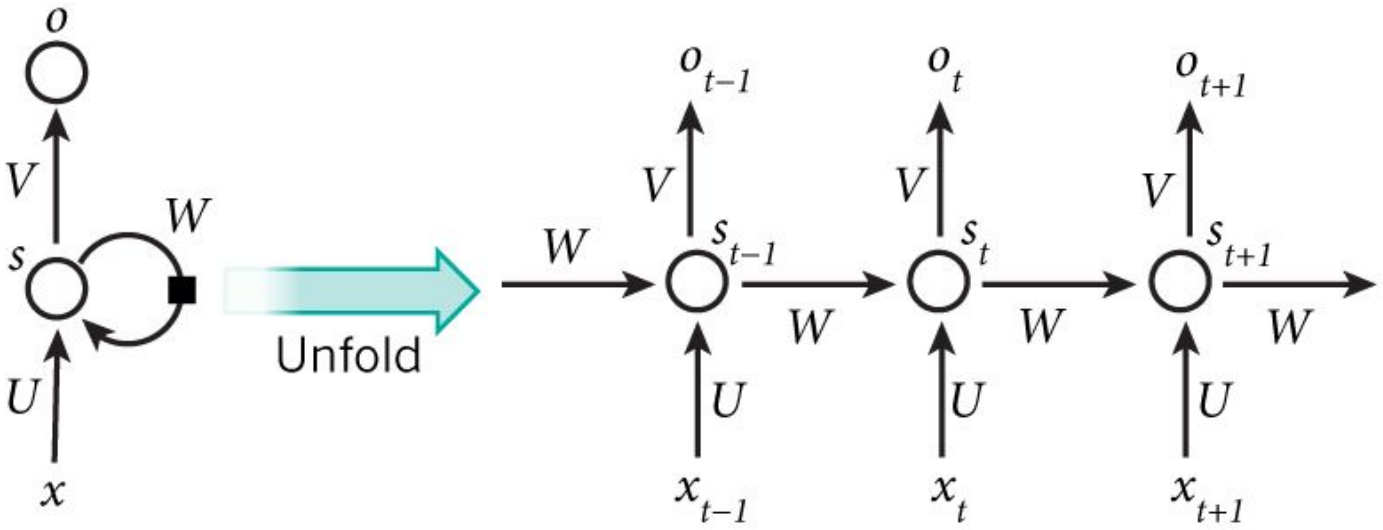


CS6101 - Deep Learning for NLP

Week 5: Recurrent Neural Networks & Language Models

Kee Yuan Chuan
Zhou Yizhuo
Wu Jiacheng
Liu Juncheng
Jin Zhe

Overview

Today we will:

- Introduce a new NLP task
  - **Language Modeling**

motivates

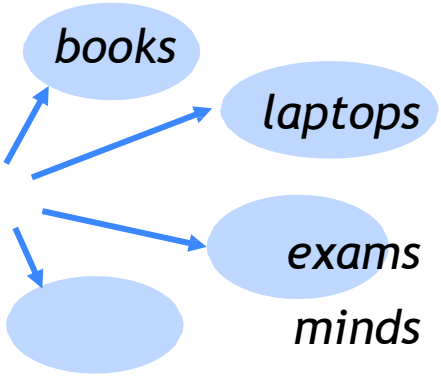
- Introduce a new family of neural networks
  - **Recurrent Neural Networks (RNNs)**

THE most important idea for the rest of the class!

Language Modeling

- **Language Modeling** is the task of predicting what word comes next

the students opened their



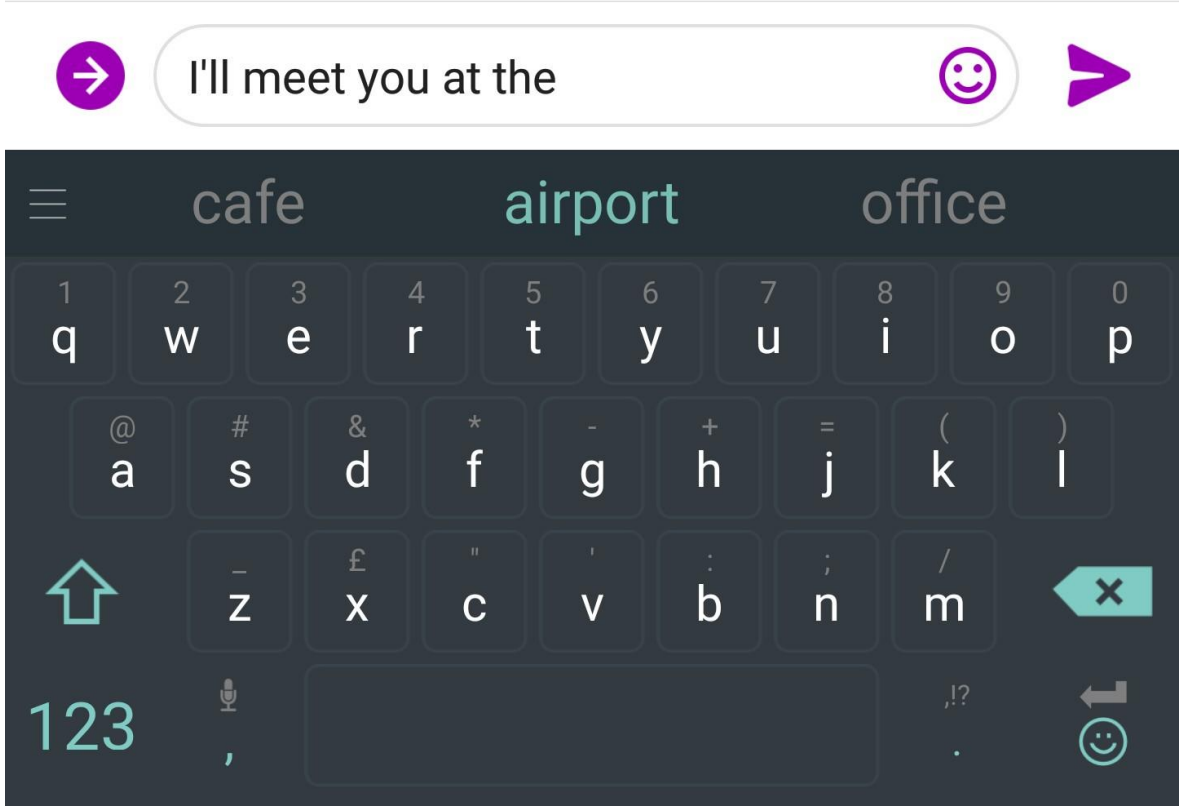
- More formally: given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$, compute the probability distribution of the next word $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

