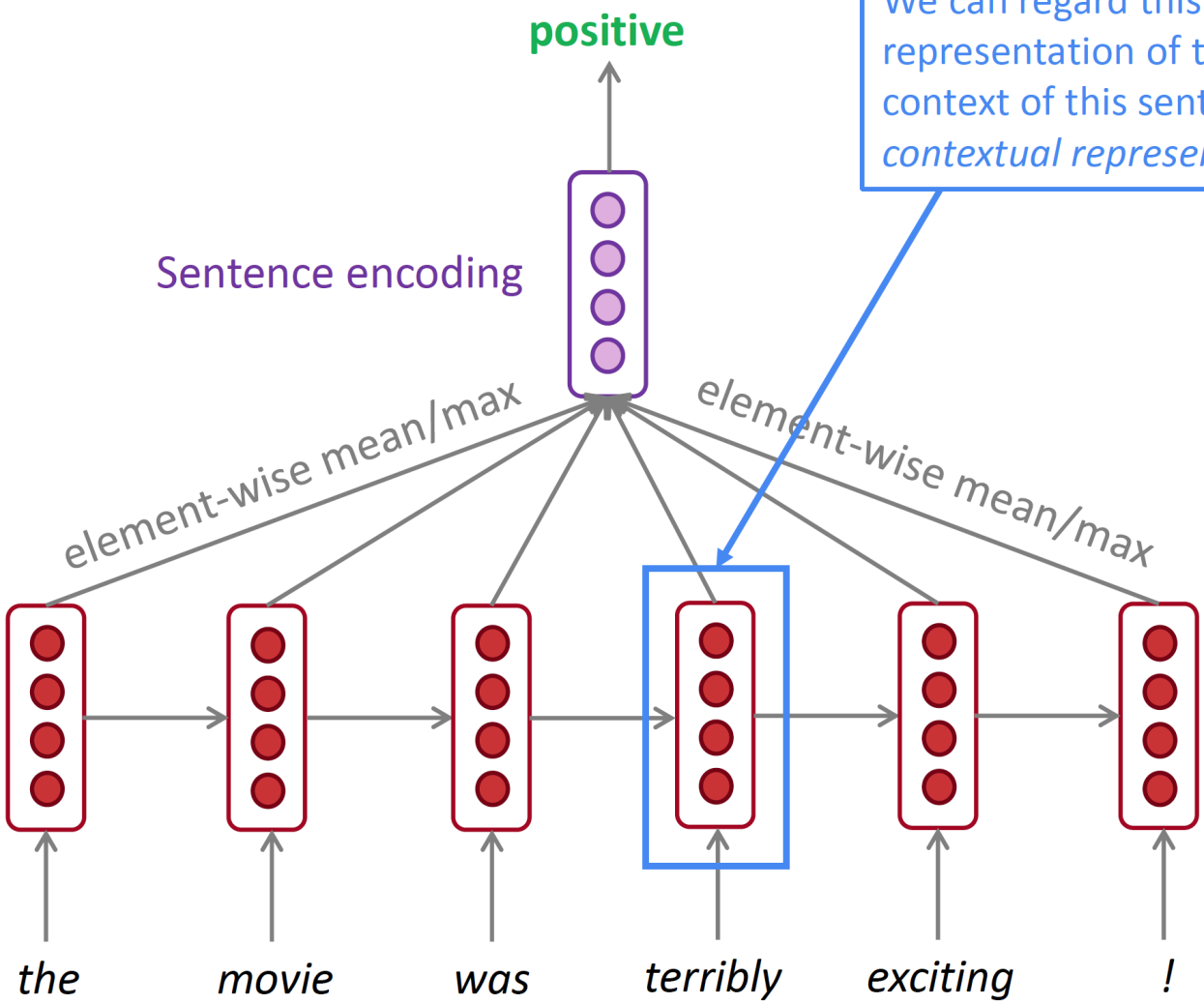
Limitation of Unidirectional RNNs

Task: Sentiment Classification



Limitation of Unidirectional RNNs

Task: Sentiment Classification



