UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

# CIS 6930 Special Topics in Large Language Models

## Recurrent Neural Network for LM

# Outline

- Recurrent Neural Network (RNN)
- Training RNN LM
- Inference and Evaluation of RNN LM
- Applications of RNN LM
- Problems of RNN – Vanishing and Exploding Gradients
- Variants of RNN – LSTM, GRU, Bi-directional RNN, Multi-layer RNN

# Feedforward Neural Language Model

output distribution

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

hidden layer

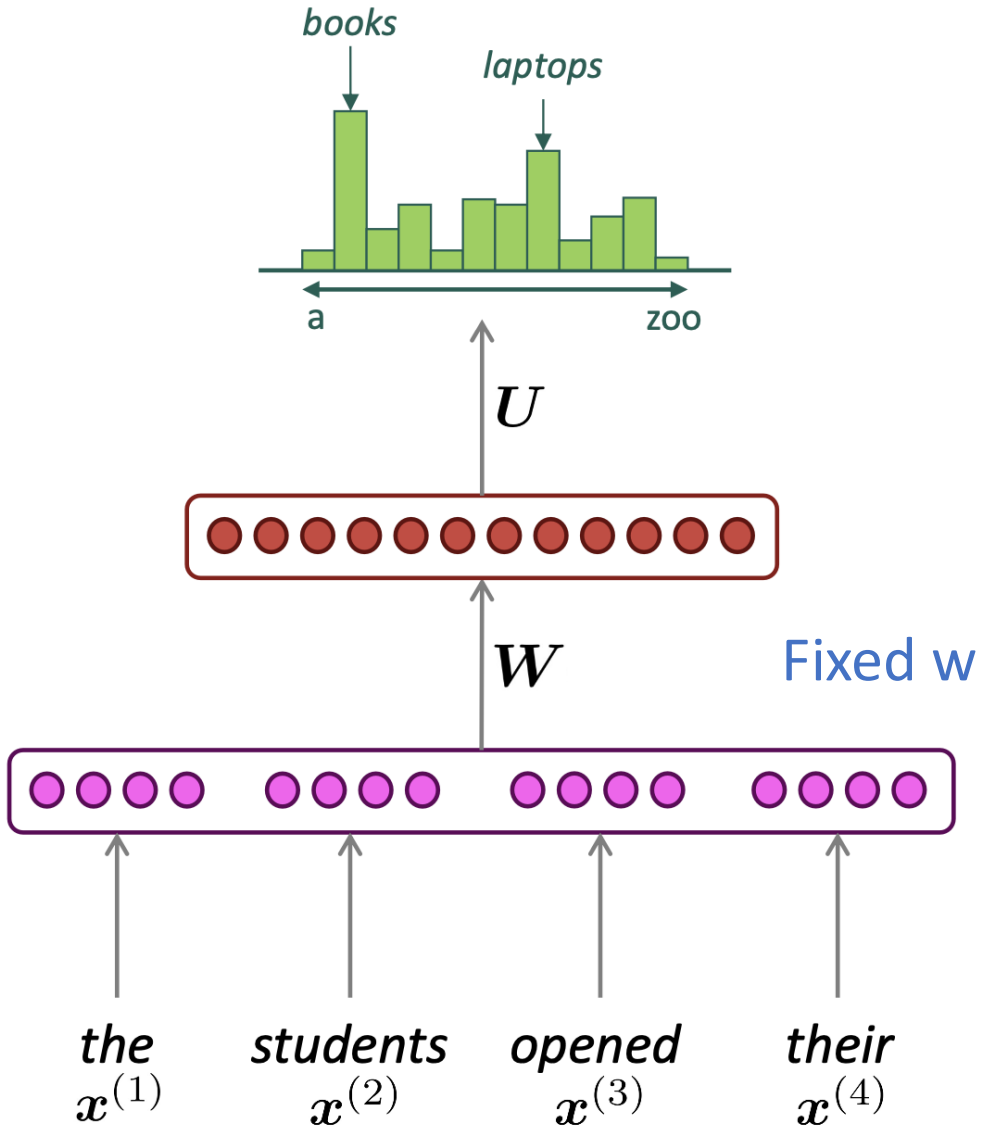
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



# Feedforward Neural Language Model

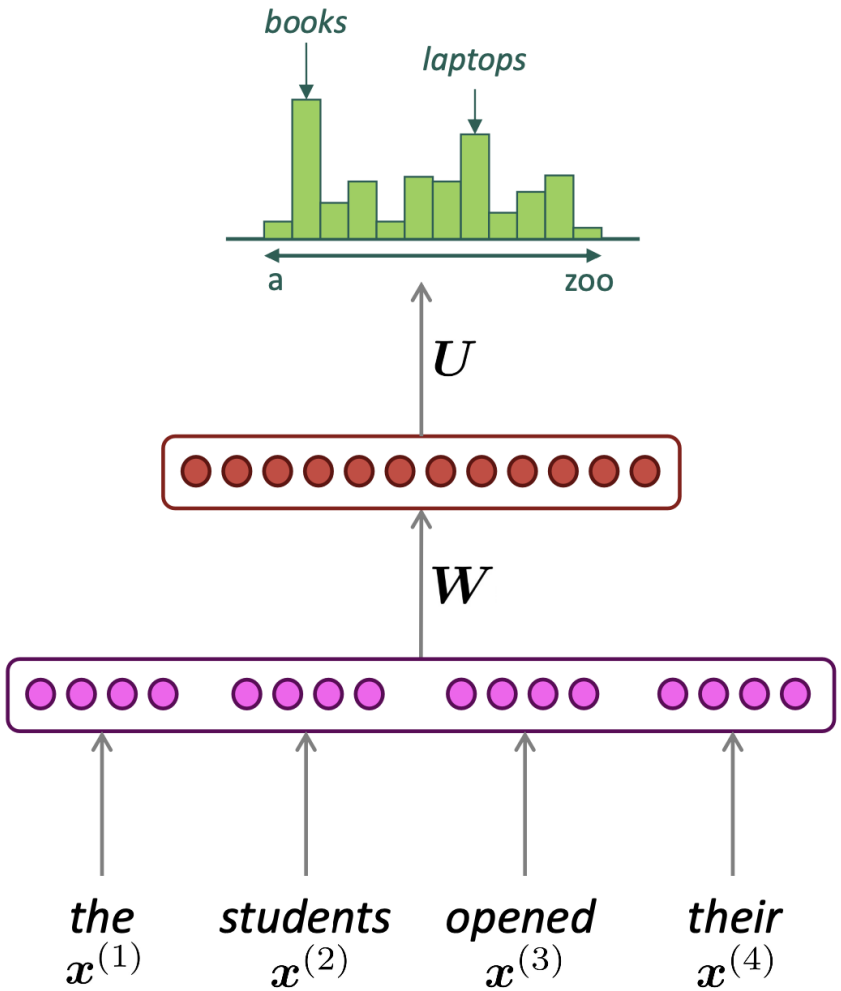
Improvements over n-gram LM:

- No sparsity problem
- Don't need to store all observed n-grams

Problems that still exists:

- Fixed window size is small
- Window size growth enlarges W
- Different W for different positions of tokens

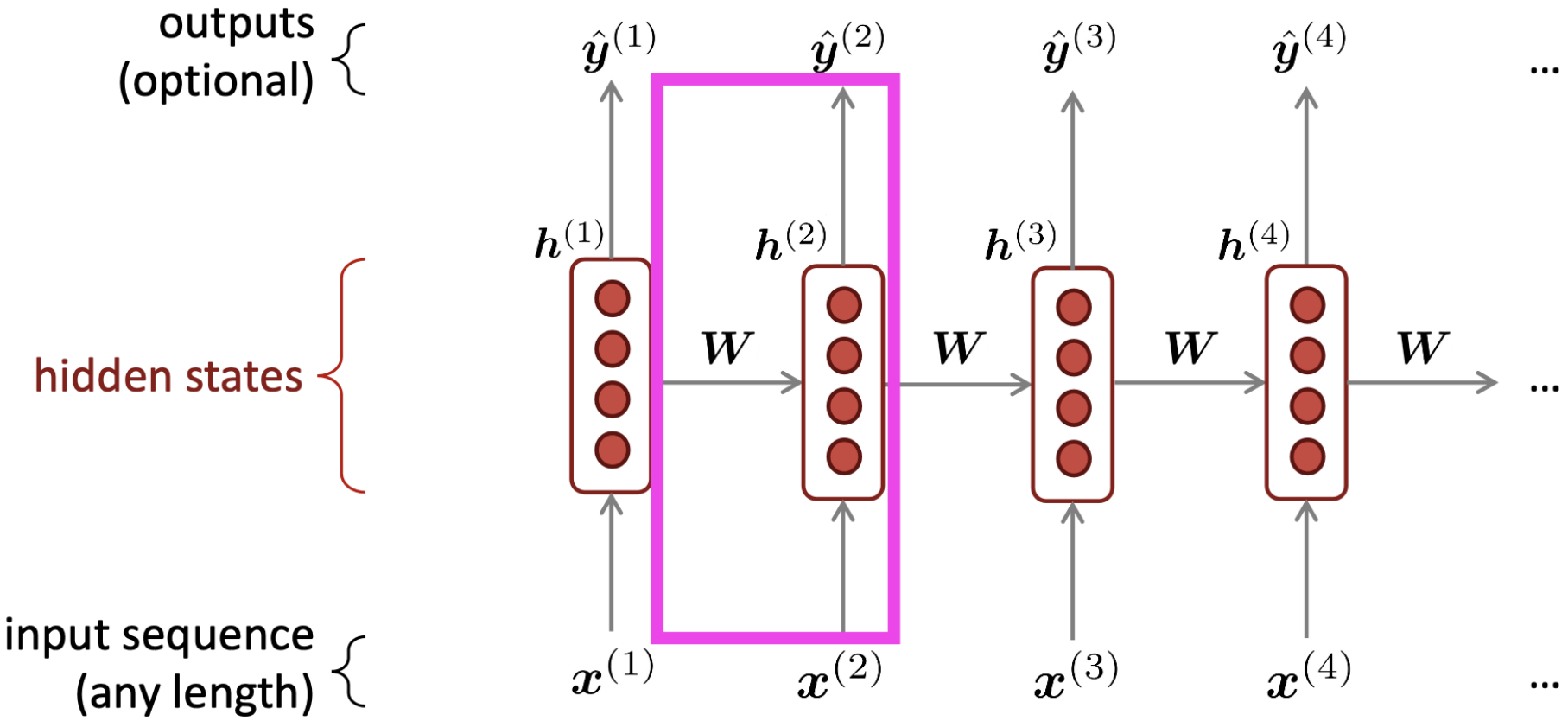
Need a model that can process **any length input**



# Recurrent Neural Network

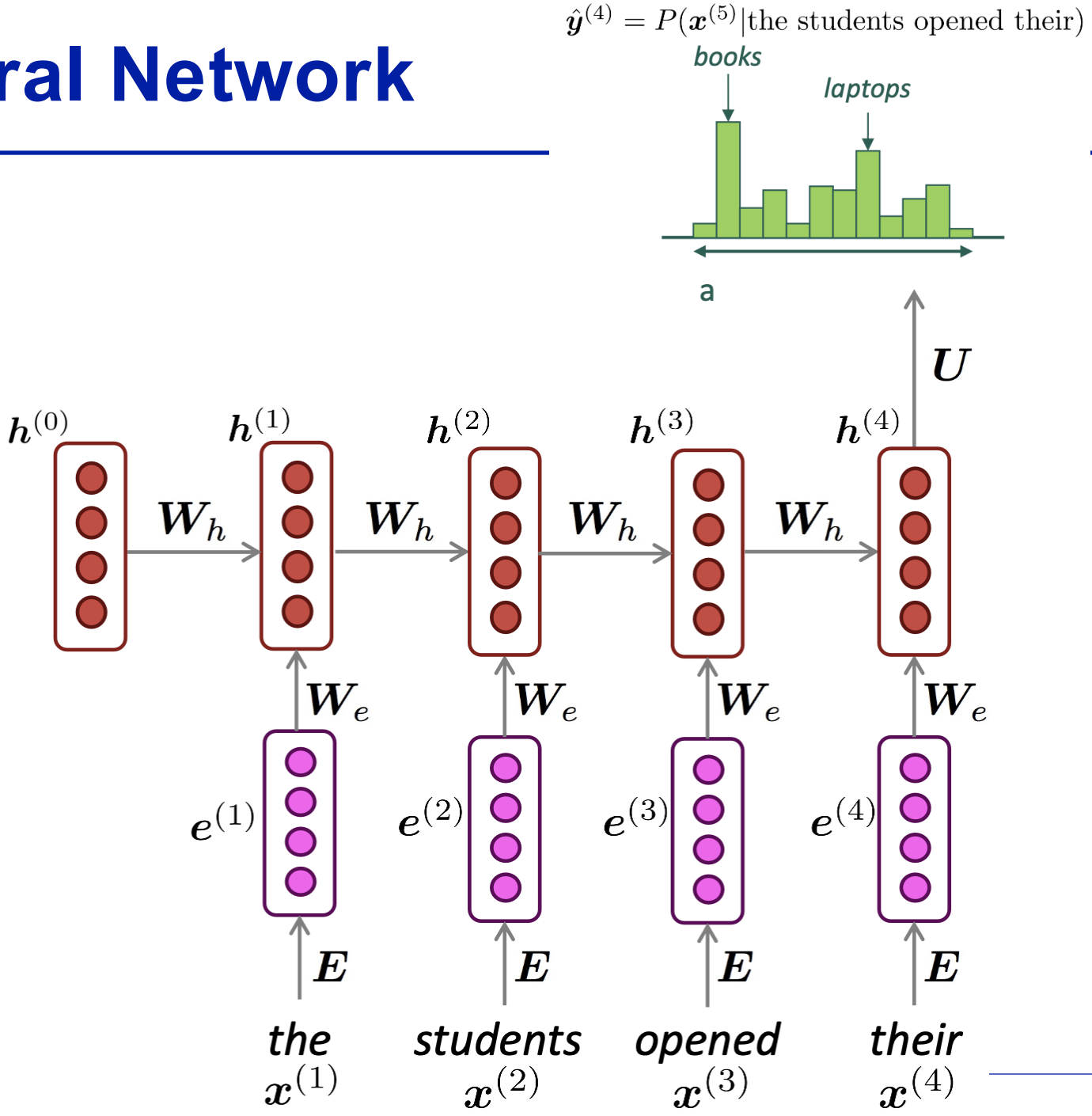
# Recurrent Neural Network

***Core Idea:*** Apply the same weights W repeatedly




RNN can handle **variabl**
**length input**

# Recurrent Neural Network




## 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

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

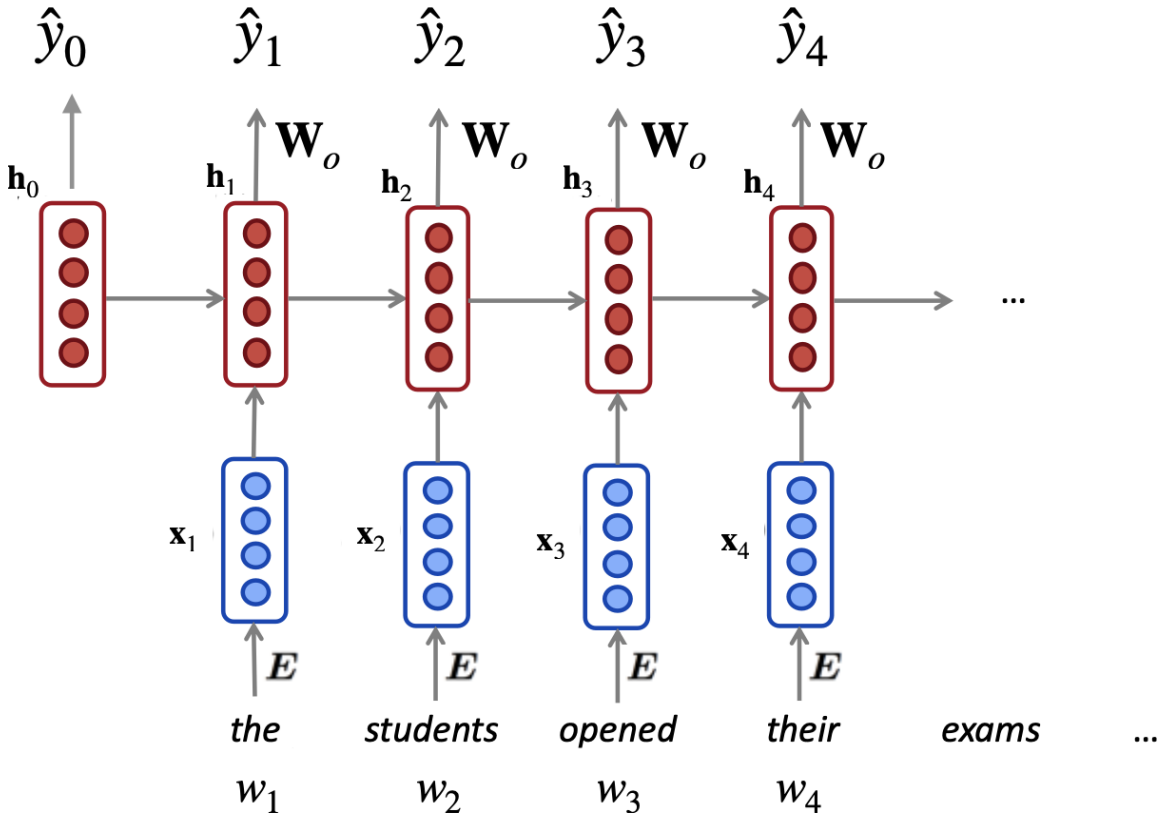
# Recurrent Neural Network

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \times P(w_2 \mid w_1) \times P(w_3 \mid w_1, w_2) \times \ldots \times P(w_n \mid w_1, w_2, \ldots, w_{n-1})$$

$$\approx P(w_1 \mid \mathbf{h}_0) \times P(w_2 \mid \mathbf{h}_1) \times P(w_3 \mid \mathbf{h}_2) \times \ldots \times P(w_n \mid \mathbf{h}_{n-1})$$

No Markov assumption here!