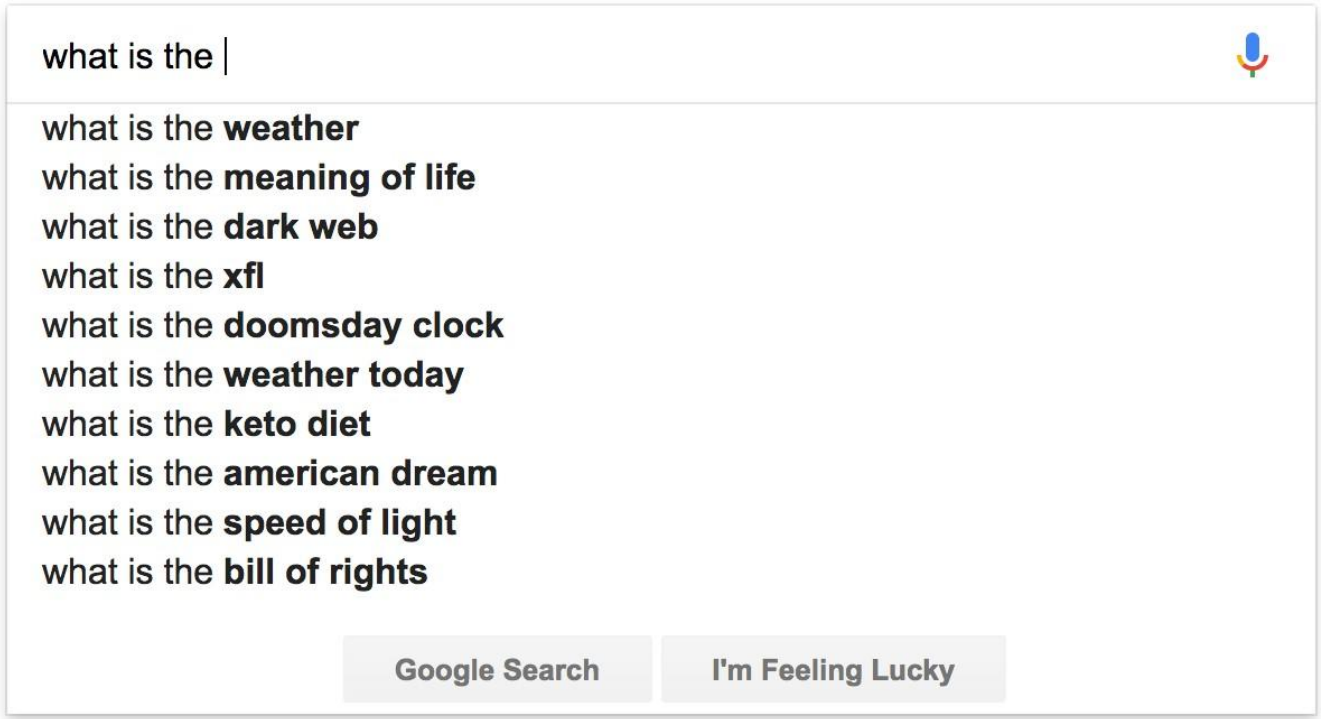
where $\boldsymbol{w}_j$ is a word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

- A system that does this is called a **Language Model**.

You use Language Models every day!



You use Language Models every day!



n-gram Language Models

the students opened their

- **Question**: How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn a *n*-gram Language Model!

- Definition: A *n*-gram is a chunk of *n* consecutive words.
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - 4-grams: “the students opened their”

- Idea: Collect statistics about how frequent different n-grams are, and use these to predict next word.

n-gram Language Models

- First we make a simplifying assumption: $x^{(t+1)}$ depends only on the preceding *(n-1)* words

$$P(x^{(t+1)}|x^{(t)},\ldots,x^{(1)}) = P(x^{(t+1)}|\overbrace{x^{(t)},\ldots,x^{(t-n+2)}}^{n\text{-1 words}}) \quad \text{(assumption)}$$

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{P(x^{(t+1)},x^{(t)},\ldots,x^{(t-n+2)})}{P(x^{(t)},\ldots,x^{(t-n+2)})}$$

(definition of conditional prob)

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(x^{(t+1)},x^{(t)},\ldots,x^{(t-n+2)})}{\text{count}(x^{(t)},\ldots,x^{(t-n+2)})}$$

(statistical approximation)

Why is it an approximation?

$P(x^{(t+1)}, x^{(t)} \ldots, x^{(t-n+2)})$ is the joint probability of the words sequence

*"Out of sequence of length **n**, how many of them is the sequence of interest?"*

Recall conditional probability of 2 random variables, ***x*** and a known ***y*** is

$$P(X|Y = y_i) = \frac{P(X, Y = y_i)}{P(Y = y_i)}$$

Rearranging $P(X, Y) = P(X|Y = y_i)P(Y = y_i)$ aka chain rule

Generalising $P(X_1 \ldots X_n) = \prod_{k=1}^{n} P(X_k|X_1^{k-1})$

$$P(\text{“book”}, \text{“their”}, \text{“opened”}, \text{“students”})$$
$$= P(\text{“book”}|\text{“their”}, \text{“opened”}, \text{“students”})P(\text{“their”}|\text{“opened”}, \text{“students”})P(\text{“opened”}|\text{“students”})P(\text{“students”})$$

https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf

n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~as the proctor started the clock, the~~ *students opened their*

discard

condition on this

$$P(\boldsymbol{w}_j|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w}_j)}{\text{count}(\text{students opened their})}$$

In the corpus:

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times
  - → P(books | students opened their) = 0.4
- "students opened their exams" occurred 100 times
  - → P(exams | students opened their) = 0.1

Should we have discarded the "proctor" context?

Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if "*students opened their* w_j " never occurred in data? Then w_j has probability 0!

(Partial) Solution: Add small δ to count for every $w_j \in V$. This is called *smoothing*.

$$P(w_j|\text{students opened their}) = \frac{\text{count}(\text{students opened their } w_j)}{\text{count}(\text{students opened their})}$$

Sparsity Problem 2

Problem: What if "*students opened their*" never occurred in data? Then we can't calculate probability for *any* w_j !

(Partial) Solution: Just condition on "*opened their*" instead. This is called *backoff*.

Note: Increasing *n* makes sparsity problems *worse*. Typically we can't have *n* bigger than 5.

Problems with n-gram Language Models

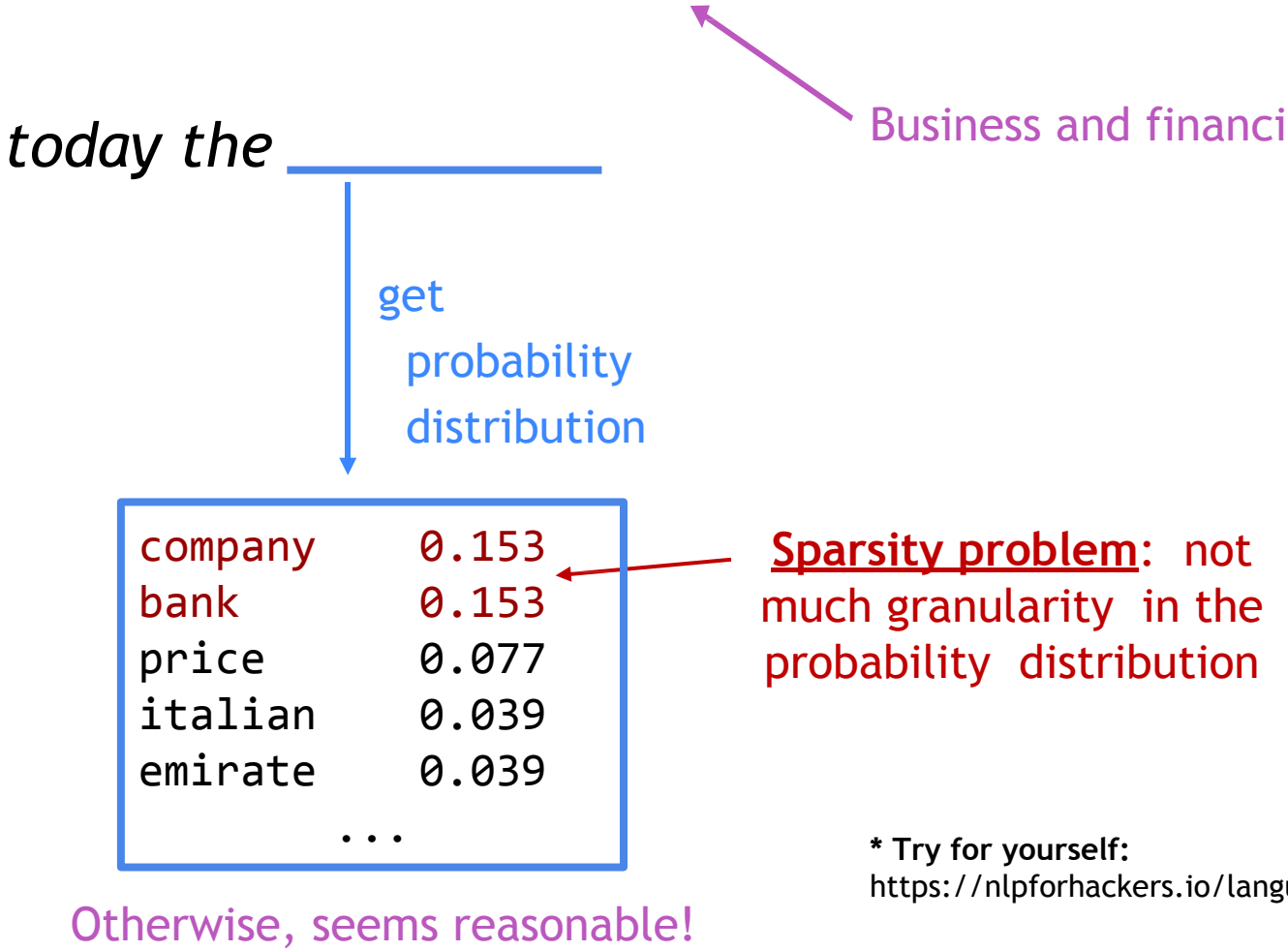
Storage: Need to store count for all possible *n*-grams. So model size is O(exp(*n*)).

$$P(\boldsymbol{w}_j|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w}_j)}{\text{count}(\text{students opened their})}$$

Increasing *n* makes model size huge!