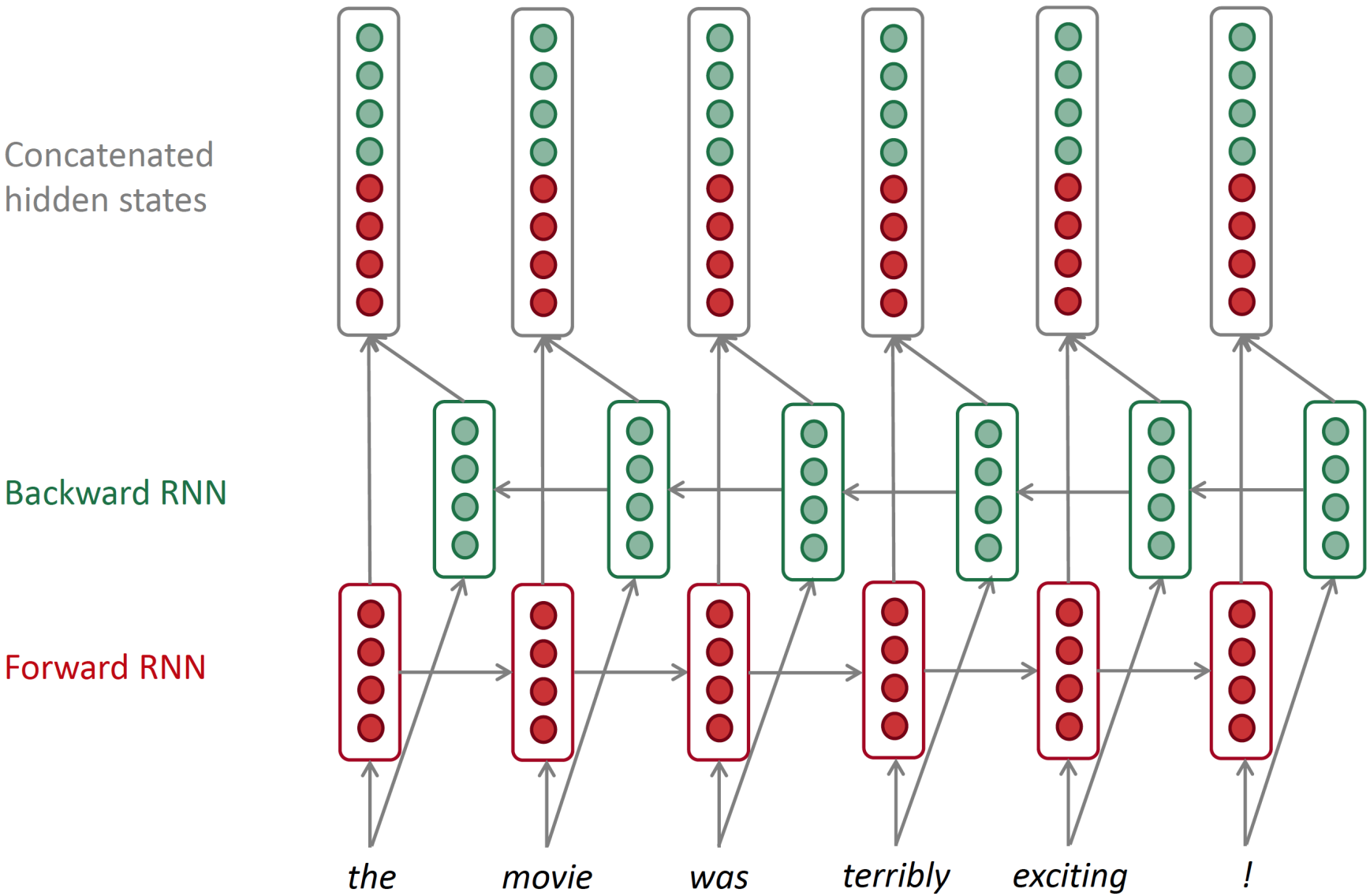
We can regard this hidden state as a representation of the word *"terribly"* in the context of this sentence. We call this a *contextual representation.*