Assume hidden states contain all information from the past

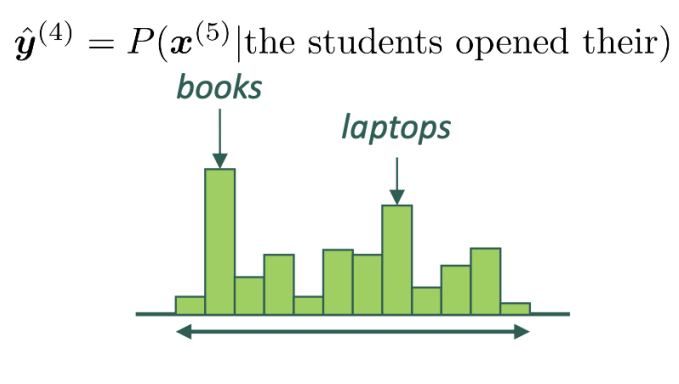



Denote $\hat{\mathbf{y}}_t = softmax(\mathbf{W}_o \mathbf{h}_t)$, $\mathbf{W}_o \in \mathbb{R}^{|V| \times h}$

$$= \hat{y}_0(w_1) \times \hat{y}_1(w_2) \ldots \times \hat{y}_{n-1}(w_n)$$

$\hat{y}_1(w_2)$ = the probability of $w_2$

# RNN Language Model

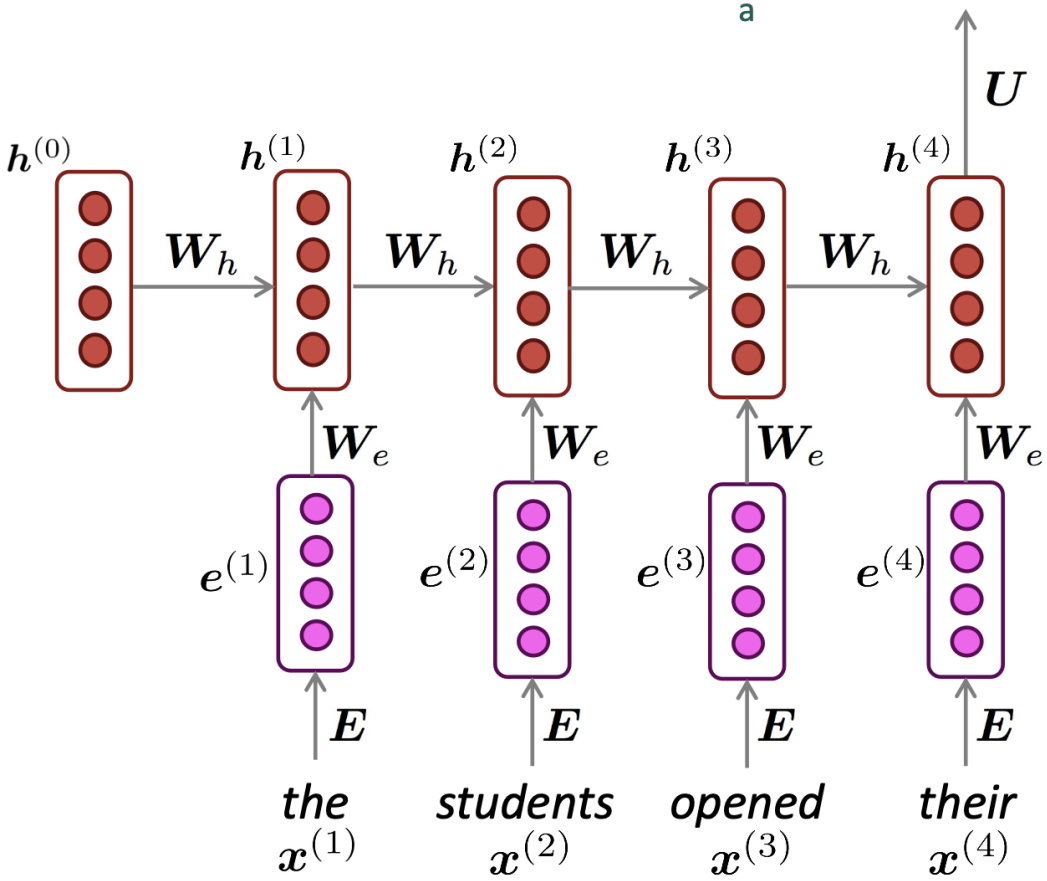



**Pros**

- Can process **any length** input
- Memory size **not increase** for longer input
- Can use info from **many steps back**
- Same weights W applied on every position

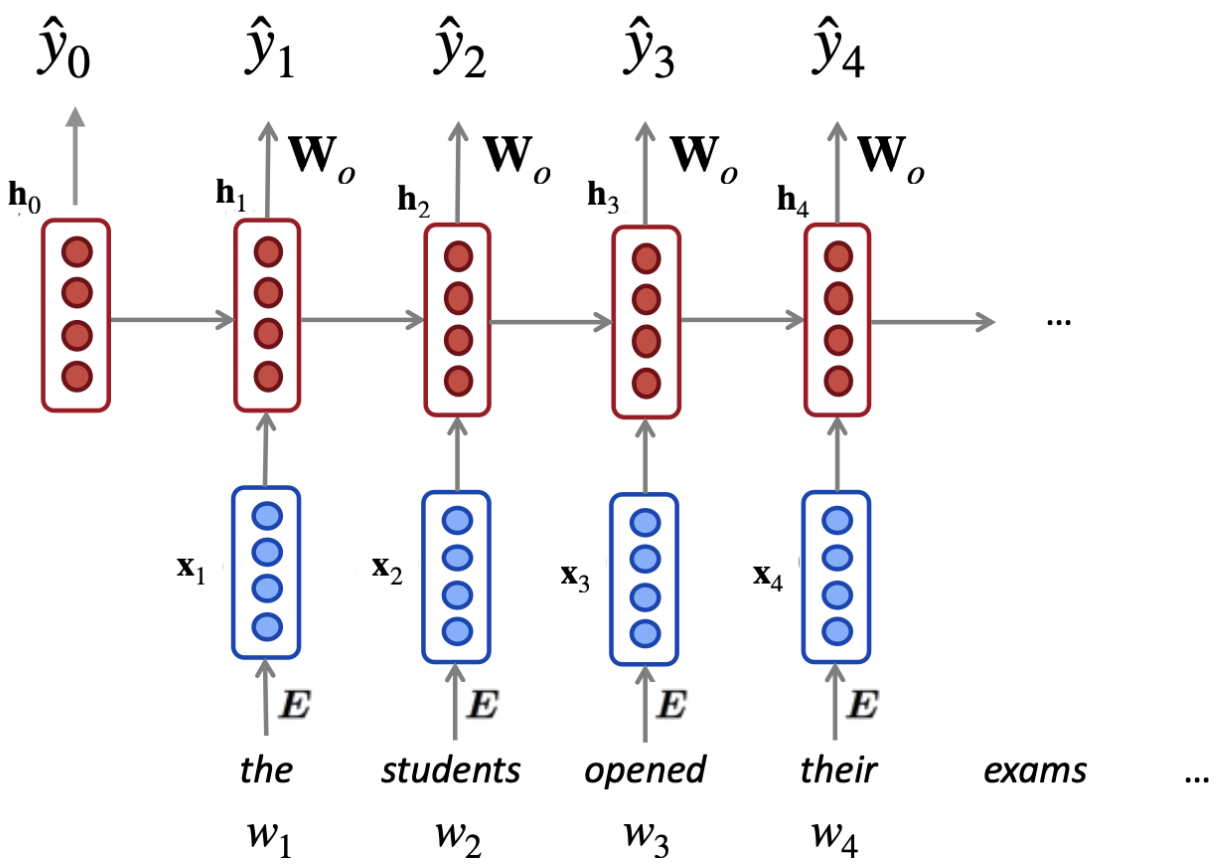
**Cons:**

- Recurrent computation is slow, no parallel
- Hard to access info from many steps back (in practice)



# Training Recurrent Neural Network LM

# Training RNN LM




$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^h$$

$$\hat{\mathbf{y}}_t = softmax(\mathbf{W}_o\mathbf{h}_t)$$

Training loss:

$$L(\theta) = -\frac{1}{n}\sum_{t=1}^{n} \log \hat{\mathbf{y}}_{t-1}(w_t)$$

Trainable parameters:

$$\theta = \{\mathbf{W}, \mathbf{U}, \mathbf{b}, \mathbf{W}_o, \mathbf{E}\}$$

# Training RNN LM

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

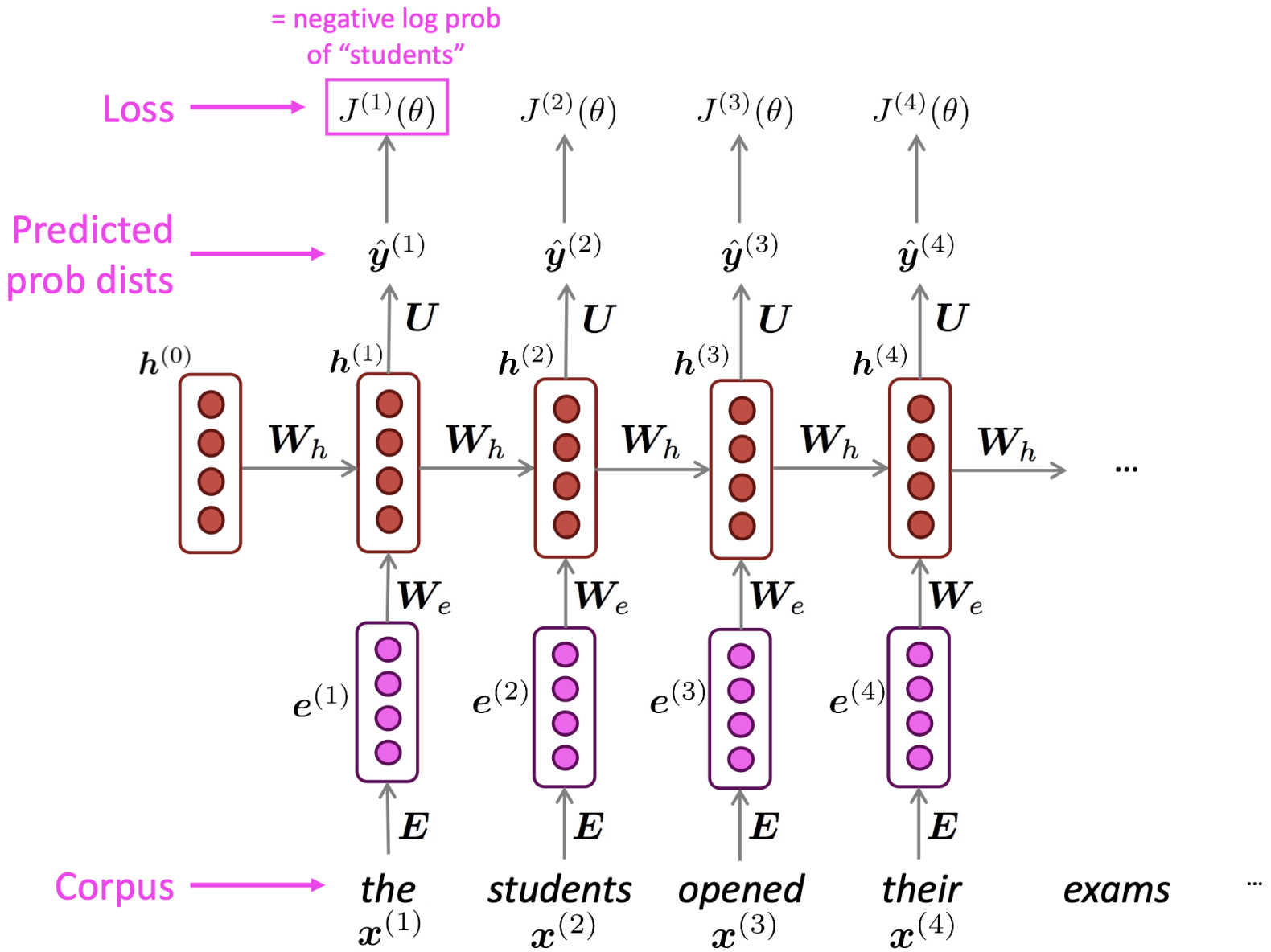
- Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