n-gram Language Models in practice

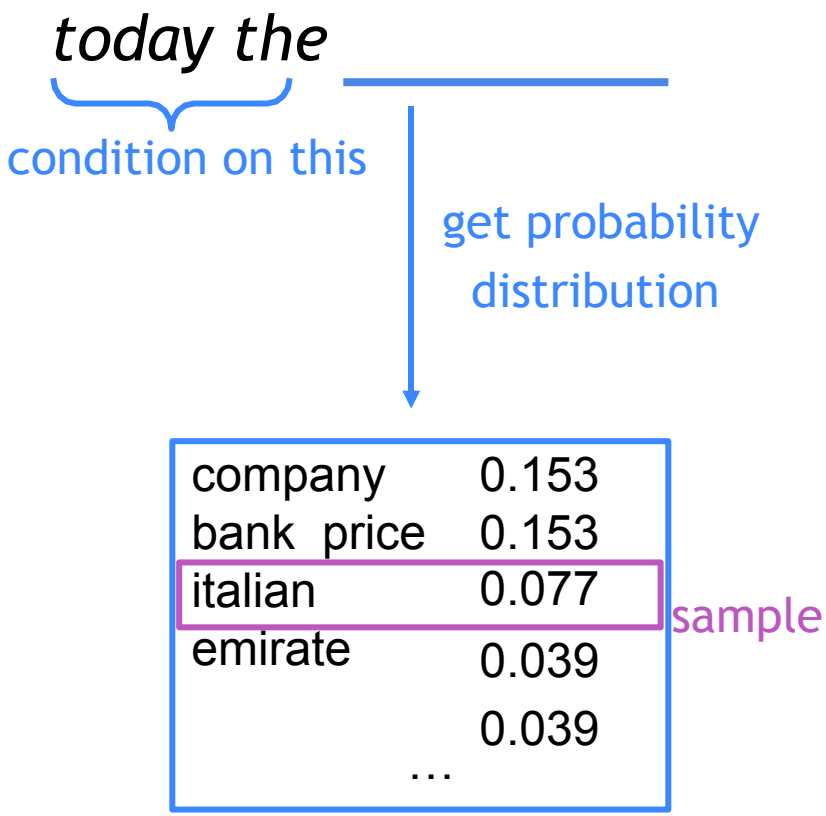
- You can build a simple trigram Language Model over a 1.7 million word corpus (Reuters) in a few seconds on your laptop*



*** Try for yourself:**
https://nlpforhackers.io/language-models/

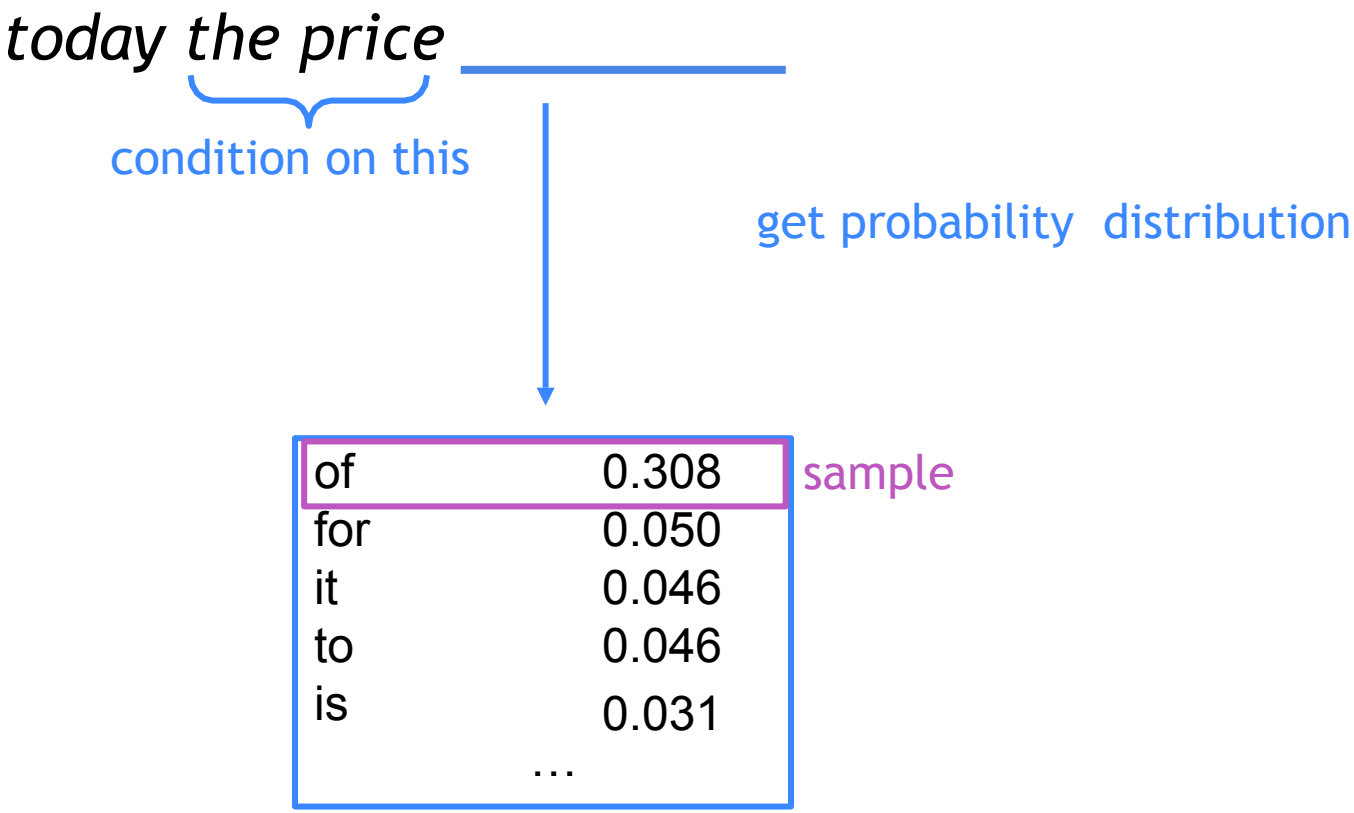
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.