These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

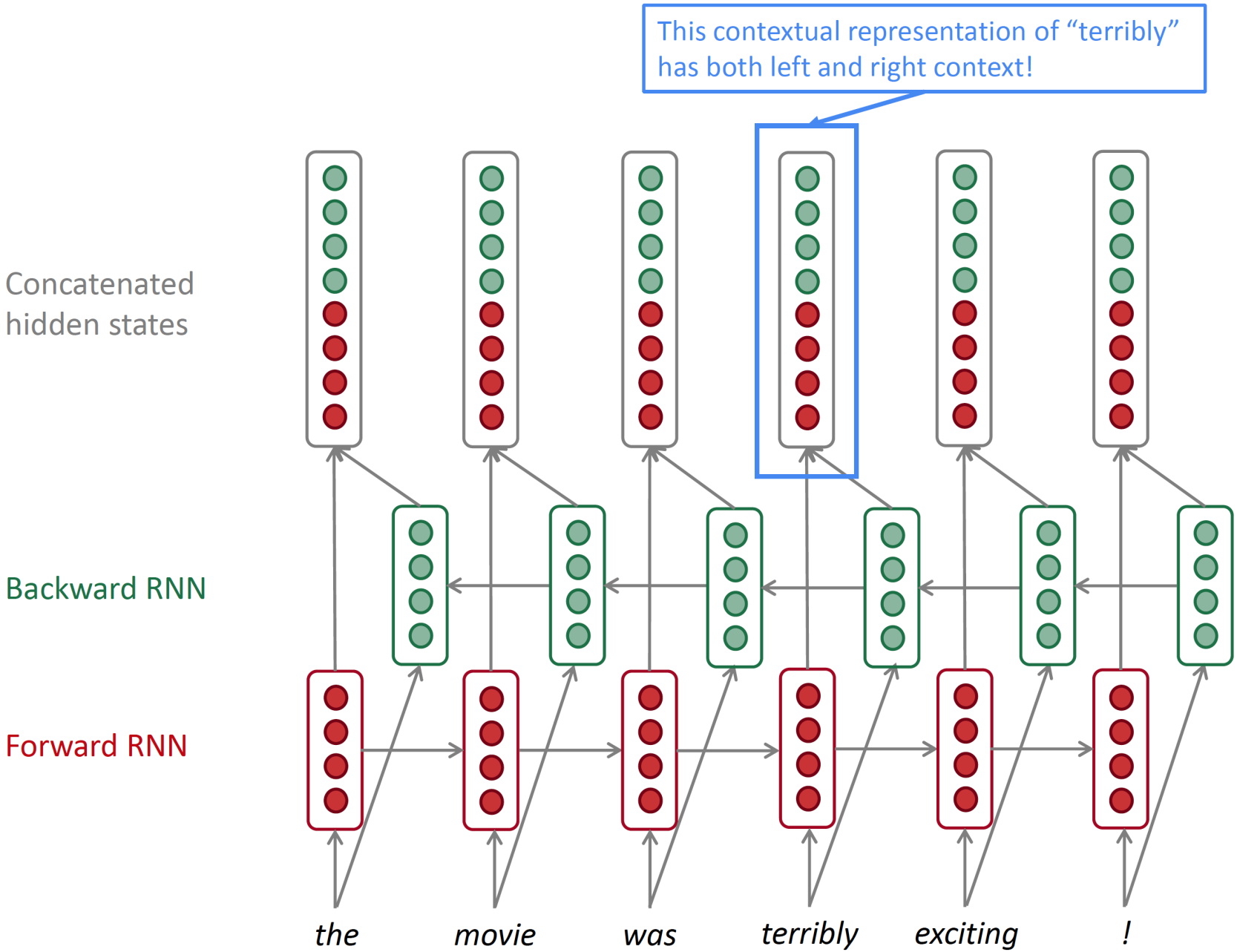
What about *right* context?

In this example, *"exciting"* is in the right context and this modifies the meaning of *"terribly"* (from negative to positive)