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

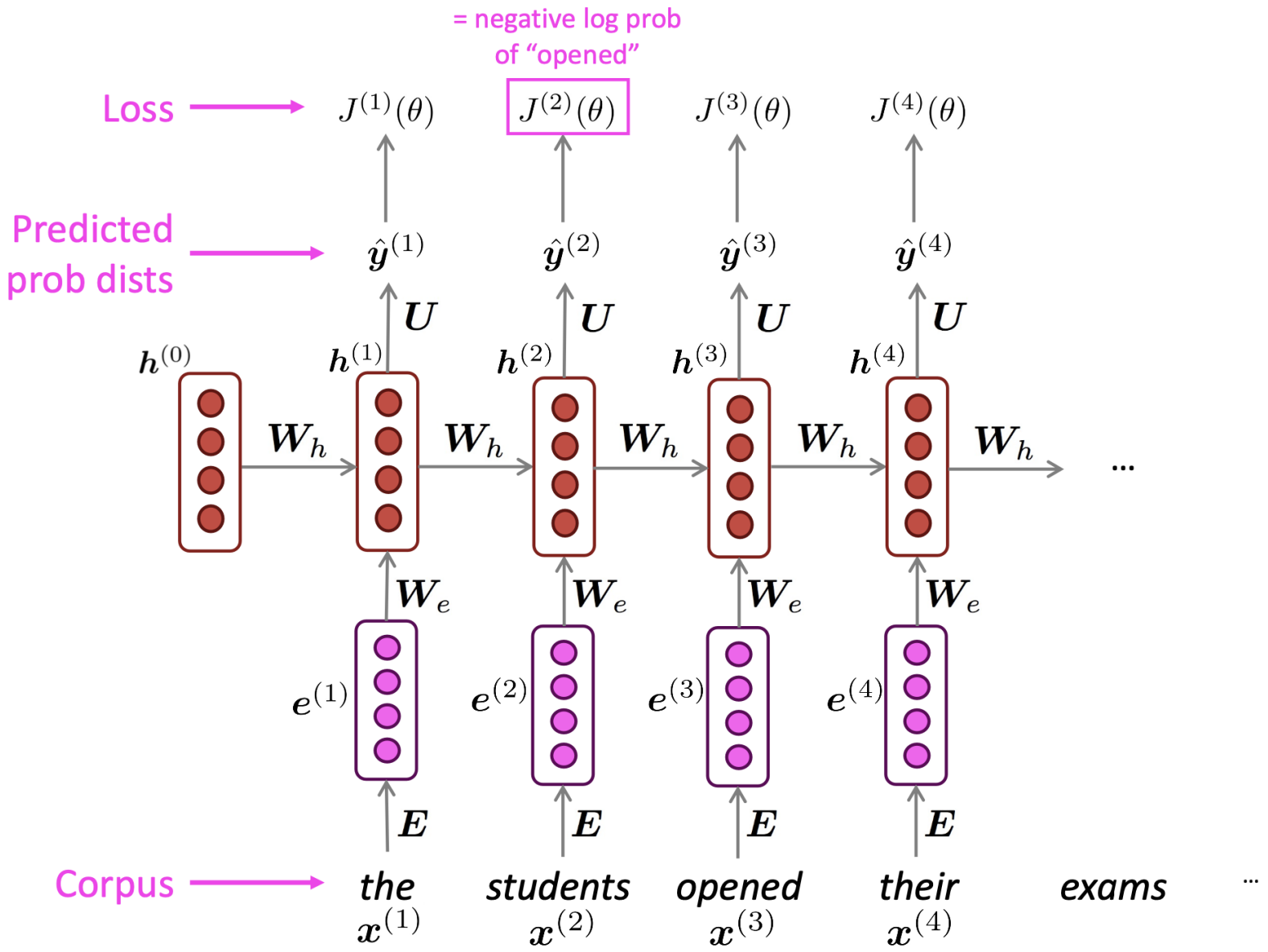
- Average this to get overall loss for entire training set:

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

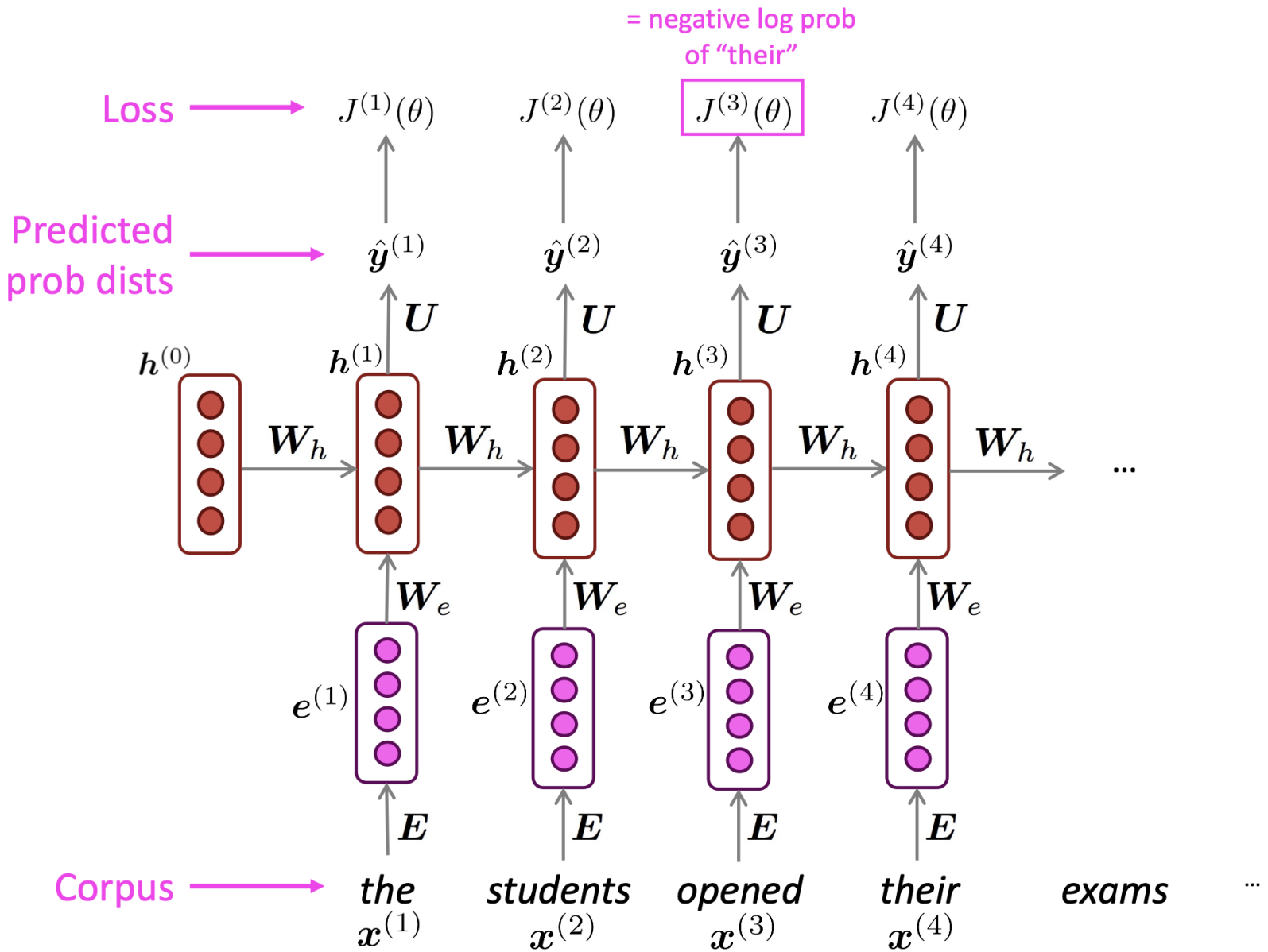
# Training RNN LM



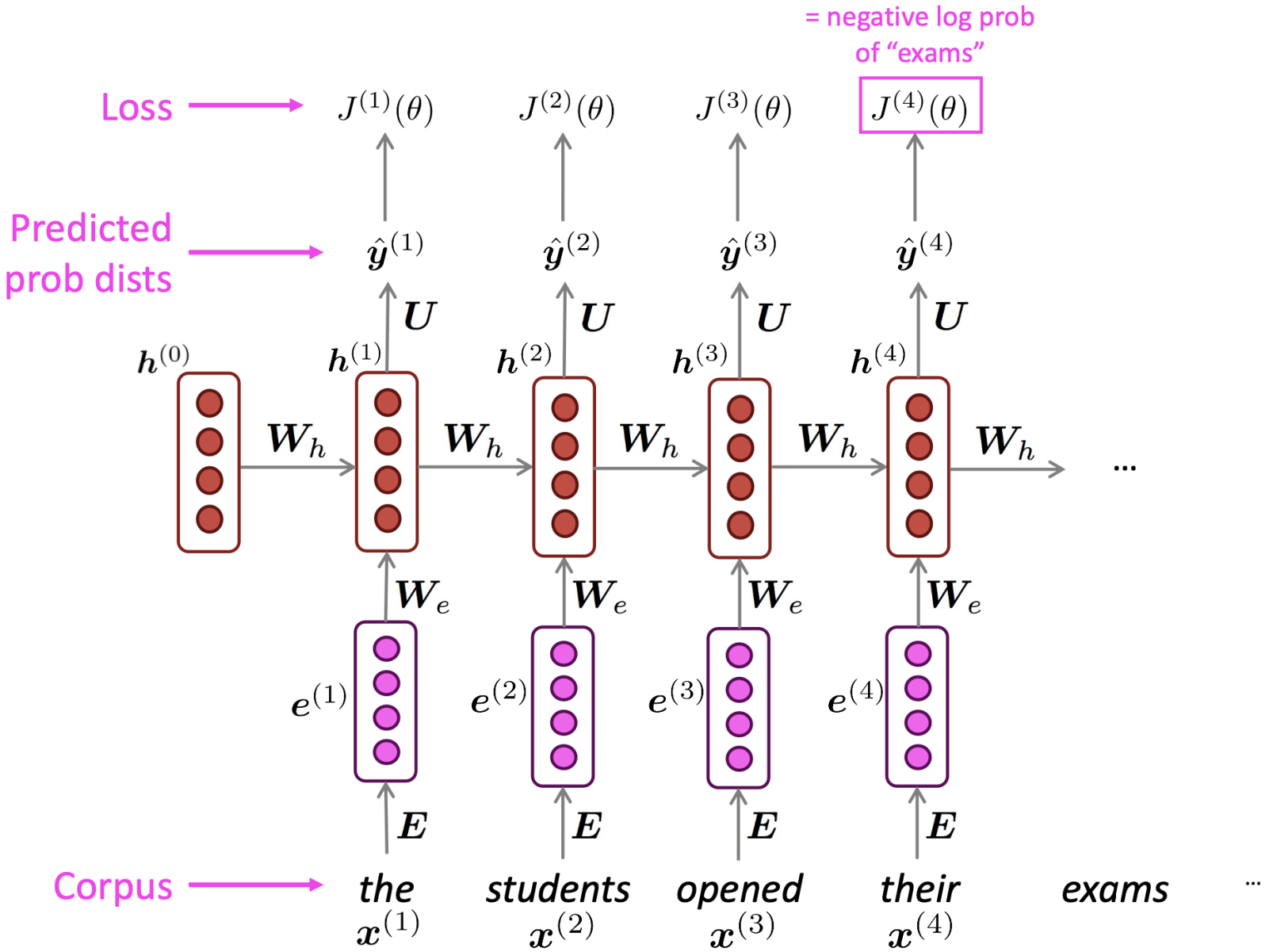
# Training RNN LM



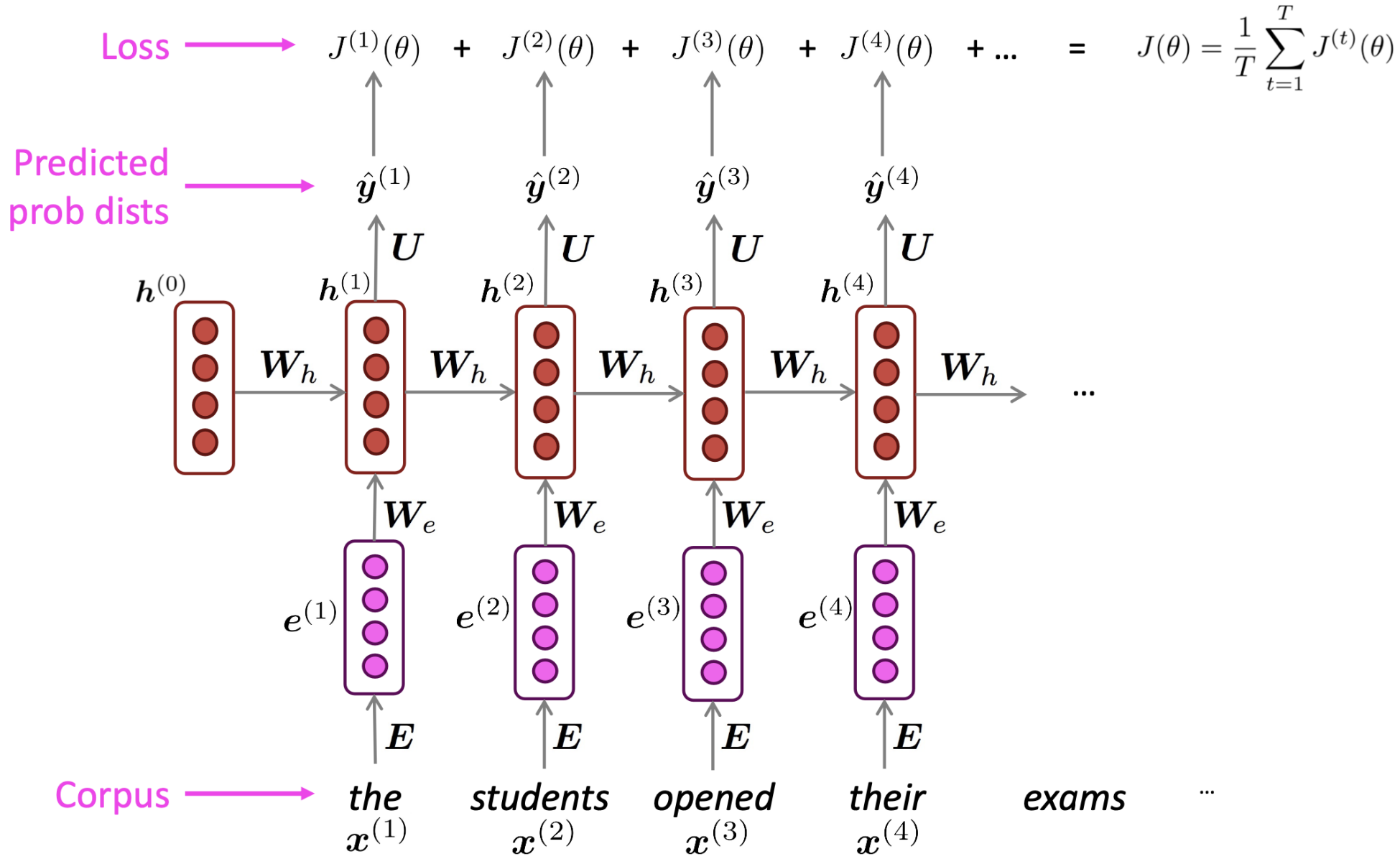
# Training RNN LM



# Training RNN LM



# Training RNN LM



# Training RNN LM

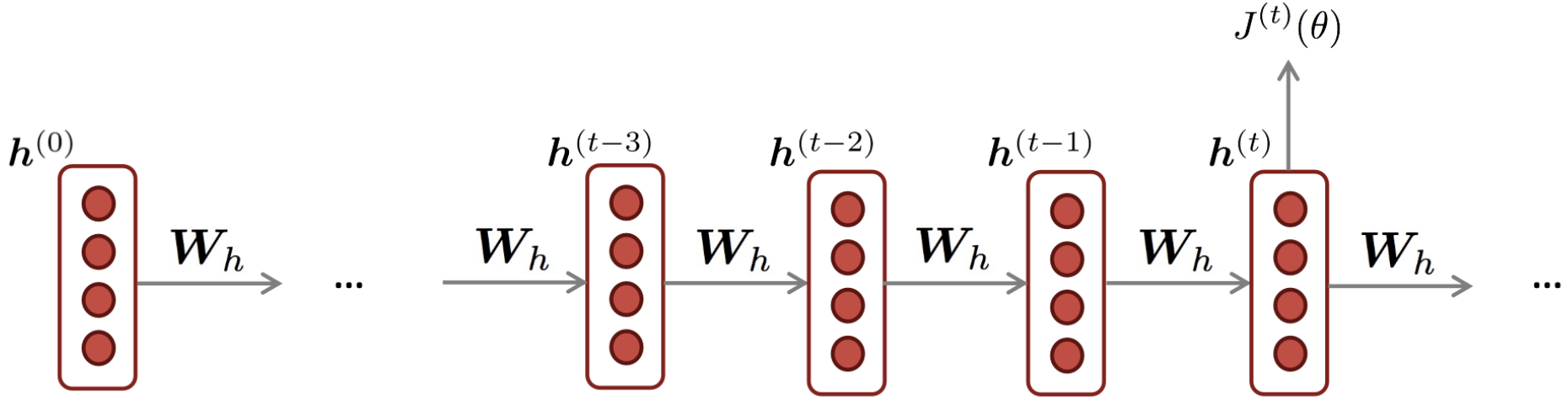
- Computing loss and gradients across entire corpus is too expensive

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

- In practice, consider training on sentence or document

- Compute loss $J(\theta)$ for a sentence, or a batch of sentences, and compute gradients and update weights, iteratively repeat until converge