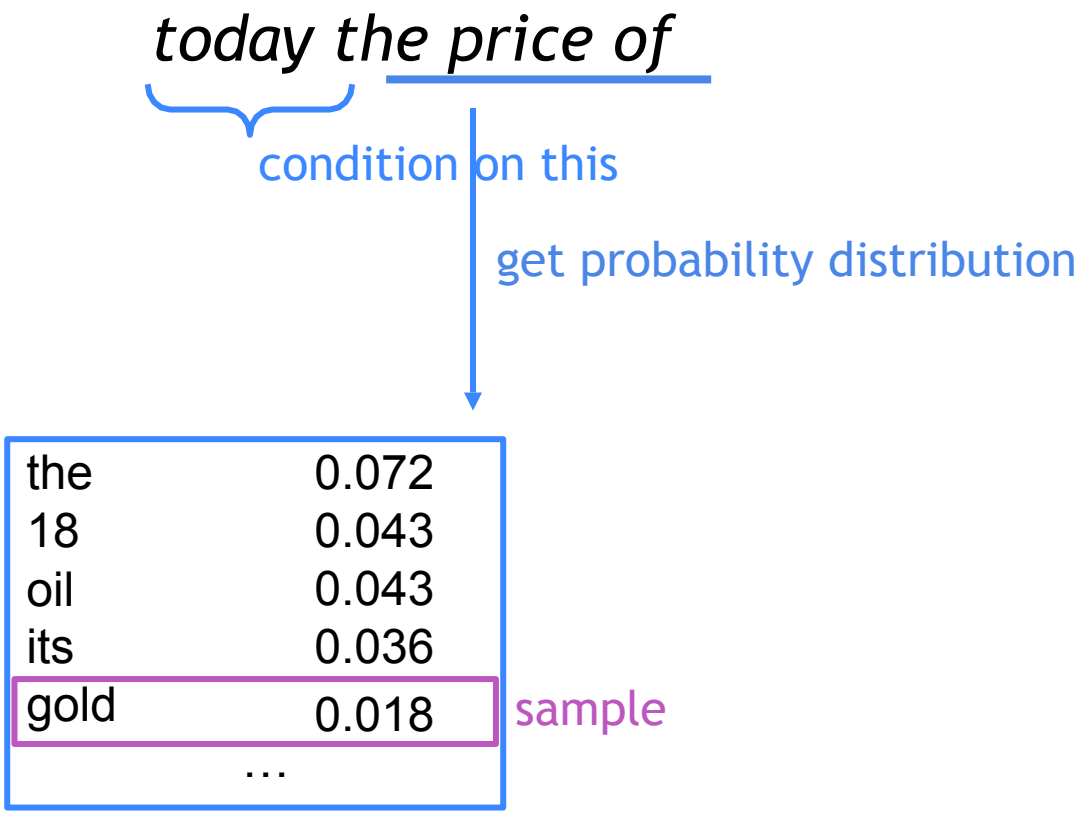
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.



Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.



Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

today the price of gold _____

Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

Incoherent! We need to consider more than 3 words at a time if we want to generate good text.

But increasing *n* worsens sparsity problem, and exponentially increases model size…

How to build a *neural* Language Model?

- Recall the Language Modeling task:
  - Input: sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$
  - Output: prob dist of the next word $P(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$

- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition in Lecture 4

A fixed-window neural Language Model

~~as the proctor started the clock~~ the students opened their _____

discard

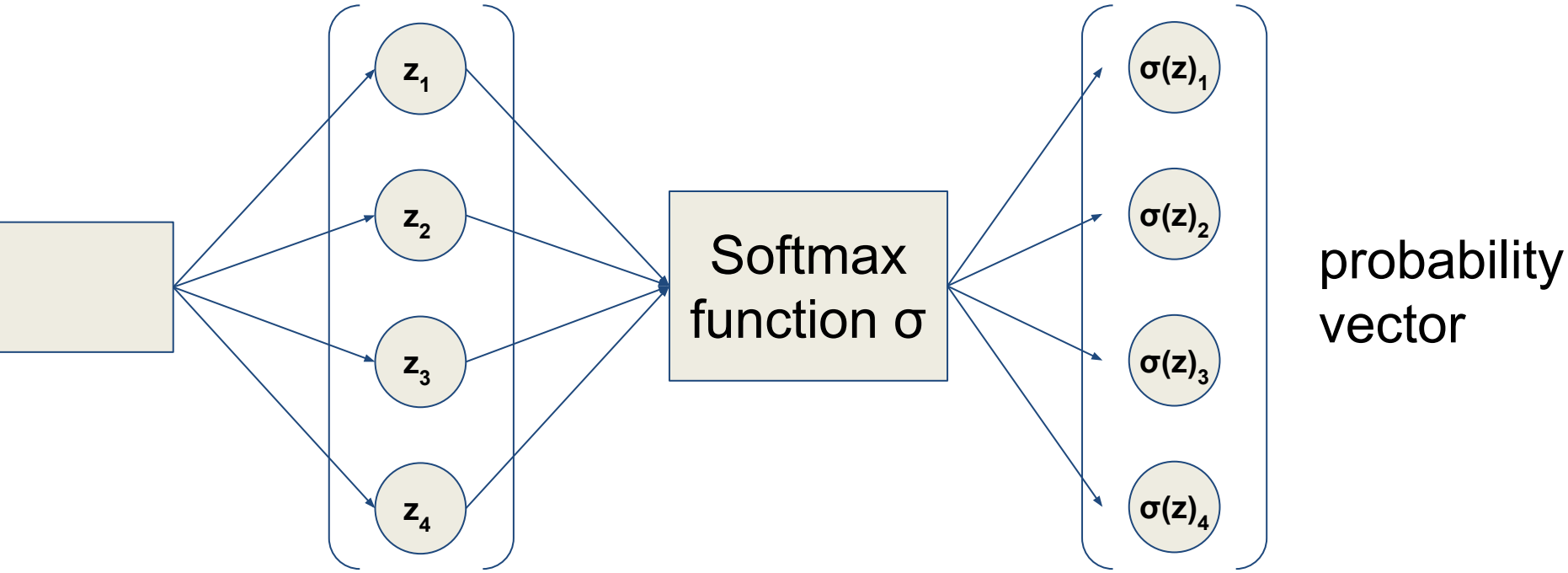
fixed window

Softmax

- A.K.A *normalized exponential function*
- is a generalization of the *logistic function* AKA *Sigmoid function*
- Used to provide a probabilistic interpretation of classification prediction
- More appropriate for **mutually-exclusive classes** because during training, values of correct classes pushed towards +ive inf, values of wrong classes pushed towards -ive inf ⇒ Only one correct class, all other classes are wrong!