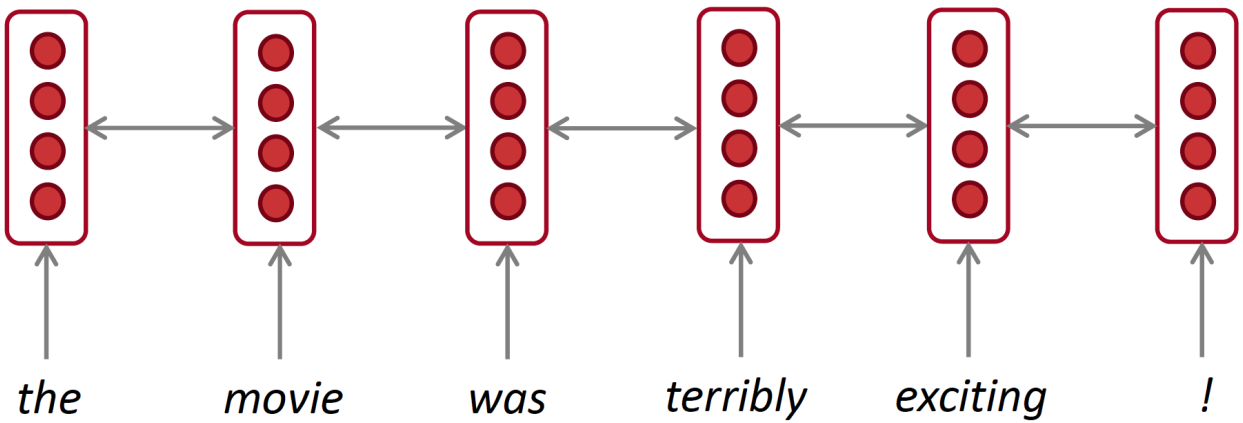
Bidirectional RNNs



Bidirectional RNNs



Bidirectional RNNs: simplified diagram



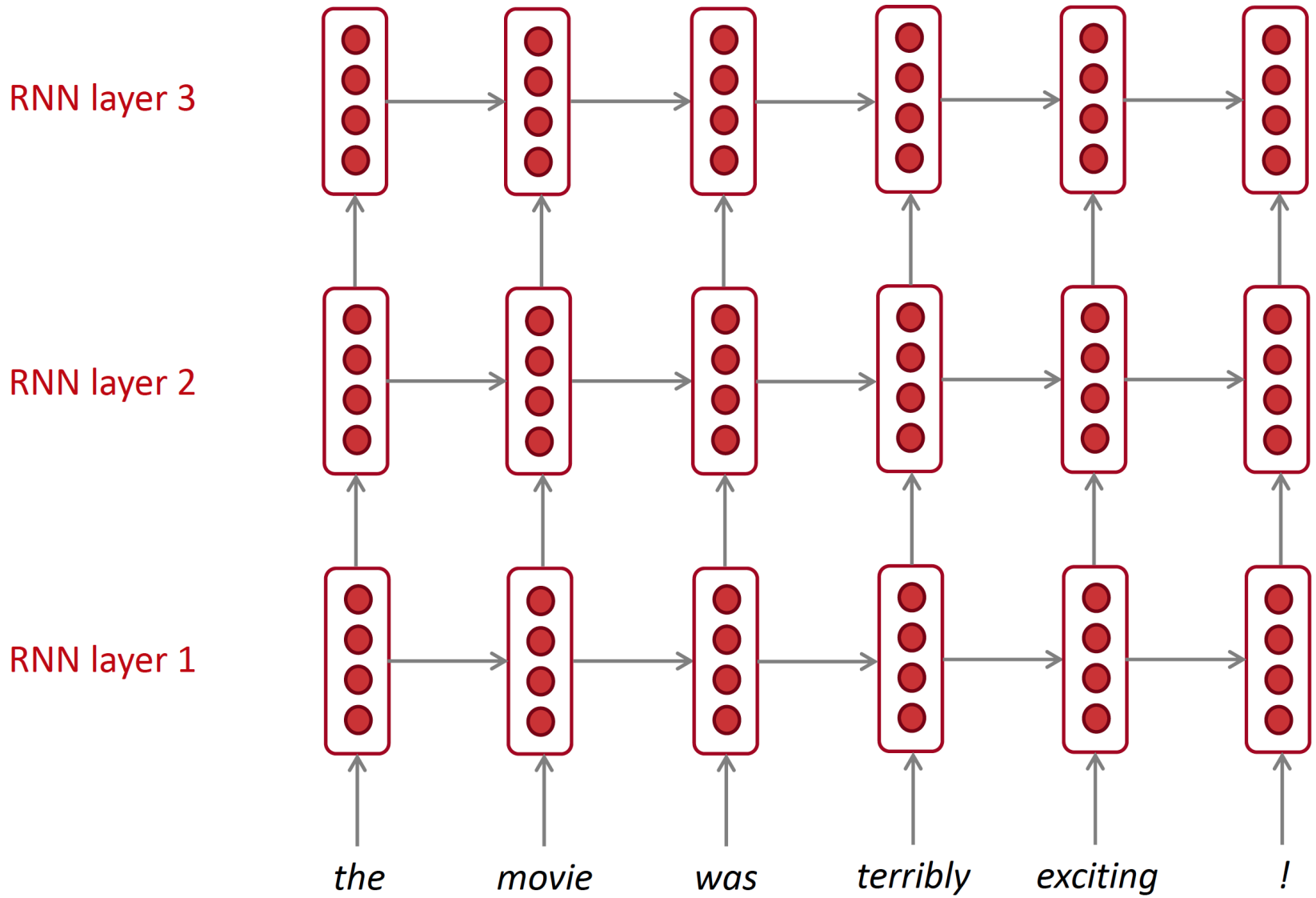
The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

Multi-layer RNNs

- RNNs are already "deep" on one dimension (they unroll over many timesteps)
- We can also make them "deep" in another dimension by applying multiple RNNs - this is a multi-layer RNN.
- This 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 RNNs are also called stacked RNNs.

Multi-layer RNNs

The hidden states from RNN layer i are the inputs to RNN layer $i+1$.



RNNs for sequence modeling

- Classify each word into:
  - NER
  - Entity level sentiment in context
  - Opinionated expressions
- Example application and slides from paper
  Opinion Mining with Deep Recurrent Nets by Irsoy and Cardie 2014.
  https://www.cs.cornell.edu/ ∼oirsoy/files/emnlp14drnt.pdf

Opinion Mining with Deep Recurrent Nets

- Goal: Classify each word in a sentence as:
  direct subjective expressions (DSEs) and
  expressive subjective expressions (ESEs).

  DSE: Explicit mentions of private states or speech events expressing private states

  ESE: Expressions that indicate sentiment, emotion, etc. without explicitly conveying them.