# Backpropagation for RNN




**Question:** What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W}_h$ ?

**Answer:** $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\Big|_{(i)}$$

"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

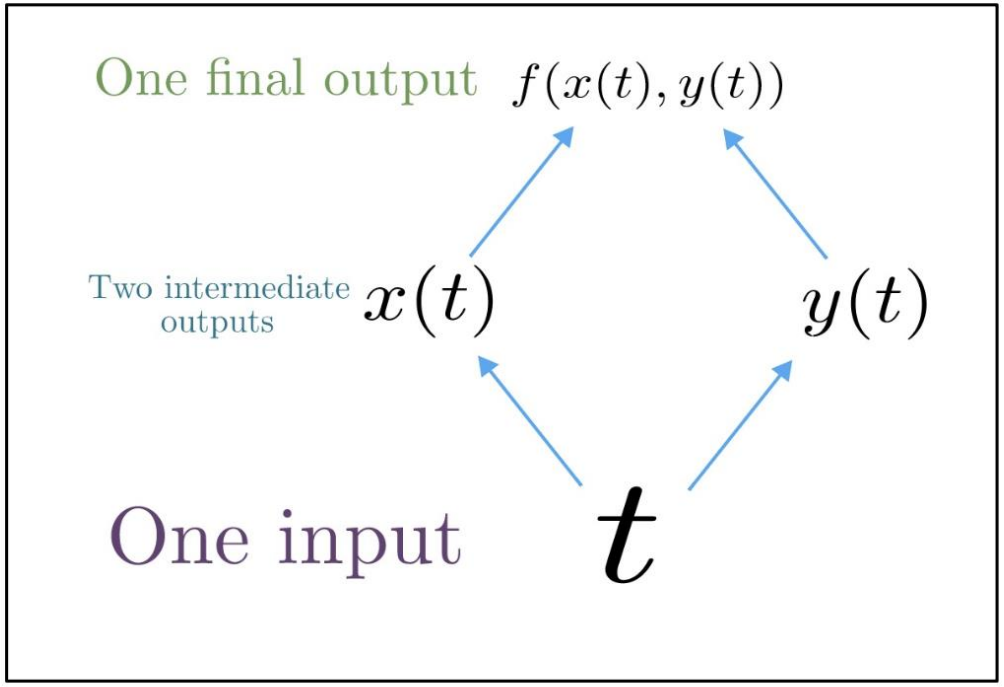
**Why?**

# Backpropagation for RNN

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

## Multivariable Chain Rule

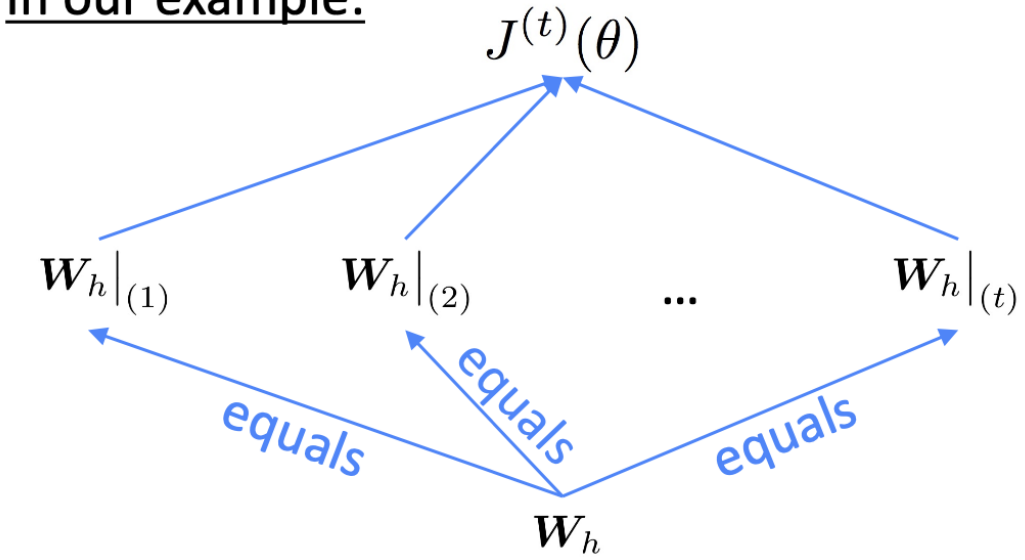


# Backpropagation for RNN

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

**In our example:**



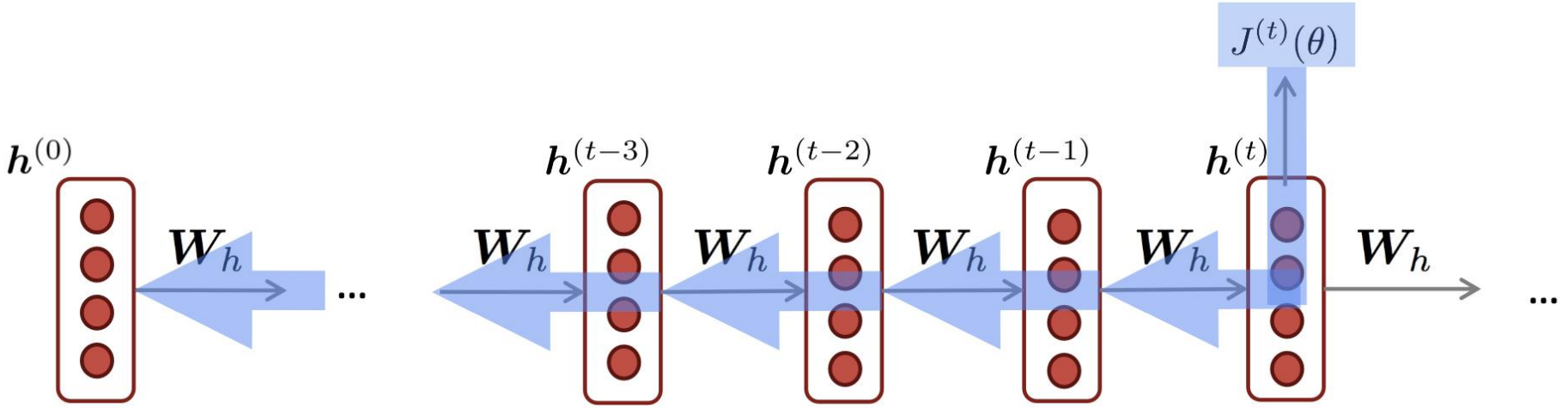

**Apply the multivariable chain rule:**

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)} \frac{\partial \boldsymbol{W}_h|_{(i)}}{\partial \boldsymbol{W}_h}$$

$= 1$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)}$$

# Backpropagation for RNN



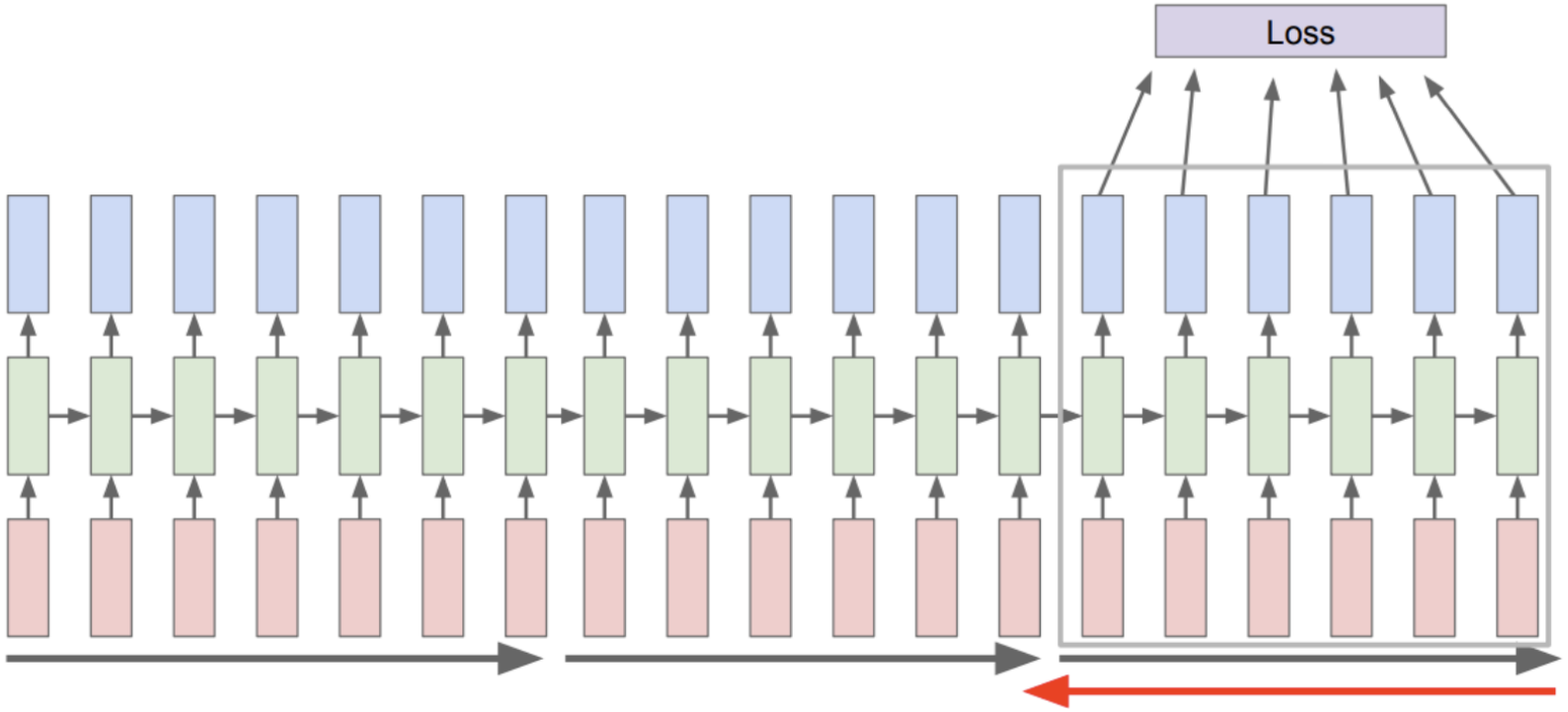

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

**Backpropagation through time**
(Backpropagate over timesteps *i=t,…,0*, summing gradients as you go)

In practice, truncate after 20 timesteps for training efficiency

# Truncated Backpropagation for RNN

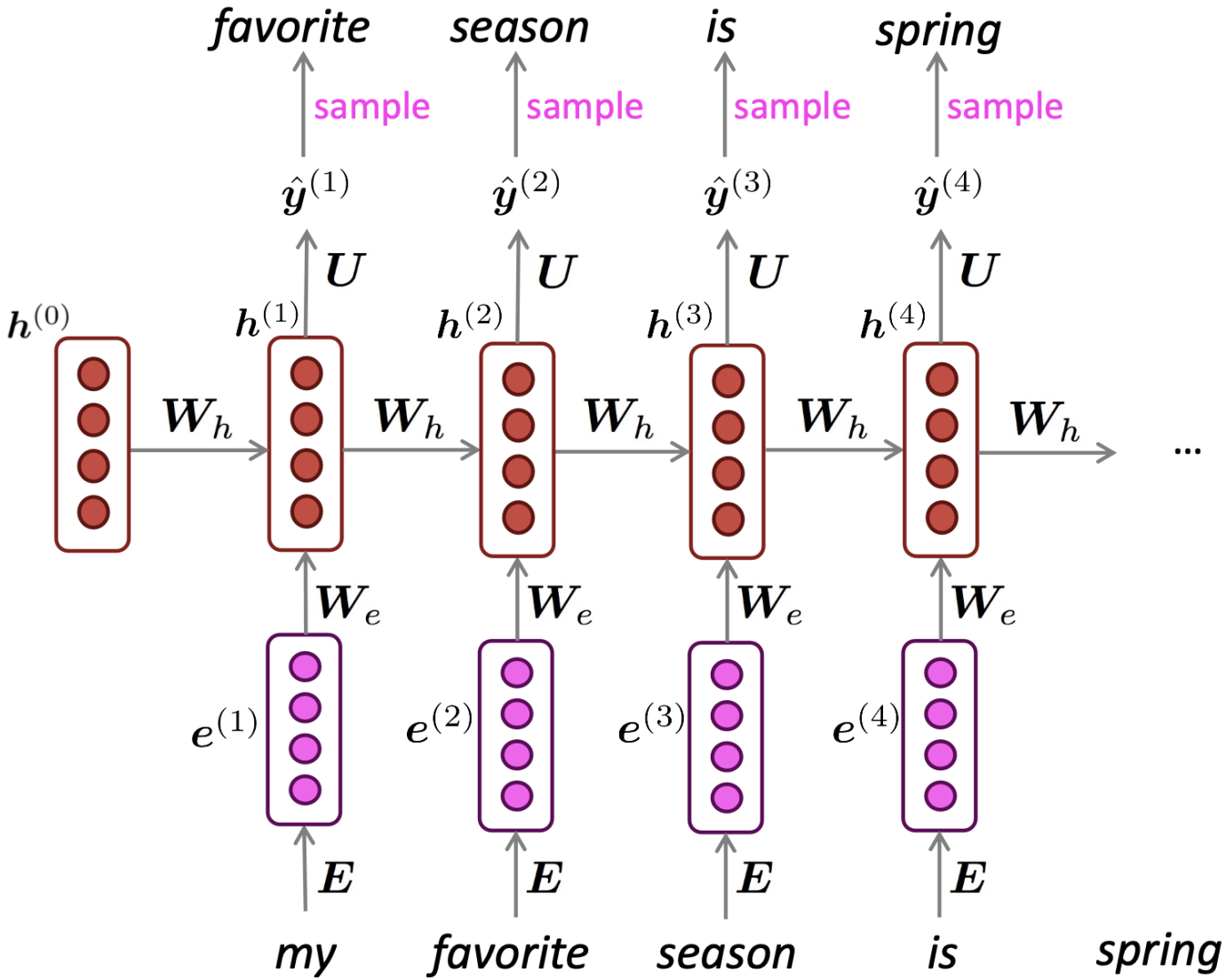
- Backpropagation is very expensive if you handle long sequences




- Run forward and backward through chunks of the sequence instead of whole sequence
- Carry hidden states forward in time forever, but only back-propagate for some smaller number of steps

# Inference of Recurrent Neural Network LM

# Generating Text with RNN LM




Generate text by repeated sampling.