Softmax

Say we have 4 classes, so final layer fan-out is 4



$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for \; j = 1, \ldots, k$$

Softmax

Question: Why do we use output values as exponent to the natural log base?

Hint: Think about the value range of layer output values

Softmax

In practice...

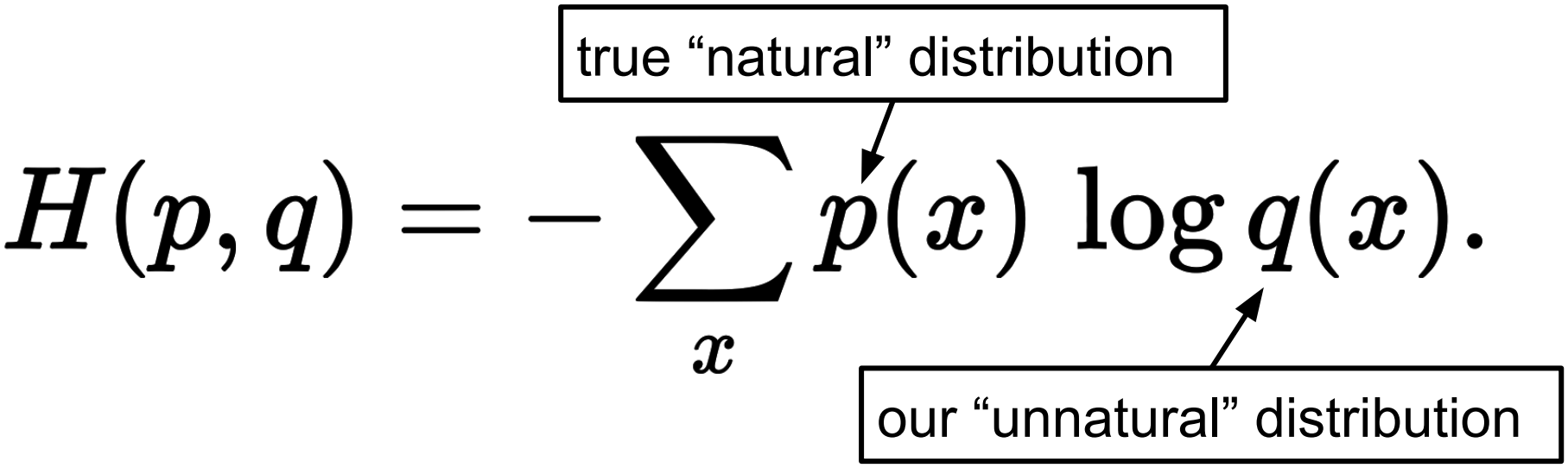
$$
\begin{aligned}
p_i &= \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\
&= \frac{Ce^{a_i}}{C\sum_{k=1}^{N} e^{a_k}} \\
&= \frac{e^{a_i+\log(C)}}{\sum_{k=1}^{N} e^{a_k+\log(C)}}
\end{aligned}
$$

Where ***log(C)*** is taken to be ***–max(a)***. Why?

Cross Entropy

- A measure of distance between what the model believes the output distribution should be (“unnatural” distribution), and the original distribution (“natural” distribution)

For discrete outcomes:



Cross Entropy

$$H(p, q) = -\sum_{x} p(x) \log q(x).$$

Say for a given input, the correct class is 3 out of classes {1,2,3,4}

$$-\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \bullet \begin{pmatrix} 0.23 \\ 0.01 \\ 0.67 \\ 0.09 \end{pmatrix} = -1 \times 0.67 = -0.67$$