Example Annotation

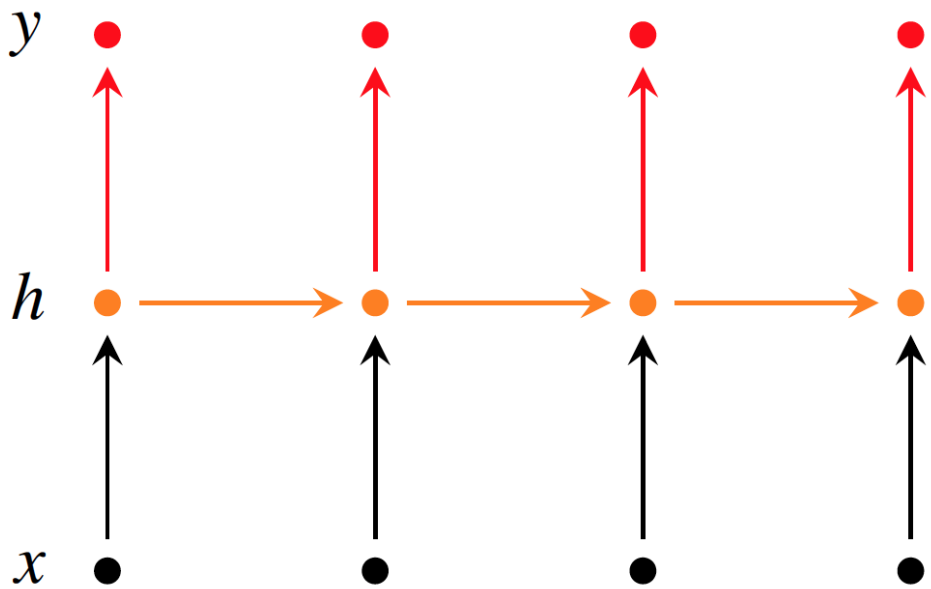
In BIO notation (tags either begin-of-entity (B_X) or continuation-of-entity (I_X)):

The committee, [as usual]$_{ESE}$, [has refused to make any statements]$_{DSE}$.

The	committee	,	as	usual	,	has
O	O	O	B_ESE	I_ESE	O	B_DSE

refused	to	make	any	statements	.
I_DSE	I_DSE	I_DSE	I_DSE	I_DSE	O

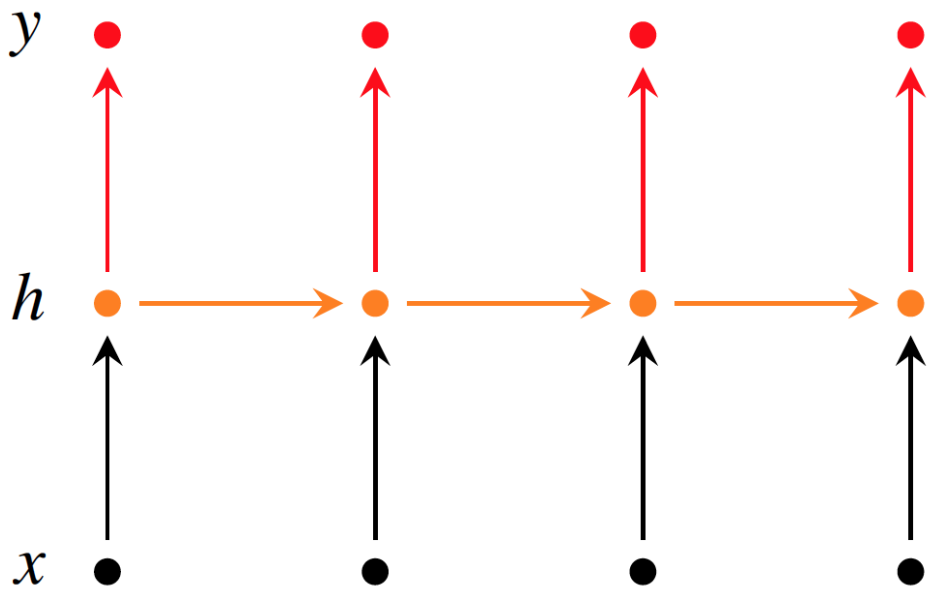
Approach: Recurrent Neural Network



$$h_t = f(Wx_t + Vh_{t-1} + b)$$
$$y_t = g(Uh_t + c)$$

- x represents a token (word) as a vector.
- y represents the output label (B, I or O) - g = softmax !
- h is the memory, computed from the past memory and current word. It summarizes the sentence up to that time.