Sampled output becomes next step's input

# Evaluating Recurrent Neural Network LM

# Evaluation of RNN LM

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \prod_{t=1}^{T} \left( \frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

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

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

# Evaluation of RNN LM

*n*-gram model →

Increasingly complex RNNs ↓

| Model | Perplexity |
|---|---|
| Interpolated Kneser-Ney 5-gram (Chelba et al., 2013) | 67.6 |
| RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013) | 51.3 |
| RNN-2048 + BlackOut sampling (Ji et al., 2015) | 68.3 |
| Sparse Non-negative Matrix factorization (Shazeer et al., 2015) | 52.9 |
| LSTM-2048 (Jozefowicz et al., 2016) | 43.7 |
| 2-layer LSTM-8192 (Jozefowicz et al., 2016) | 30 |
| **Ours small** (LSTM-2048) | 43.9 |
| **Ours large** (2-layer LSTM-2048) | 39.8 |

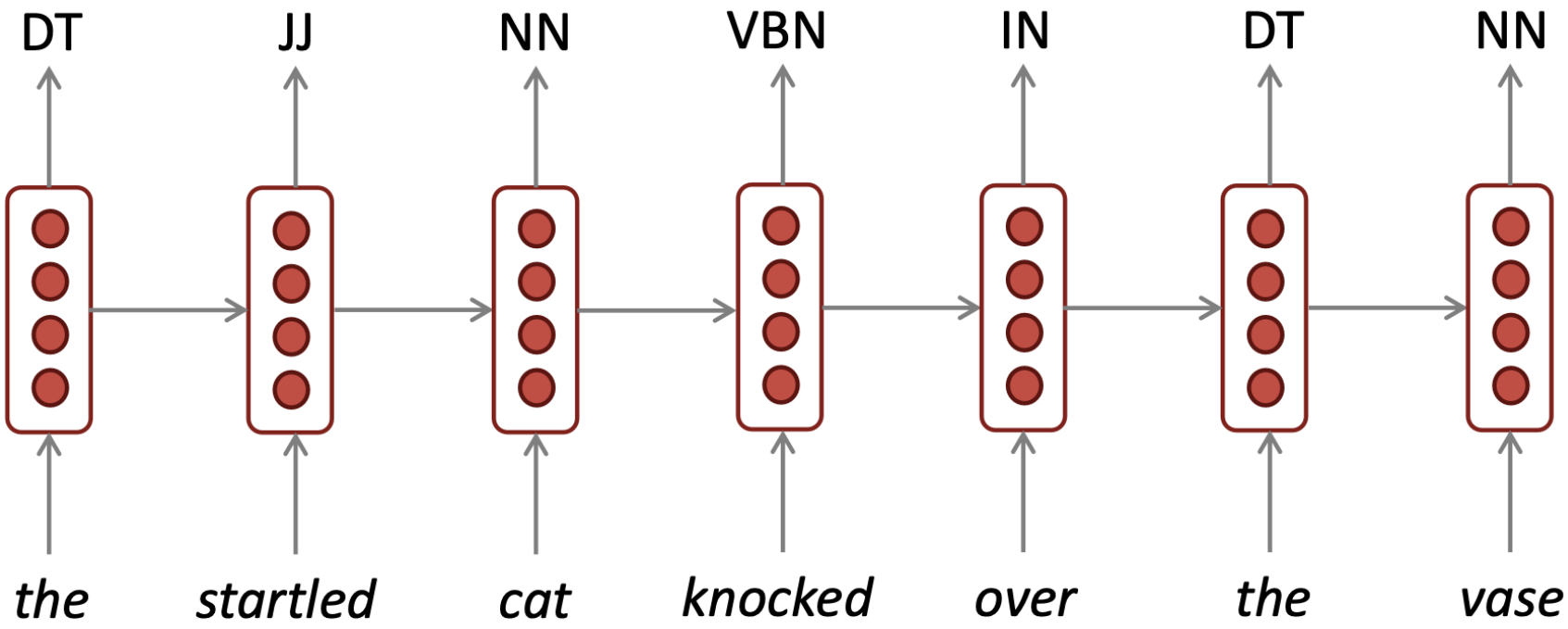
↓

Perplexity improves (lower is better)

# Application of RNN LM

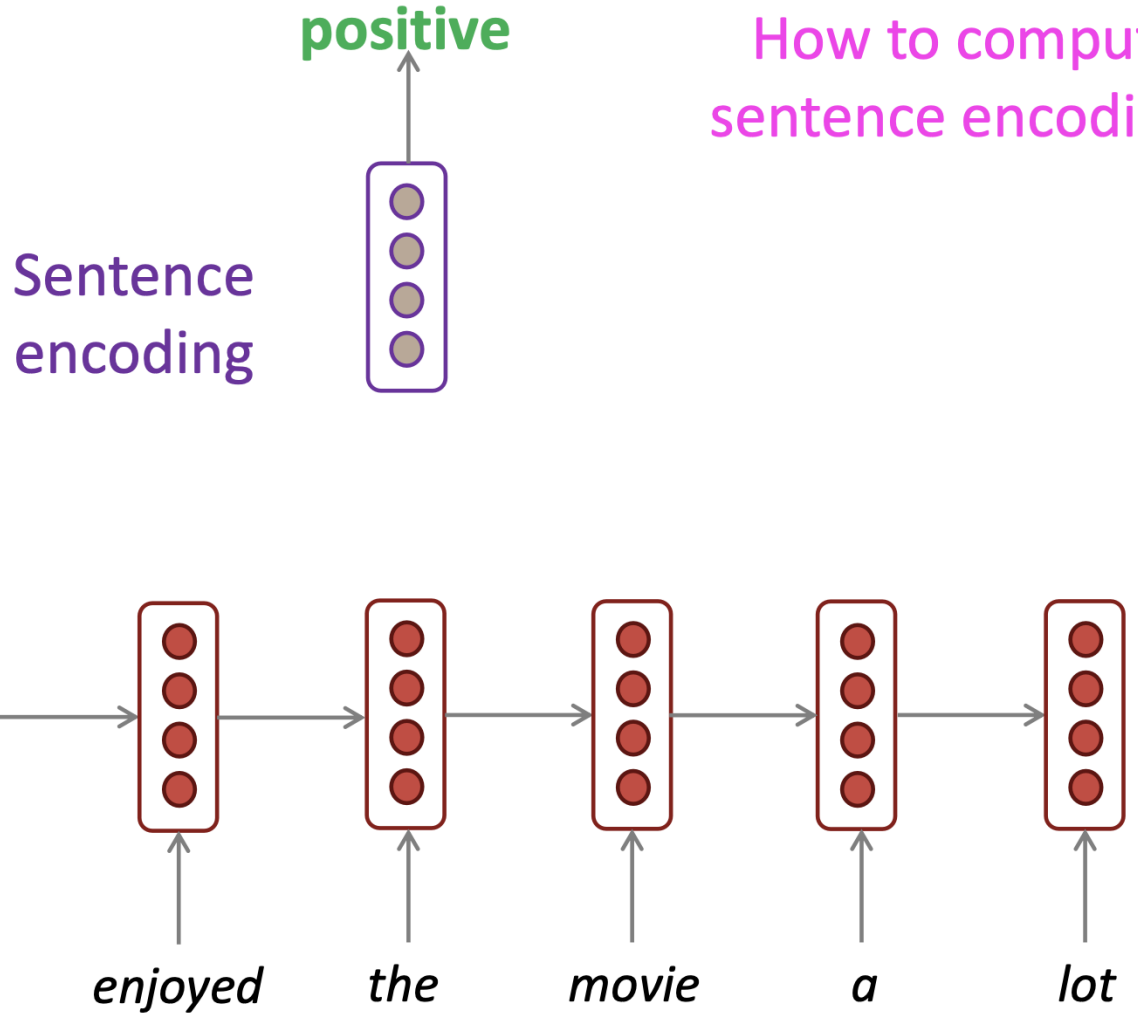
# Application 1: Token-level Classification Tasks

Part-of-Speech Tagging, Name Entity Recognition



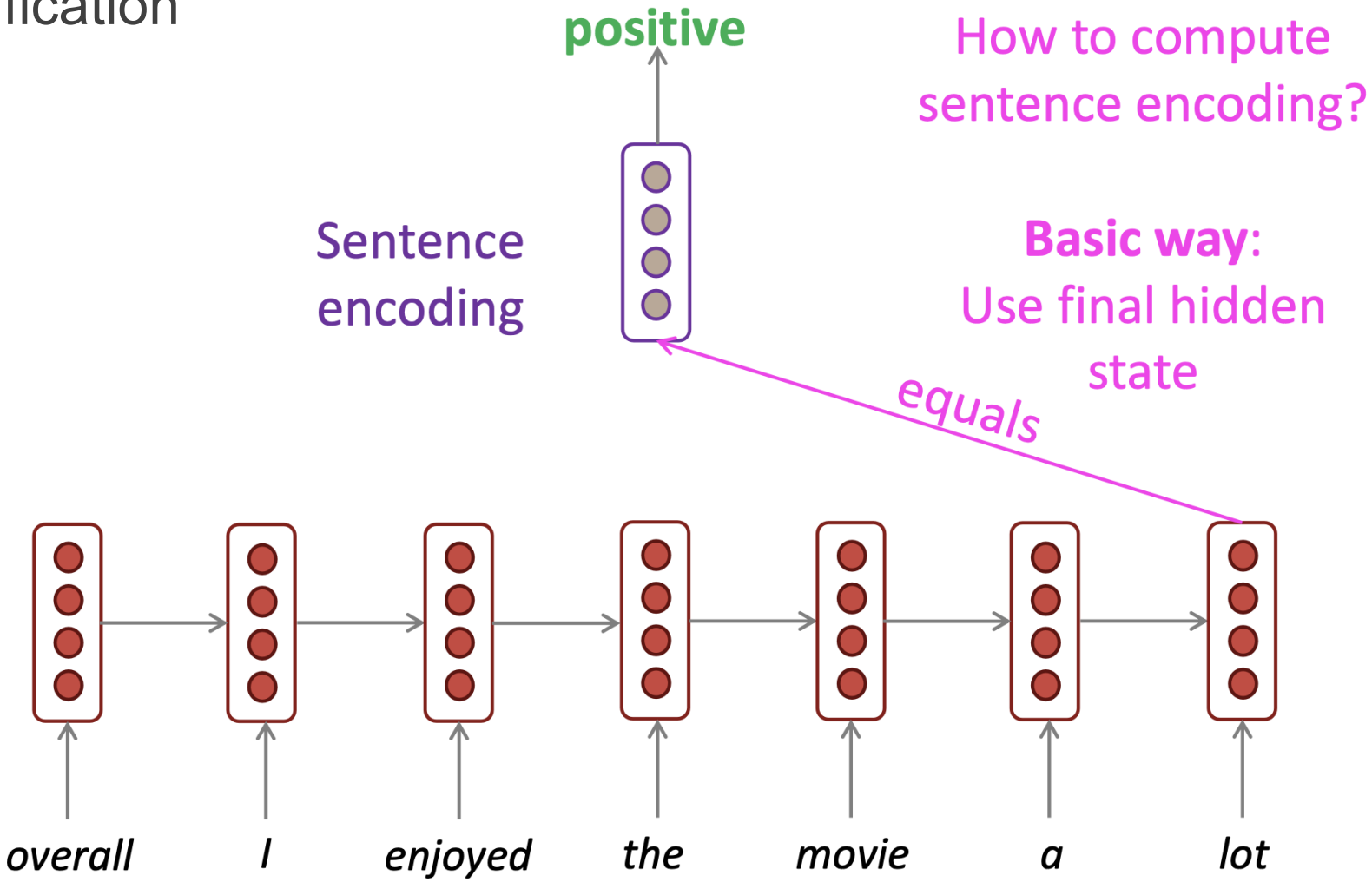
# Application 2: Sentence Classification

Sentiment Classification



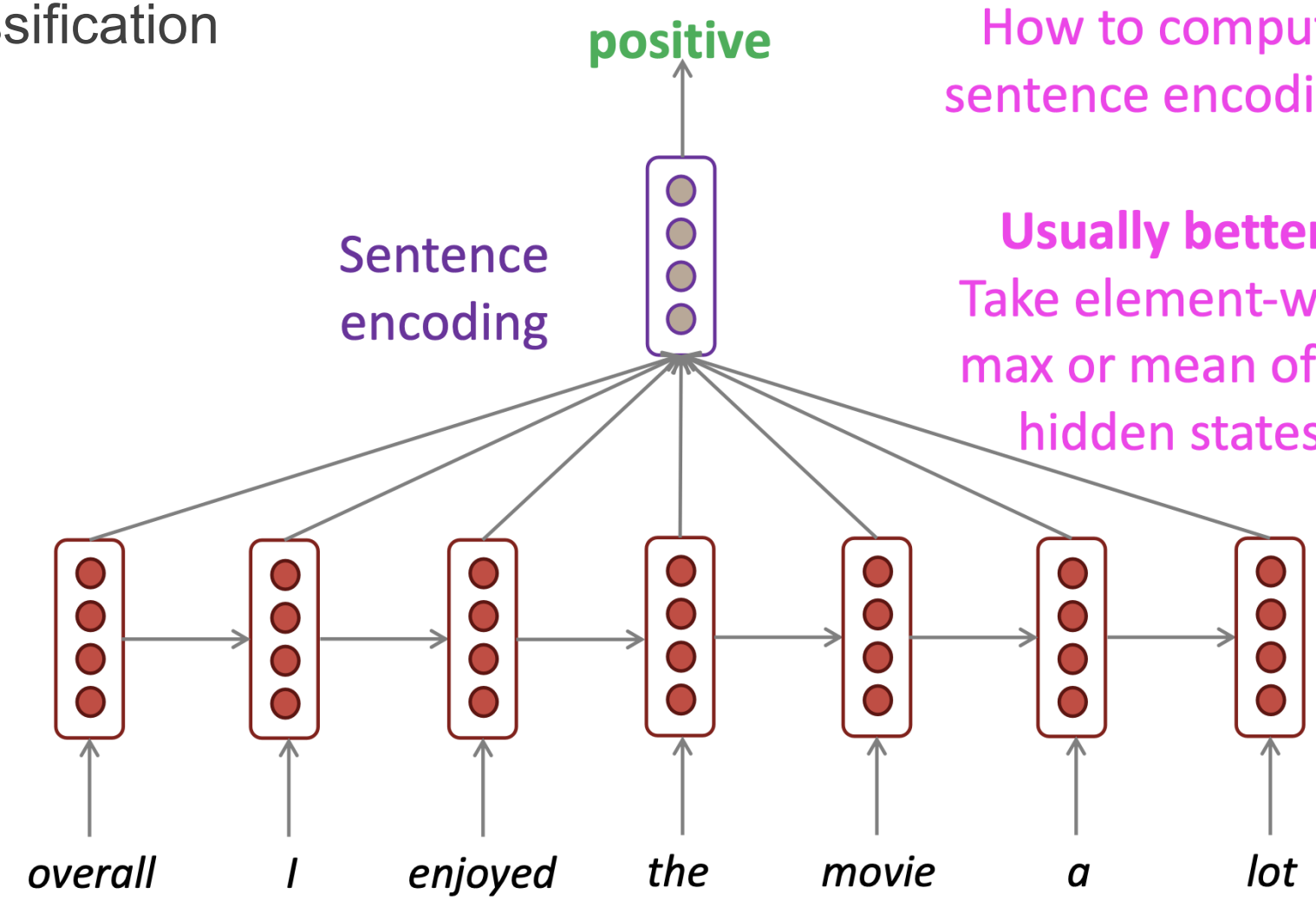
# Application 2: Sentence Classification

Sentiment Classification



# Application 2: Sentence Classification

Sentiment Classification



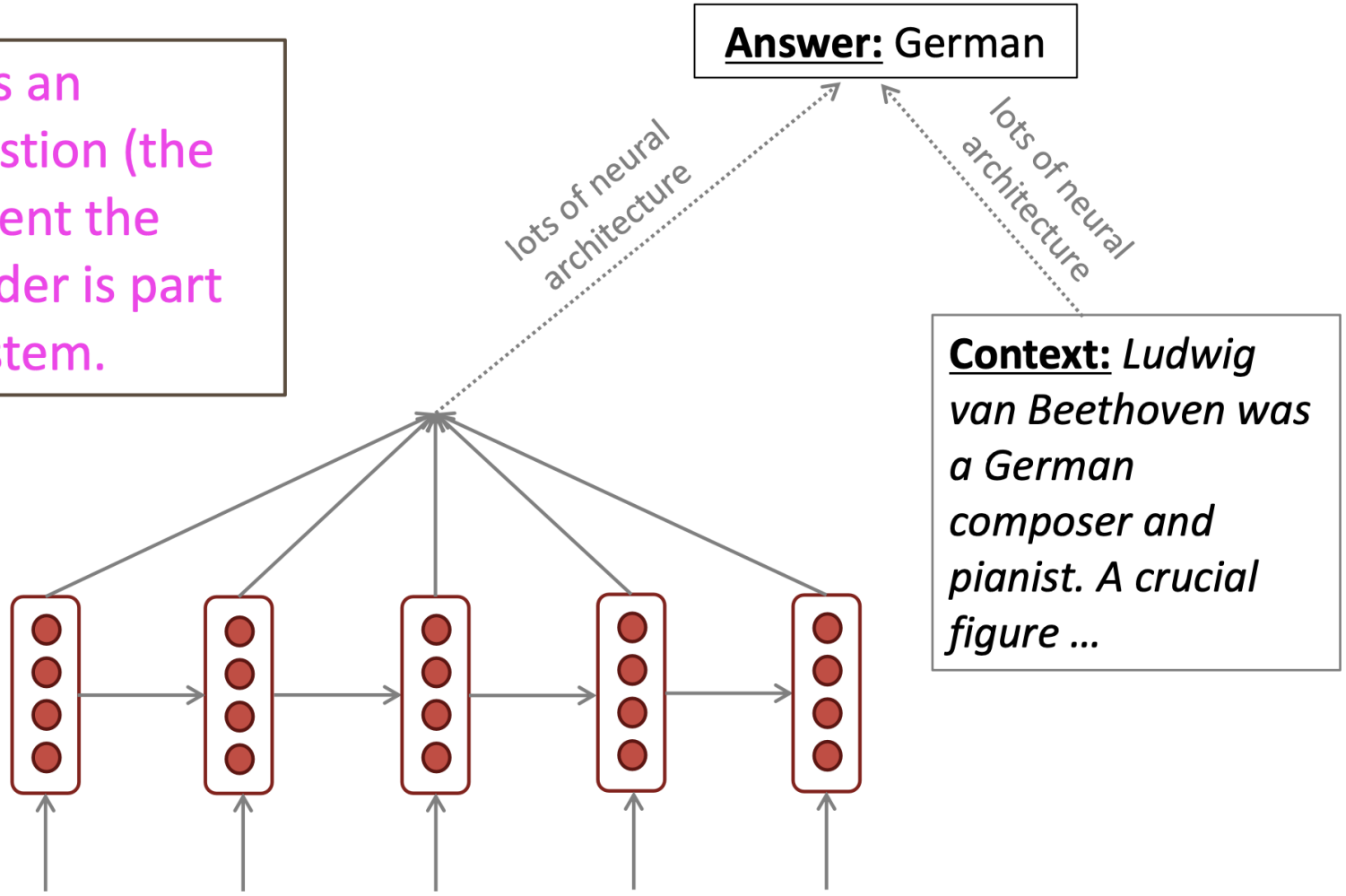

How to compute sentence encoding?