Cross Entropy

$$H(p, q) = -\sum_{x} p(x) \log q(x).$$

$$= \text{-log } q(x)$$

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

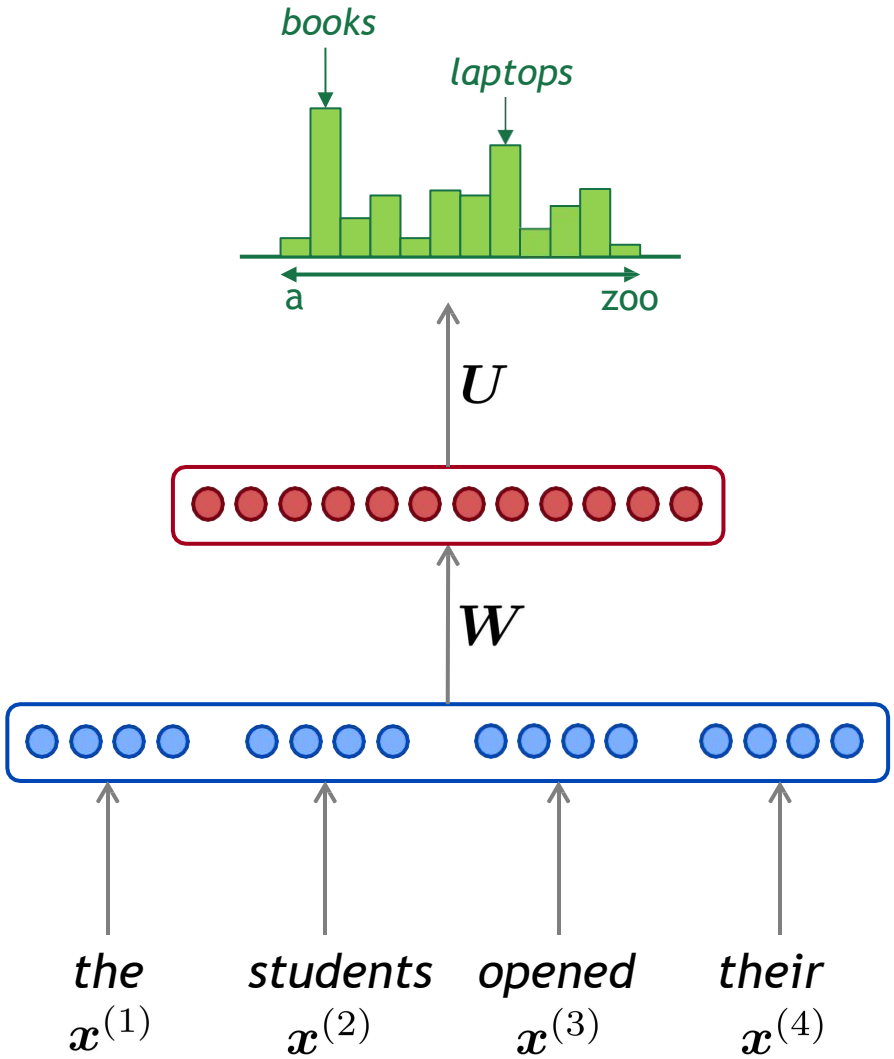
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

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

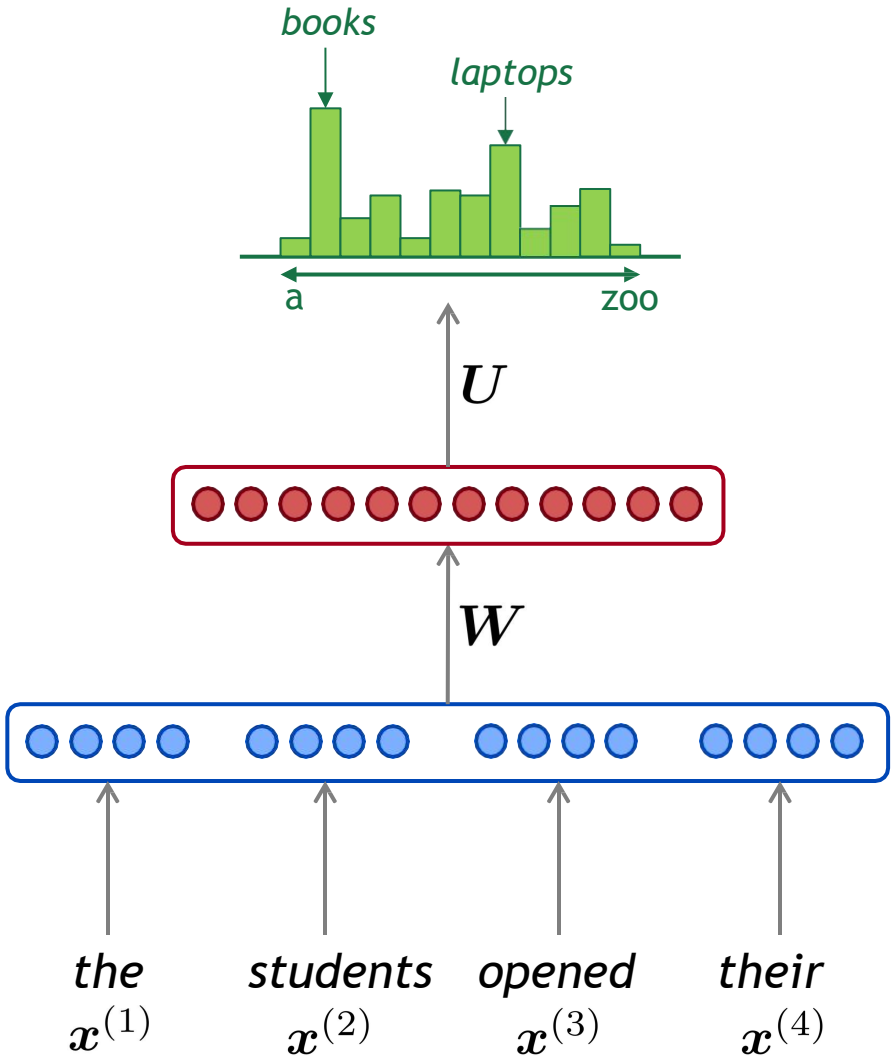
Improvements over *n*-gram LM:

- No sparsity problem
- Model size is O(*n*) not O(exp(*n*))

Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- Each $x^{(i)}$ uses different ~~rows~~ cols of W. We don't share weights across the window.