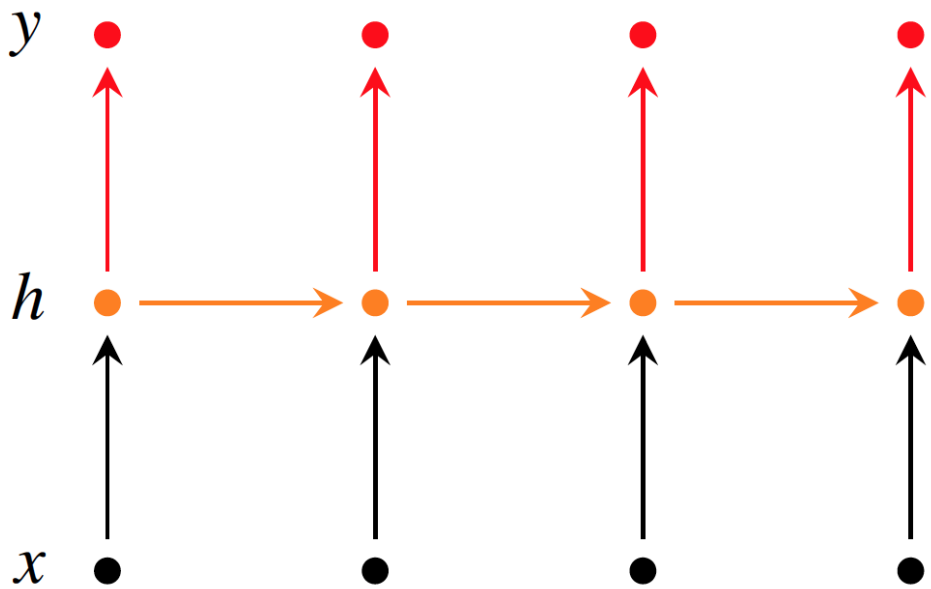
Approach: Recurrent Neural Network



$$h_t = f(Wx_t + Vh_{t-1} + b)$$
$$y_t = g(Uh_t + c)$$

- W and V are weight matrices between the input and hidden layer, and among the hidden units themselves (connecting the previous intermediate representation to the current one), respectively. U is the output weight matrix. b and c are bias vectors connected to hidden and output units, respectively.
- h_0 is assumed to be 0.

Approach: Recurrent Neural Network



$$h_t = f(Wx_t + Vh_{t-1} + b)$$
$$y_t = g(Uh_t + c)$$

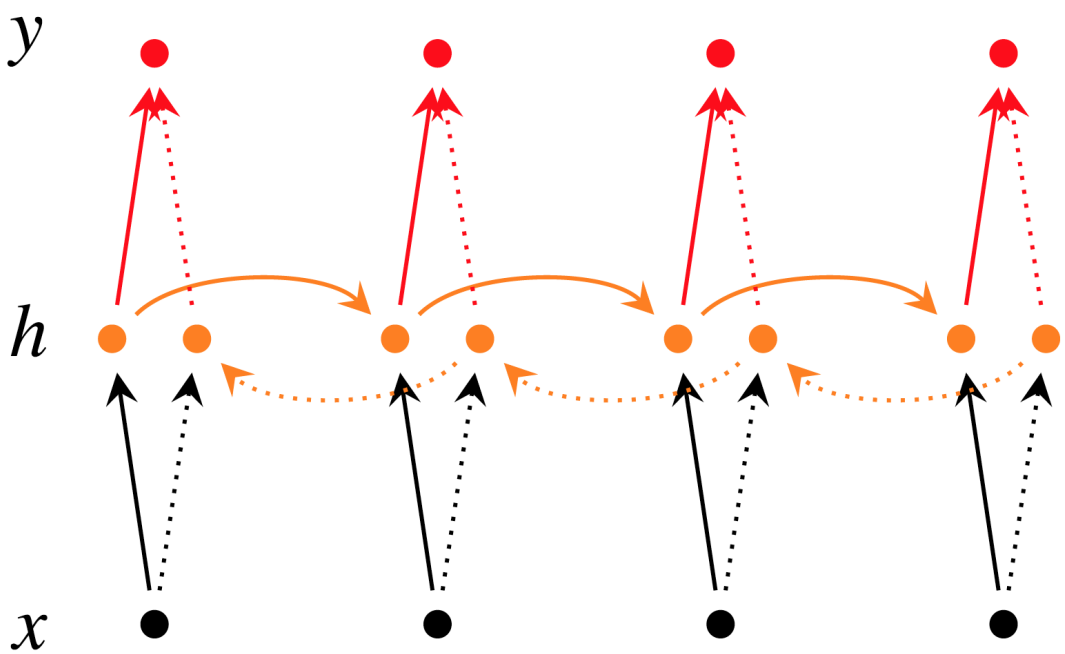
- Training an RNN can be done by optimizing a discriminative objective (e.g. the cross entropy for classification tasks) with a gradient-based method. Backpropagation through time can be used to efficiently compute the gradients. This method is essentially equivalent to unfolding the network in time and using backprop as in feedforward neural networks, while sharing the connection weights across different time steps.

Limitation of RNNs

- We have information only about the past, when making a decision on x_t, which is limiting for most NLP tasks.
- Example:
  - Consider the two sentences: "I did not accept his suggestion" and "I did not go to the rodeo".
  - The first has a DSE phrase ("did not accept") and the second does not.
  - However, any such RNN will assign the same labels for the words "did" and "not" in both sentences, since the preceding sequences (past) are the same.
  - The unidirectional RNNs lack the representational power to model this task.

Bidirectional RNNs

Problem: For classification you want to incorporate information from words both preceding and following



$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

$h = [\vec{h}; \overleftarrow{h}]$ now represents (summarizes) the past and future around a single token.