**Usually better**: Take element-wise max or mean of all hidden states

# Application 3: RNN as Encoder

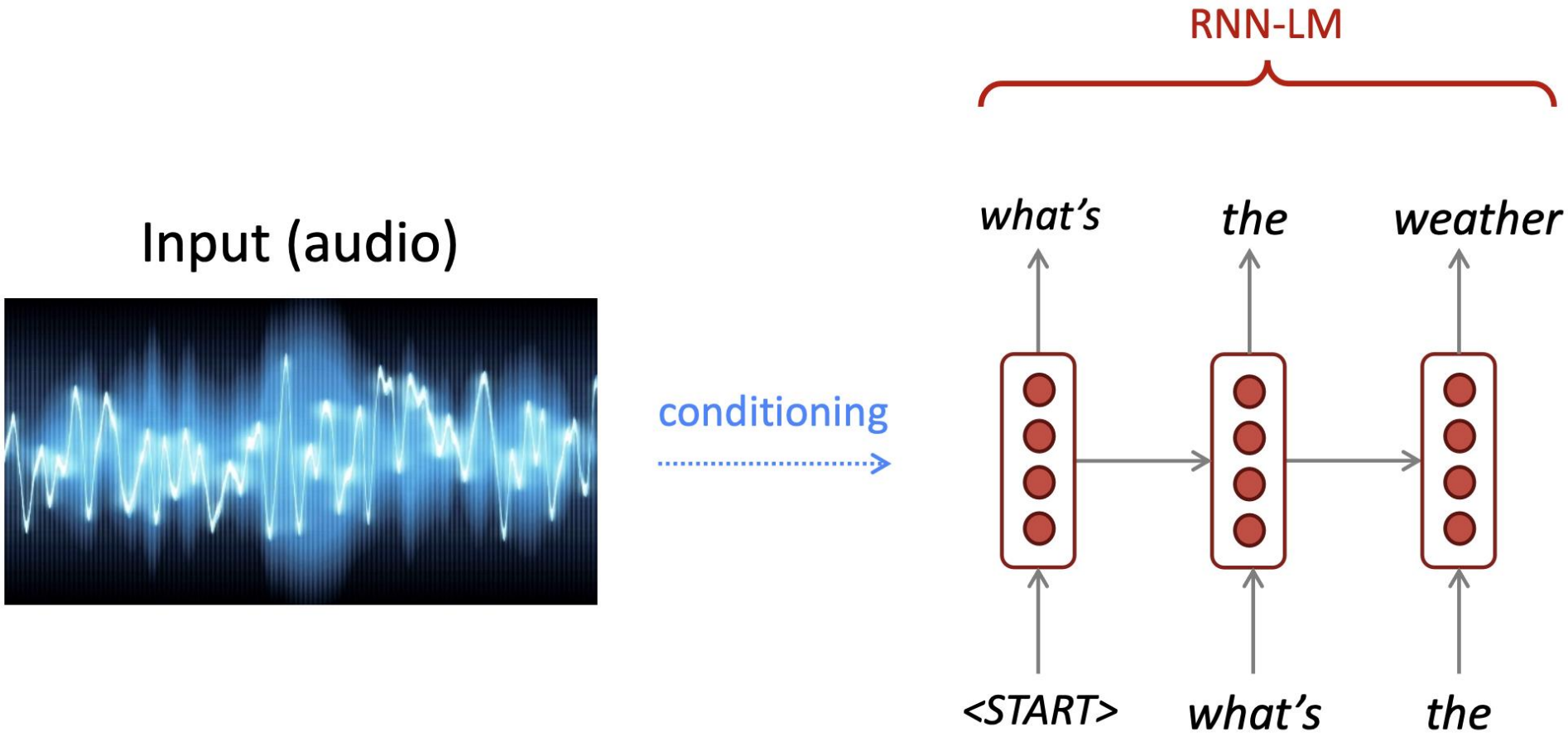
Question Answering

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.



# Application 3: RNN as Decoder

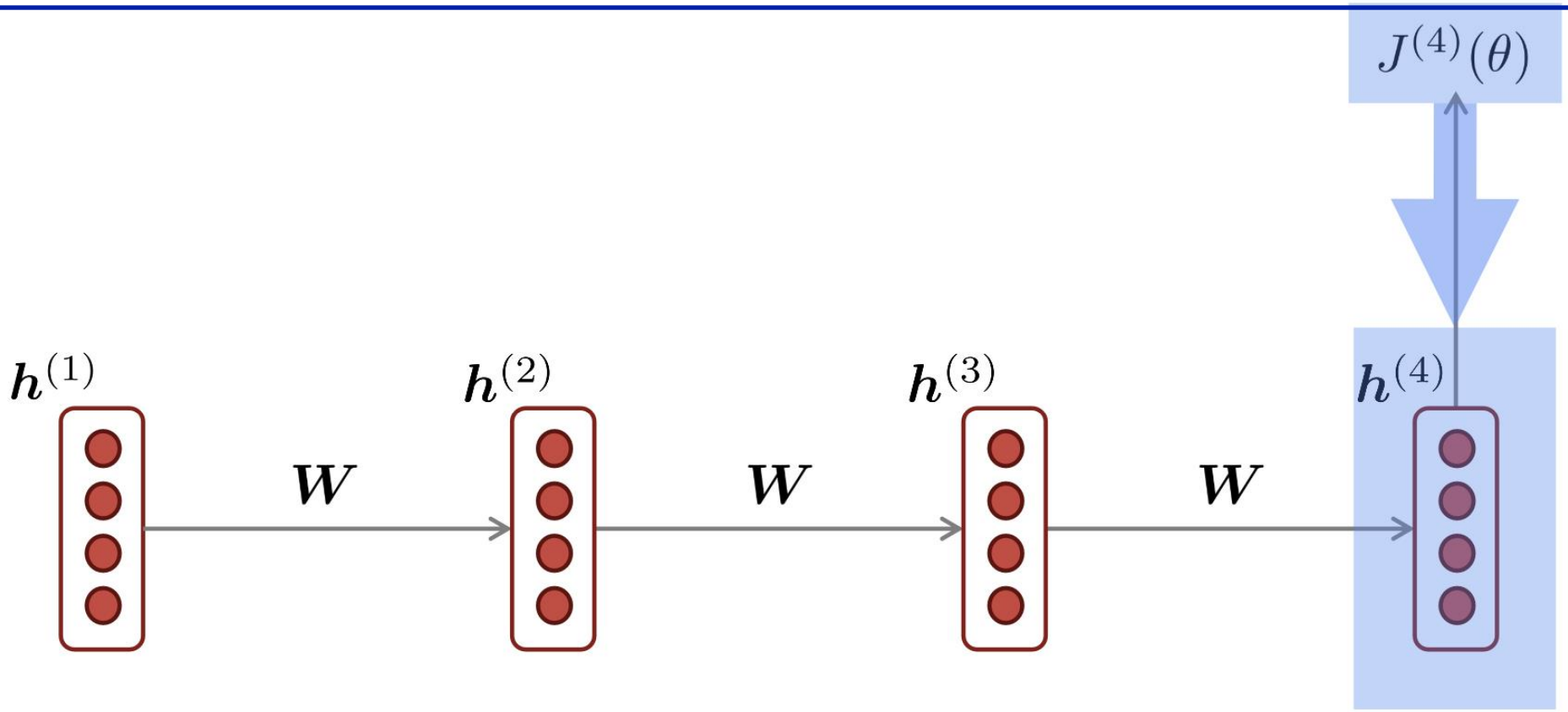
Speech Recognition, Machine Translation, Summarization etc.



# Problems of RNN LM:

# Vanishing and Exploding Gradients

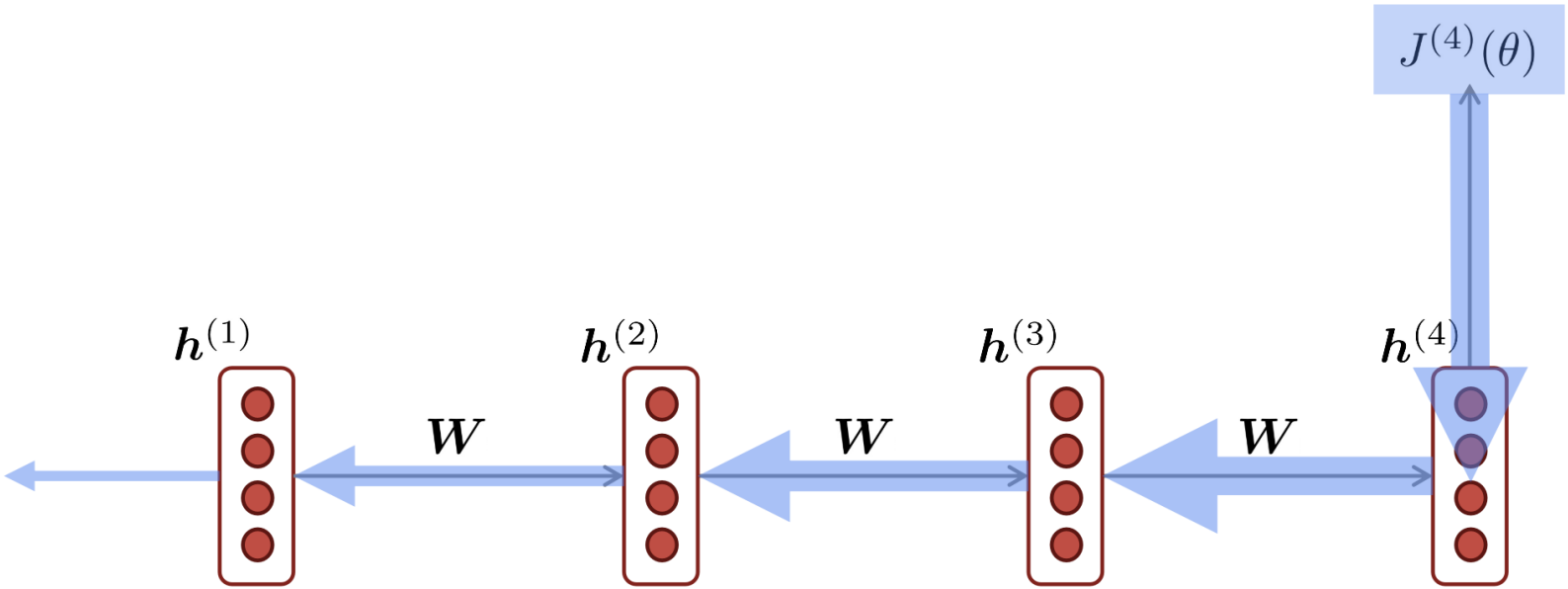
# Vanishing and Exploding Gradients




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

chain rule!

# Vanishing and Exploding Gradients



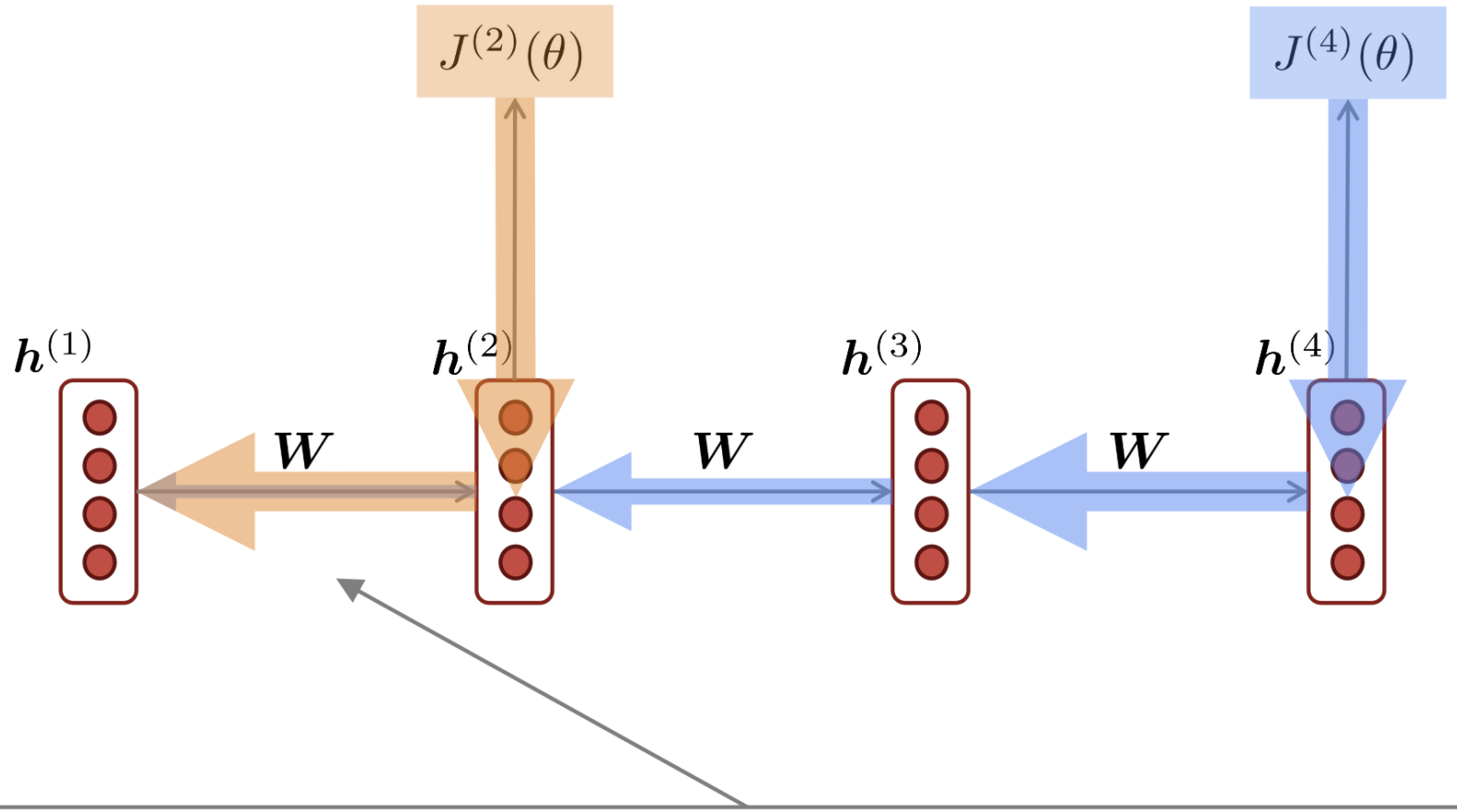

Gradients grow/shrink exponentially

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

What happens if these are small?