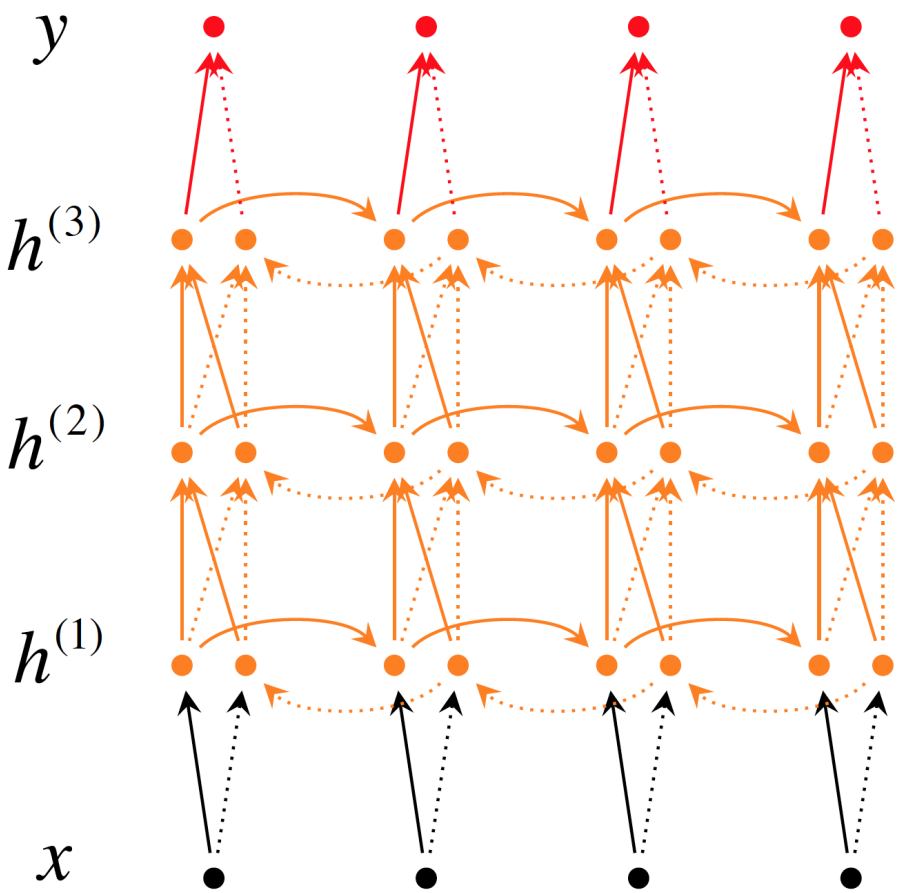
Bidirectional RNNs

- RNNs are often characterized as having depth in time: when unfolded, they are equivalent to feedforward neural networks with as many hidden layers as the number tokens in the input sequence (with shared connections across multiple layers of time).
- However, this notion of depth likely does not involve hierarchical processing of the data: across different time steps, we repeatedly apply the same transformation to compute the memory contribution of the input (W), to compute the response value from the current memory (U) and to compute the next memory vector from the previous one (V).
- Therefore, assuming the input vectors x_t together lie in the same representation space, as do the output vectors y_t, hidden representations h_t lie in the same space as well.

Bidirectional RNNs

- These representations do not necessarily become more and more abstract, hierarchical representations of one another as we traverse in time.
- However, in the more conventional, stacked deep learners (e.g. deep feedforward nets), an important benefit of depth is the hierarchy among hidden representations: every hidden layer conceptually lies in a different representation space, and constitutes a more abstract and higher-level representation of the input (Bengio, 2009).
- Thus, the authors investigated deep RNNs, which are constructed by stacking RNNs on top of each other (Hermans and Schrauwen, 2013).
- Intuitively, every layer of the deep RNN treats the memory sequence of the previous layer as the input sequence, and computes its own memory representation.

Deep Bidirectional RNNs



$$\vec{h}_t^{(i)} = f(\vec{W}^{(i)} h_t^{(i-1)} + \vec{V}^{(i)} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Each memory layer passes an intermediate sequential representation to the next.

Deep RNNs: Experimental Setup

Activation Units. We employ the standard softmax activation for the output layer: $g(x) = e^{x_i} / \sum_j e^{x_j}$. For the hidden layers we use the rectifier linear activation: $f(x) = \max\{0, x\}$. Experimentally, rectifier activation gives better performance, faster convergence, and sparse representations. Previous work also reported good results when training deep neural networks using rectifiers, without a pretraining step (Glorot et al., 2011).

Data. We use the MPQA 1.2 corpus (Wiebe et al., 2005) (535 news articles, 11,111 sentences) that is manually annotated with both DSEs and ESEs at the phrase level. As in previous work, we separate 135 documents as a development set and employ 10-fold CV over the remaining 400 documents. The development set is used during cross validation to do model selection.

Evaluation Metrics. We use precision, recall and F-measure for performance evaluation. Since the boundaries of expressions are hard to define even for human annotators (Wiebe et al., 2005), we use two soft notions of the measures: *Binary Overlap* counts every overlapping match between a predicted and true expression as correct (Breck et al., 2007; Yang and Cardie, 2012), and *Proportional Overlap* imparts a partial correctness, proportional to the overlapping amount, to each match (Johansson and Moschitti, 2010; Yang and Cardie, 2012). All statistical comparisons are done using a two-sided paired t-test with a confidence level of $\alpha = .05$.