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

# Effect of Vanishing Gradients




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

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

# Effect of Vanishing Gradients

- **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 7$^{th}$ step and the target word *"tickets"* at the end.

- But if the 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 Exploding Gradients

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

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

- In the worst case, this will result in Inf or NaN in your network (then you have to restart training from an earlier checkpoint)

# Practice of Vanishing / Exploding Gradients

What will happen if the gradients become too large or too small?

(a) If too large, the model will become difficult to converge
(b) If too small, the model can't capture long-term dependencies
(c) If too small, the model may capture a wrong recent dependency
(d) All of the above

All of these are correct, so (d) ☺

# Solution for Exploding Gradients

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

**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**: take a step in the same direction, but a smaller step

- In practice, remembering to clip gradients is important, but exploding gradients are an easy problem to solve

# How to solve Vanishing Gradients

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

- In a vanilla RNN, the hidden state is constantly being rewritten

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

- How about an RNN with separate memory which is added to?

# Long Short-Term Memory RNN (LSTM)

# LSTM Core Idea

- On step *t*, there is a hidden state $\boldsymbol{h}^{(t)}$ *and* a cell state $\boldsymbol{c}^{(t)}$
  - Both are vectors length *n*
  - The cell stores long-term information
  - The LSTM can read, erase, and write information from the cell
    - The cell becomes conceptually rather like RAM in a computer

- 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

# LSTM Architecture

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

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

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

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

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

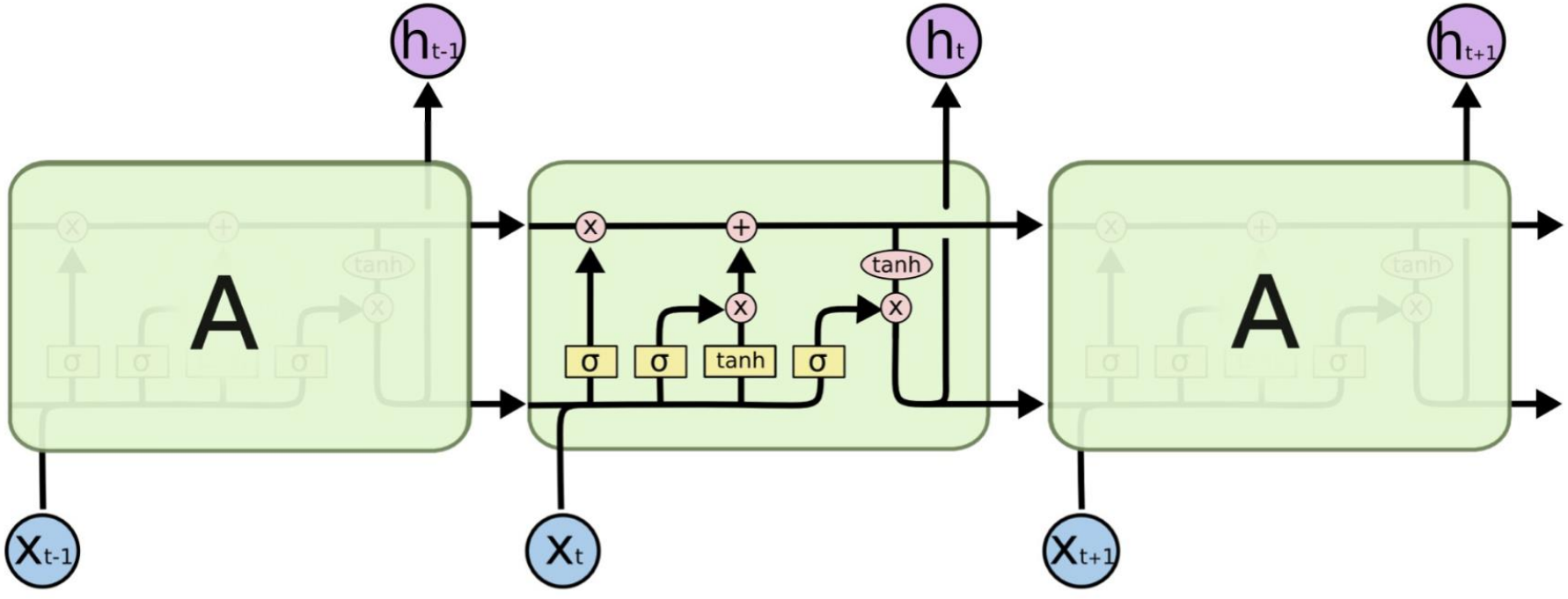
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

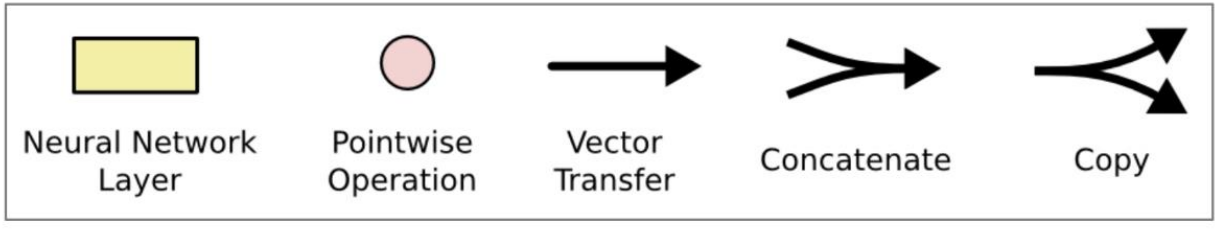
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

All these are vectors of same length $n$

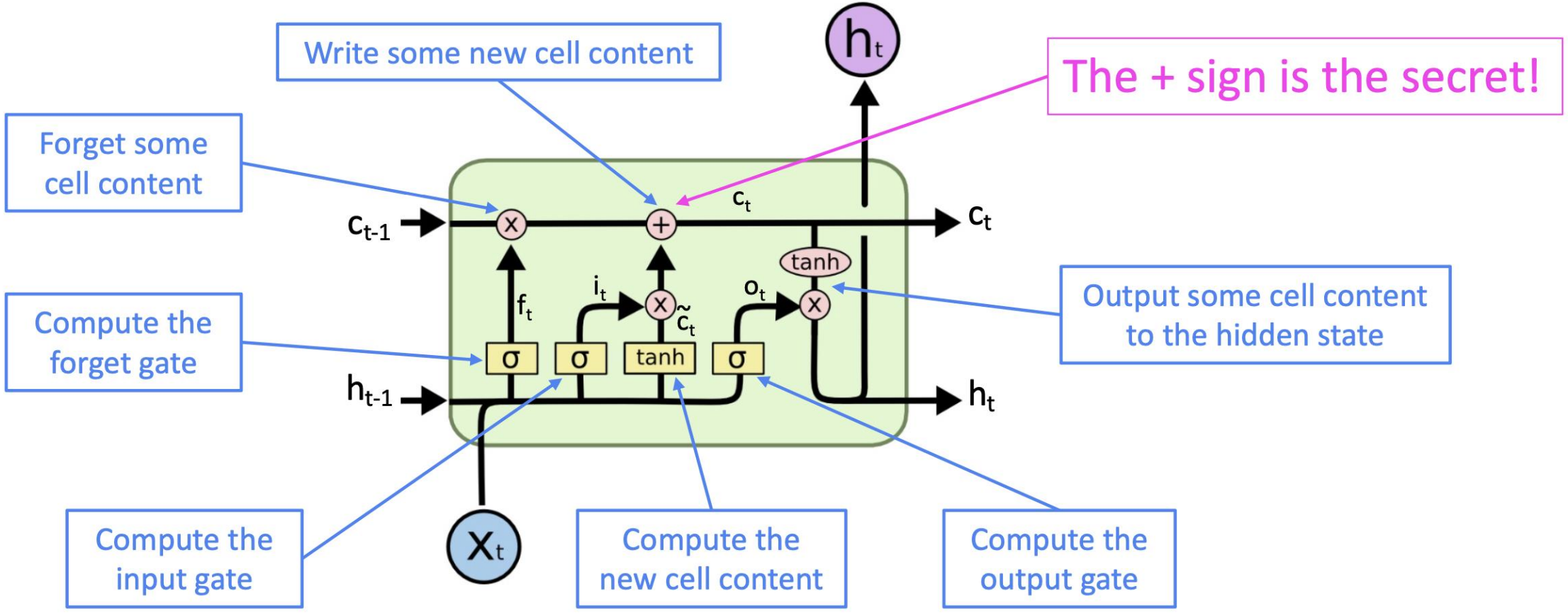
Gates are applied using element-wise (or Hadamard) product: $\odot$

# LSTM Architecture



# LSTM Architecture



# LSTM Architecture

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

- 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)

- 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

$$u^{(t)} = \sigma\left(W_u h^{(t-1)} + U_u x^{(t)} + b_u\right)$$

$$r^{(t)} = \sigma\left(W_r h^{(t-1)} + U_r x^{(t)} + b_r\right)$$

**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

$$\tilde{h}^{(t)} = \tanh\left(W_h(r^{(t)} \circ h^{(t-1)}) + U_h x^{(t)} + b_h\right)$$

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

**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)

# LSTM vs GRU

Let's compare LSTMs and GRUs. Which of the following statements is correct?

(a) GRUs can be trained faster

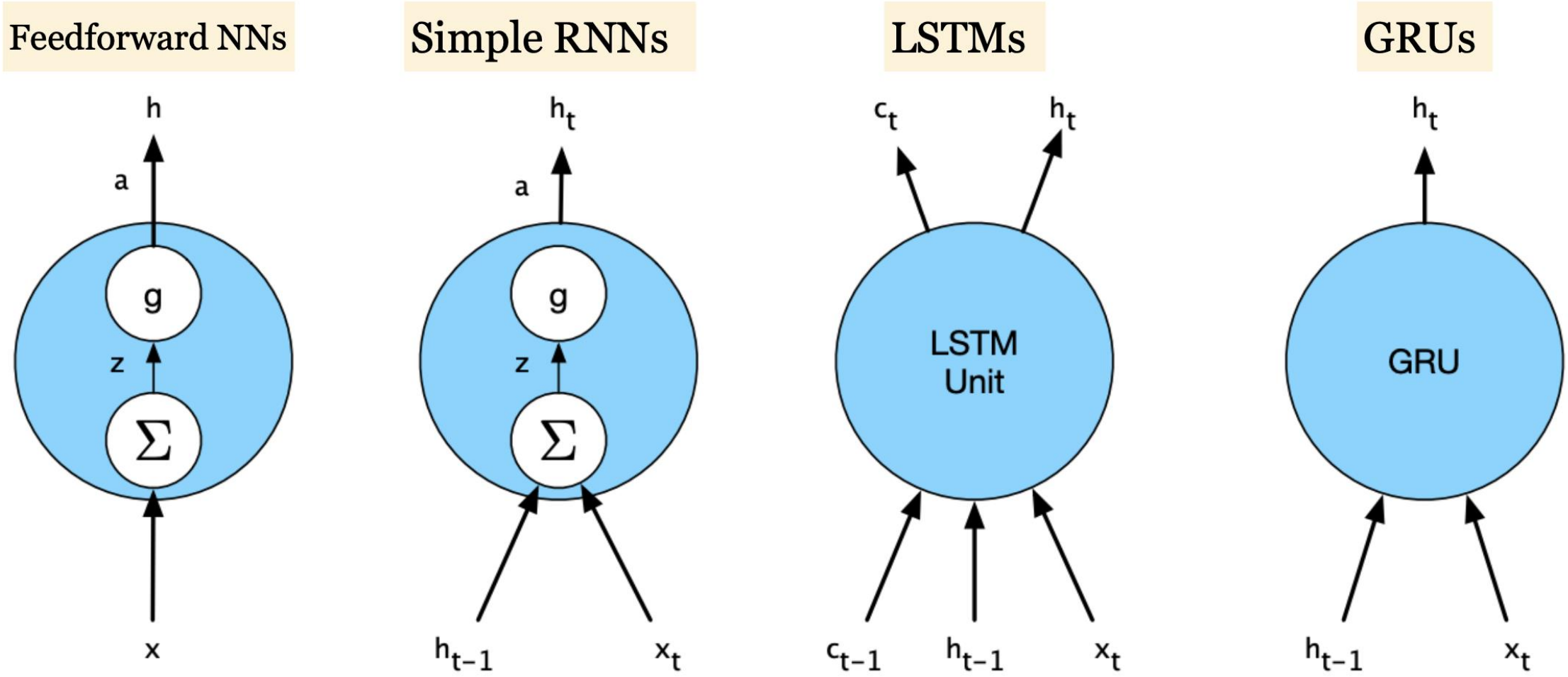
(b) In theory LSTMs can capture long-term dependencies better

(c) LSTMs have a controlled exposure of memory content while GRUs don't

(d) All of the above

The answer is (d). All of these are correct.

# FFNN vs RNN vs LSTM vs GRU



# Is Vanishing Gradient unique to RNN?

- No! It can be a problem for all neural architectures (including feed-forward and convolutional), especially very deep ones.
  - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
  - Thus, lower layers are learned very slowly (hard to train)
- Solution: lots of new deep feedforward/convolutional architectures add more direct connections (thus allowing the gradient to flow)

For example:

- Residual connections aka "ResNet"
- Also known as skip-connections
- The identity connection preserves information by default
- This makes deep networks much easier to train

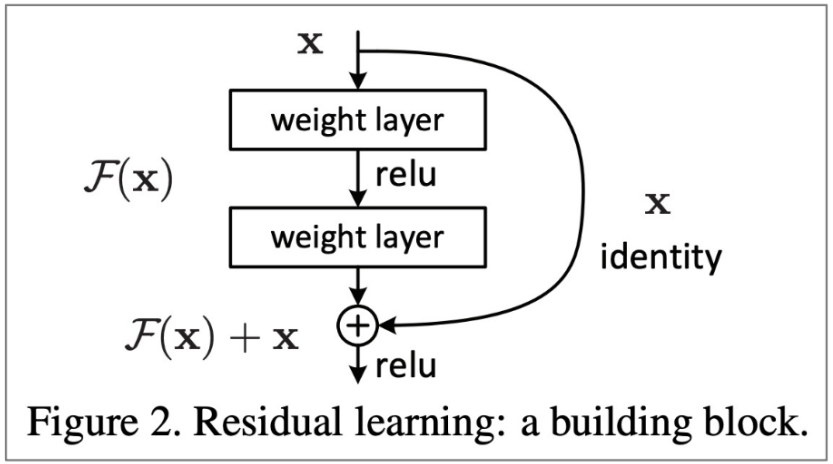


Figure 2. Residual learning: a building block.

# Is Vanishing Gradient unique to RNN?

Other methods:

- Dense connections aka "DenseNet"
- Directly connect each layer to all future layers!

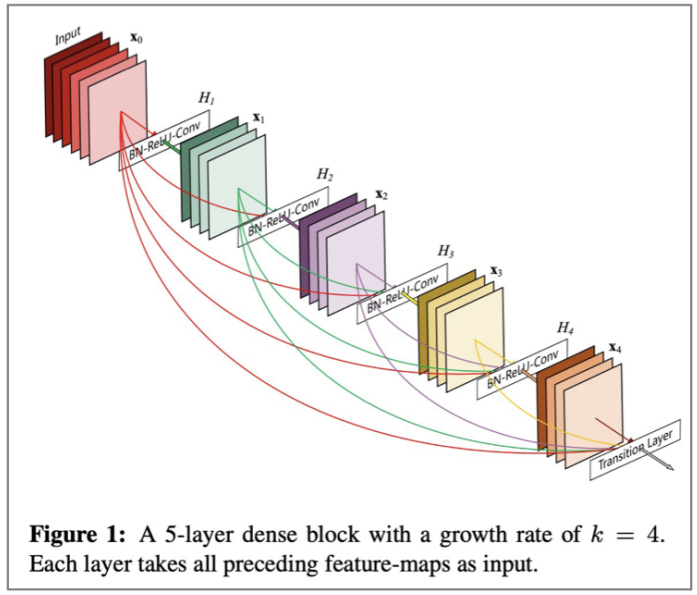
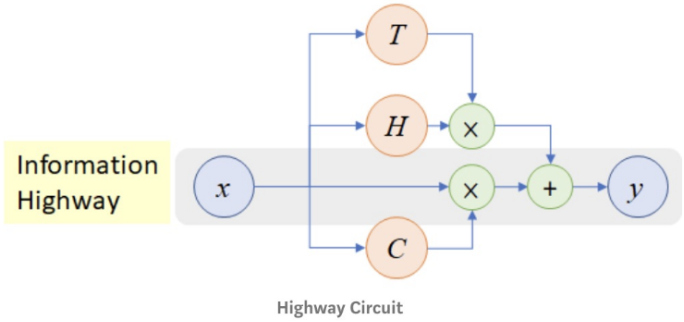



**Figure 1:** A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

- Highway connections aka "HighwayNet"
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks

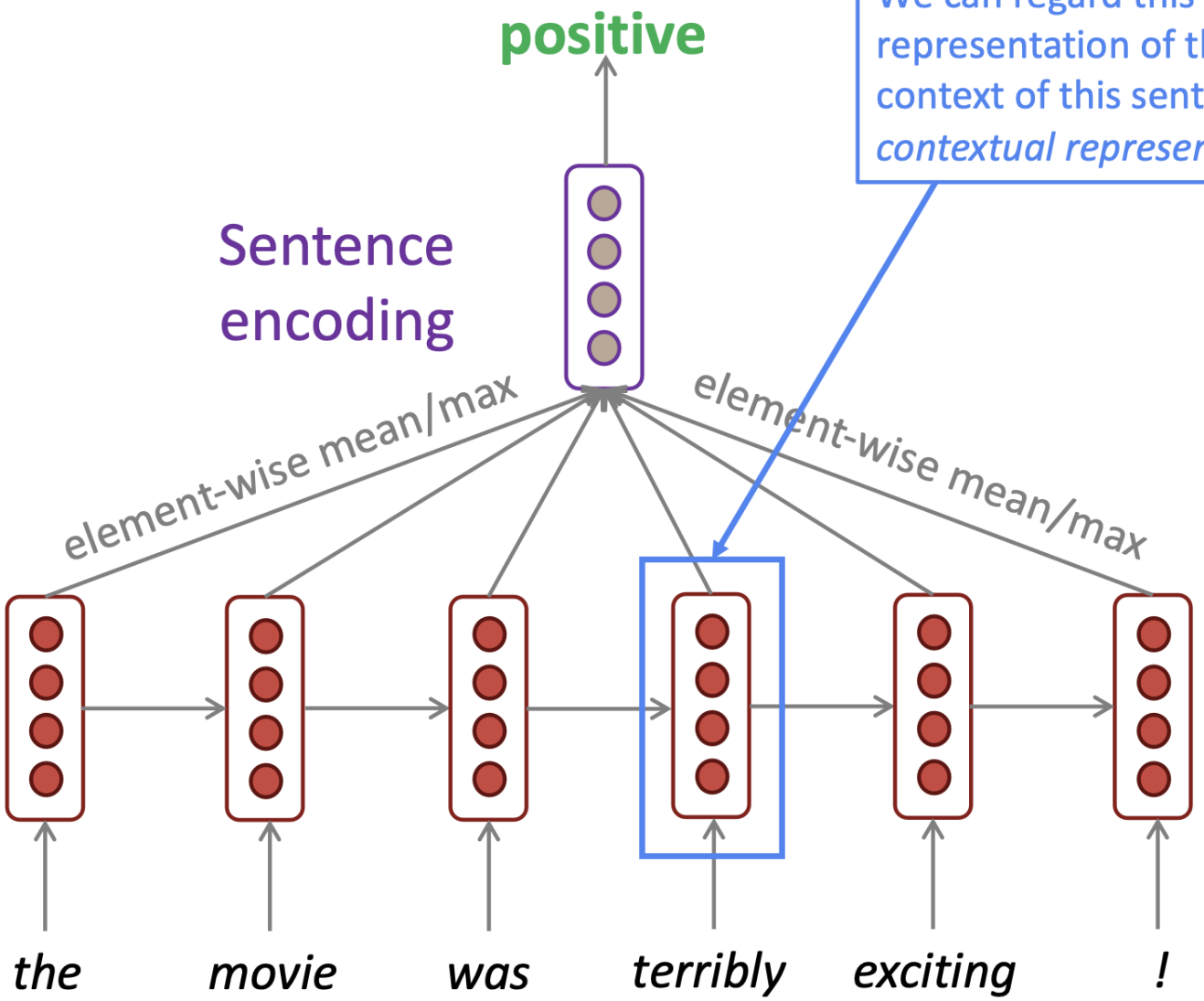



Highway Circuit

- **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]

# Other RNN Variants

# Bi-directional RNN



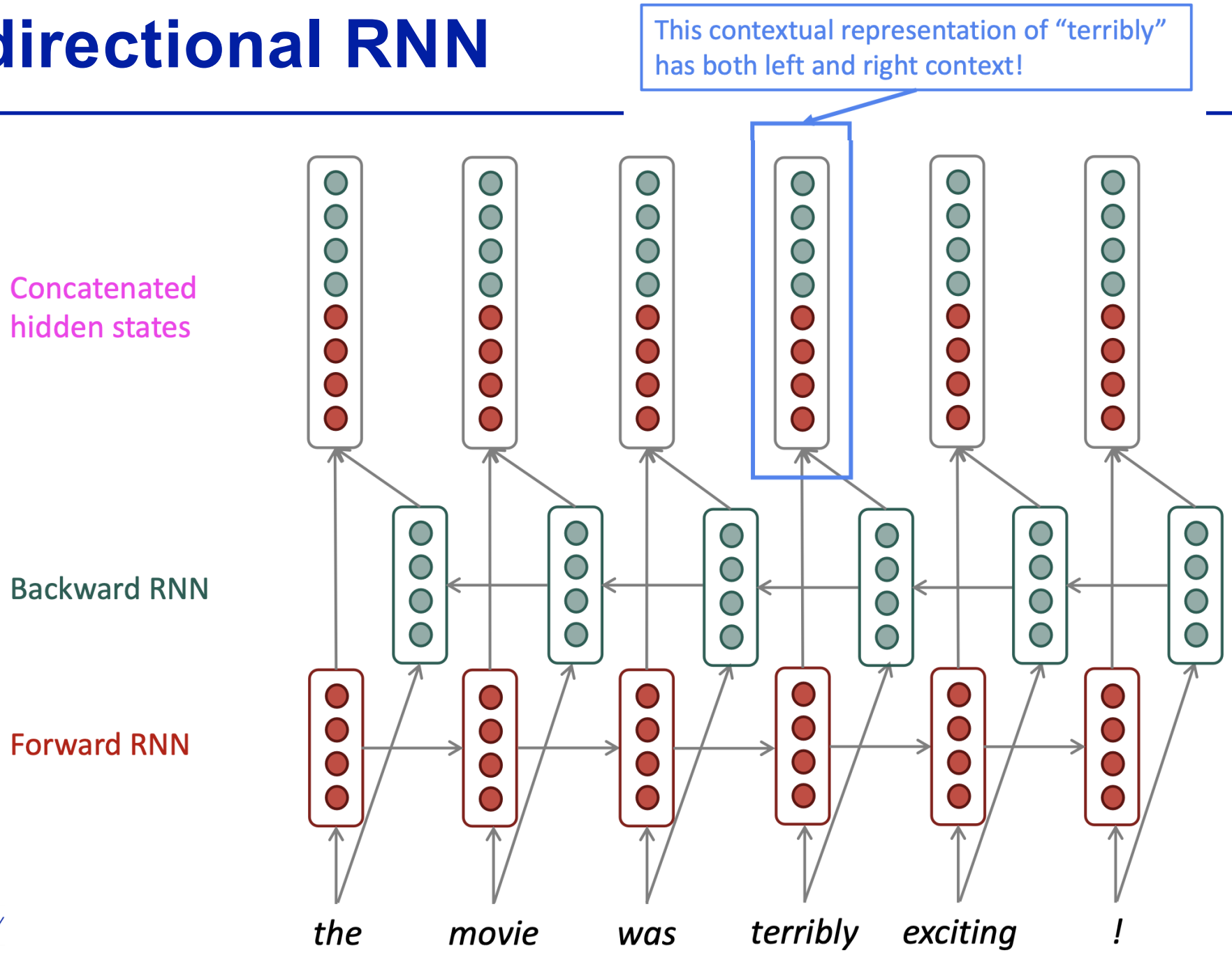

We can regard this hidden state as a representation of the word *"terribly"* in the context of this sentence. We call this a *contextual representation*.

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)

# Bi-directional RNN



# Bi-directional RNN

On timestep *t*:

This is a general notation to mean "compute one forward step of the RNN" – it could be a simple, LSTM, or other (e.g., GRU) RNN 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.

# Bi-directional RNN

- Bi-directional RNN is only applicable if you have access to the entire input sequence. If you do have entire input sequence, bidirectionality is powerful (you should use it by default).

- Bi-directional RNN is not applicable to Language Modeling, because in LM you only have left context available.

- Bi-directional RNN can be used as Encoder, not Decoder

# Practice of Bi-directional RNN

Can we use bidirectional RNNs in the following tasks?

(1) text classification, (2) sequence tagging, (3) text generation

(a) Yes, Yes, Yes

(b) Yes, No, Yes

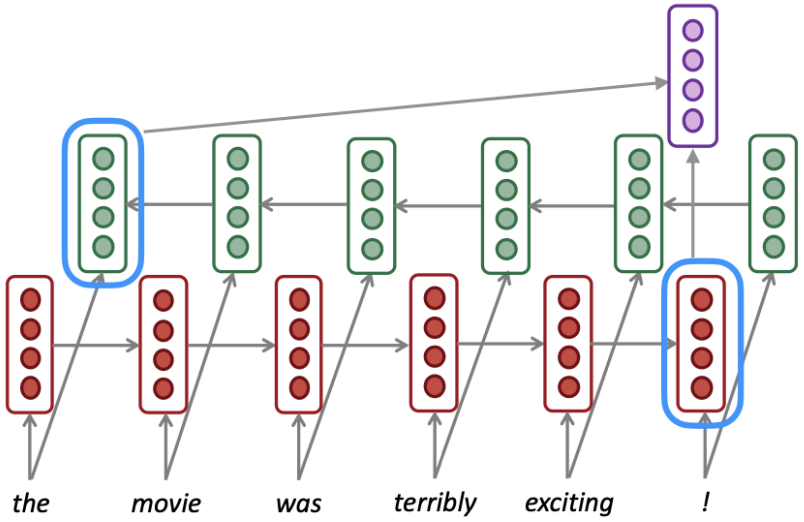
(c) Yes, Yes, No

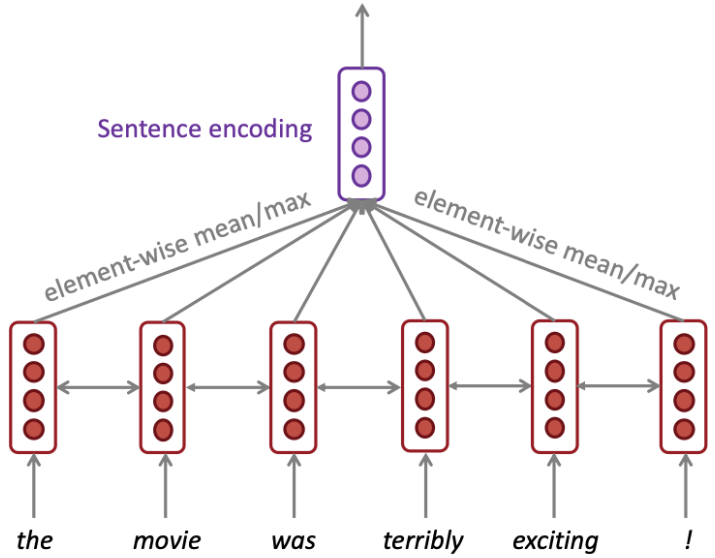
(d) No, Yes, No

The answer is (c).

# Bi-directional RNN

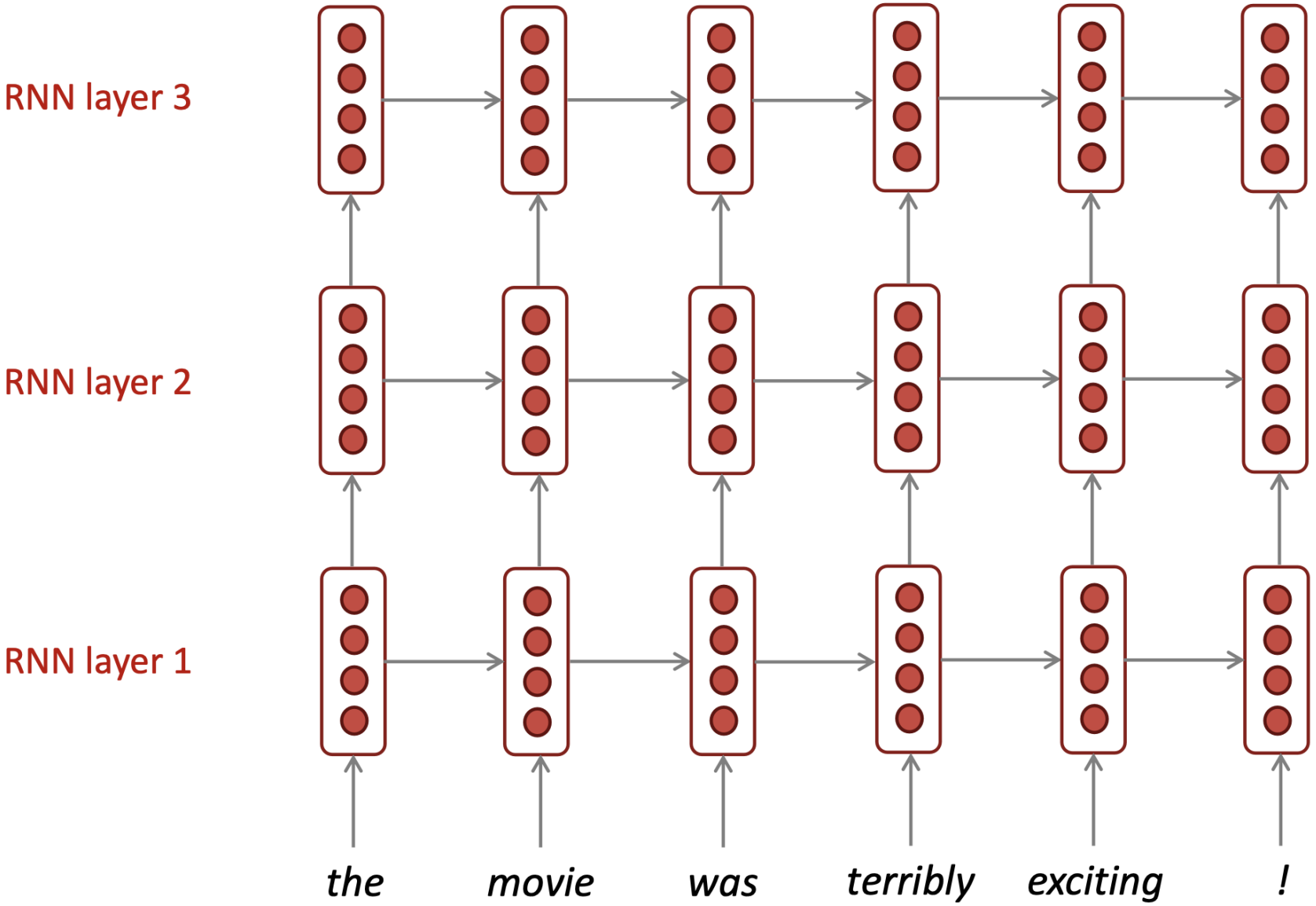
- Sequence tagging: Yes!
- Text classification: Yes!
  - Common practice: concatenate the last hidden vectors in two directions or take the mean/max over all the hidden vectors




- Text generation: No. Because we can't see the future to predict the next word.

# Multi-layer RNNs




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

# 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*.


Thank you!
UF
Herbert Wertheim
College of Engineering
UNIVERSITY of FLORIDA
LEADING THE CHARGE, CHARGING AHEAD