Deep RNNs: Experimental Setup

Network Training. We use the standard multiclass cross-entropy as the objective function when training the neural networks. We use stochastic gradient descent with momentum with a fixed learning rate (.005) and a fixed momentum rate (.7). We update weights after minibatches of 80 sentences. We run 200 epochs for training. Weights are initialized from small random uniform noise. We experiment with networks of various sizes, however we have the same number of hidden units across multiple forward and backward hidden layers of a single RNN. We do not employ a pre-training step; deep architectures are trained with the supervised error signal, even though the output layer is connected to only the final hidden layer. With these configurations, every architecture successfully converges without any oscillatory behavior. Additionally, we employ early stopping for the neural networks: out of all iterations, the model with the best development set performance (Proportional F1) is selected as the final model to be evaluated.

Results: Bidirectional vs. Unidirectional RNNs

- The bidirectional RNN obtains higher F1 scores than the unidirectional RNN - 63.83 vs. 60.35 (proportional overlap) and 69.62 vs. 68.31 (binary overlap) for DSEs; 54.22 vs. 51.51 (proportional) and 65.44 vs. 63.65 (binary) for ESEs.
- All differences are statistically significant at the 0.05 level.

Results: Adding Depth

Layers	$\|h\|$	Precision		Recall		F1	
		Prop.	Bin.	Prop.	Bin.	Prop	Bin.
Shallow	36	62.24	65.90	65.63*	73.89*	63.83	69.62
Deep 2	29	63.85*	67.23*	65.70*	**74.23***	64.70*	70.52*
Deep 3	25	63.53*	67.67*	65.95*	73.87*	64.57*	70.55*
Deep 4	22	**64.19***	**68.05***	**66.01***	73.76*	**64.96***	**70.69***
Deep 5	21	60.65	61.67	56.83	69.01	58.60	65.06
Shallow	200	62.78	66.28	65.66*	74.00*	64.09	69.85
Deep 2	125	62.92*	66.71*	66.45*	**74.70***	64.47	70.36
Deep 3	100	**65.56***	**69.12***	**66.73***	74.69*	**66.01***	**71.72***
Deep 4	86	61.76	65.64	63.52	72.88*	62.56	69.01
Deep 5	77	61.64	64.90	62.37	72.10	61.93	68.25

Table 2: Experimental evaluation of RNNs for DSE extraction

Layers	$\|h\|$	Precision		Recall		F1	
		Prop.	Bin.	Prop.	Bin.	Prop	Bin.
Shallow	36	51.34	59.54	57.60	72.89*	54.22	65.44
Deep 2	29	51.13	59.94	**61.20***	**75.37***	**55.63***	**66.64***
Deep 3	25	**53.14***	**61.46***	58.01	72.50	55.40*	66.36*
Deep 4	22	51.48	60.59*	59.25*	73.22	54.94	66.15*
Deep 5	21	49.67	58.42	48.98	65.36	49.25	61.61
Shallow	200	**52.20***	60.42*	58.11	72.64	54.75	65.75
Deep 2	125	51.75*	60.75*	60.69*	74.39*	55.77*	66.79*
Deep 3	100	52.04*	**60.50***	**61.71***	**76.02***	**56.26***	**67.18***
Deep 4	86	50.62*	58.41*	53.55	69.99	51.98	63.60
Deep 5	77	49.90*	57.82	52.37	69.13	51.01	62.89

Table 3: Experimental evaluation of RNNs for ESE extraction

Adding Depth. Next, we quantitatively investigate the effects of adding depth to RNNs. Tables 2 and 3 show the evaluation of RNNs of various depths and sizes. In both tables, the first group networks have approximately 24,000 parameters and the second group networks have approximately 200,000 parameters. Since all RNNs within a group have approximately the same number of parameters, they grow narrower as they get deeper. Within each group, bold shows the best result with an asterisk denoting statistically indistinguishable performance with respect to the best. As noted above, all statistical comparisons use a two-sided paired t-test with a confidence level of $\alpha = .05$.

In both DSE and ESE detection and for larger networks (bottom set of results), 3-layer RNNs provide the best results. For smaller networks (top set of results), 2, 3 and 4-layer RNNs show equally good performance for certain sizes and metrics and, in general, adding additional layers degrades performance. This could be related to how we train the architectures as well as to the decrease in width of the networks. In general, we observe a trend of increasing performance as we increase the number of layers, until a certain depth.

Recap

- Recurrent Neural Network is one of the most widely used deepNLP model families.
- Training them is hard because of vanishing and exploding gradient problems.
- They can be extended in many ways and their training could be improved with many tricks.

Next

- Sequence to sequence models