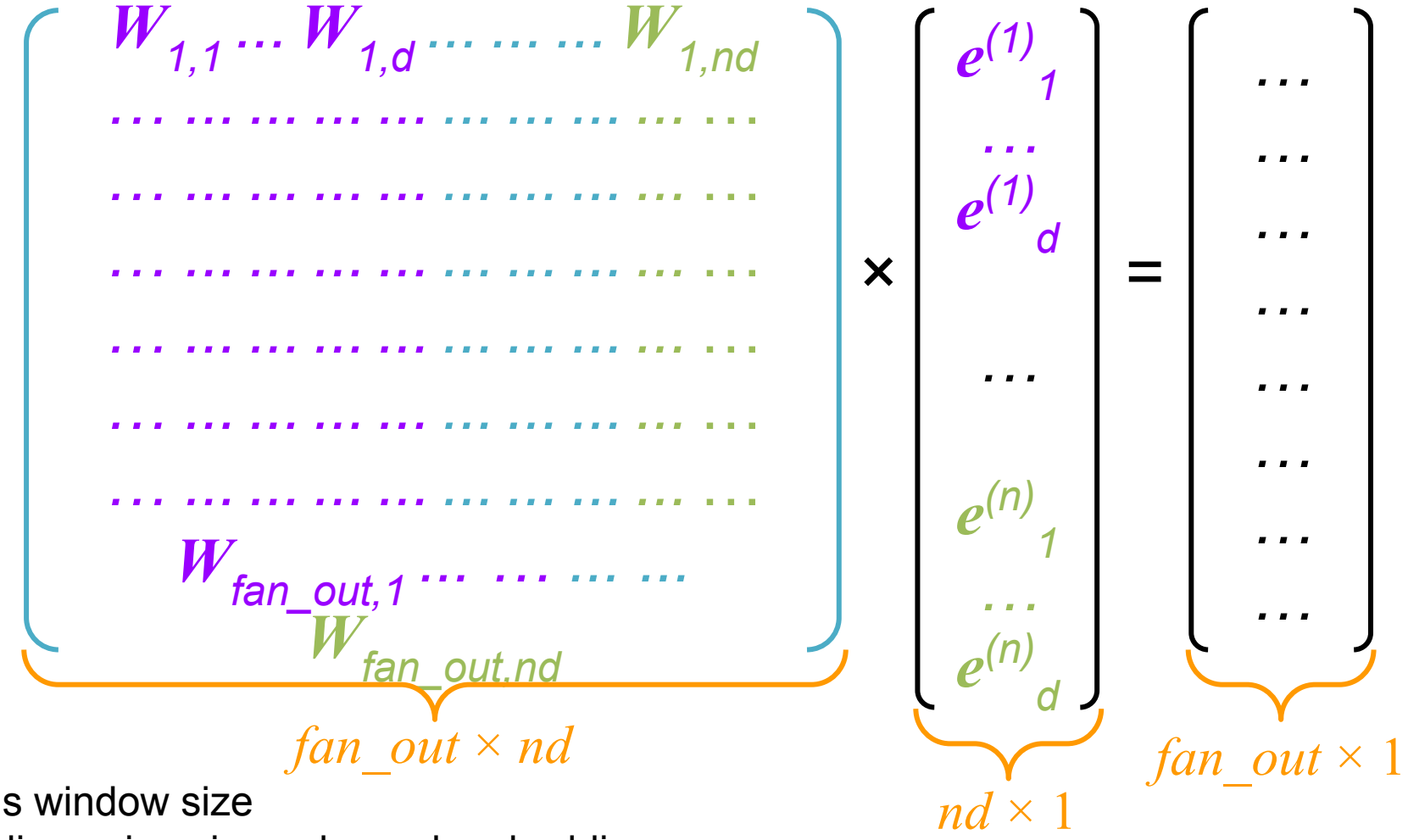
We need a neural architecture that can process *any length input*



A fixed-window neural Language Model

How does our weight matrix look like?

$$W \times e =$$



n is window size

d dimensions in each word embedding

A fixed-window neural Language Model

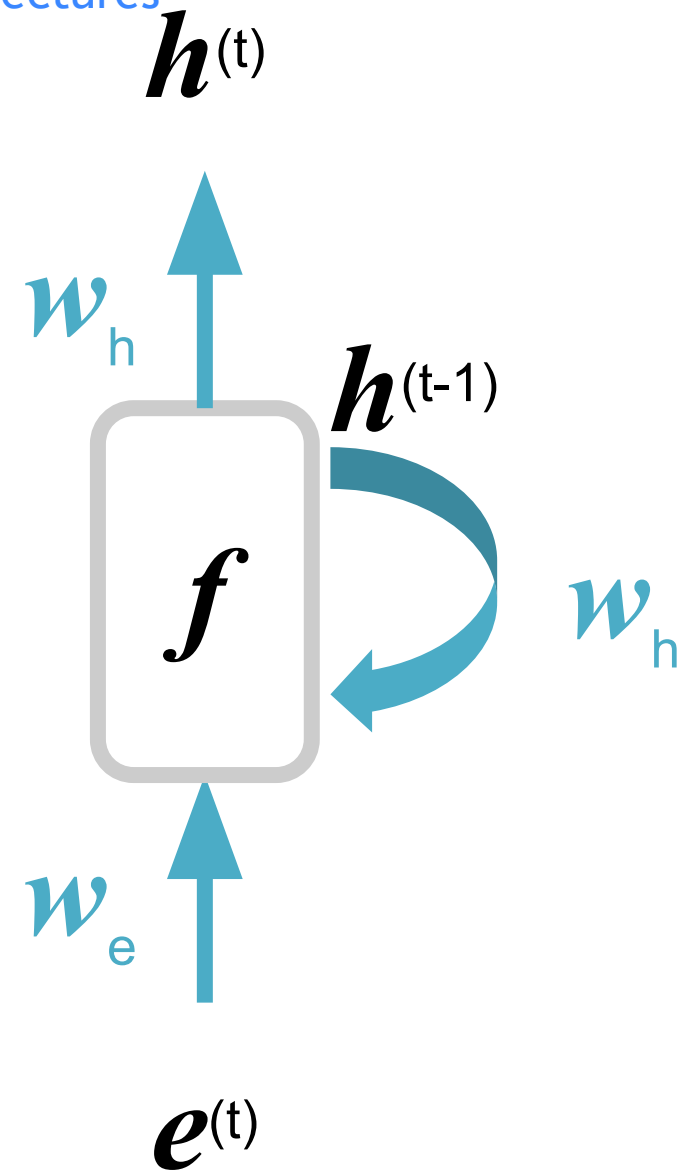
Question: Why do we want to share weights across the window?

Why do we want to share weights across windows?

- Else the number of weights would grow linearly with the number of time steps (our window size) like a feed-forward network
- We want to capture shared representations across sequences of text

Recurrent Neural Networks (RNN)

A family of neural architectures

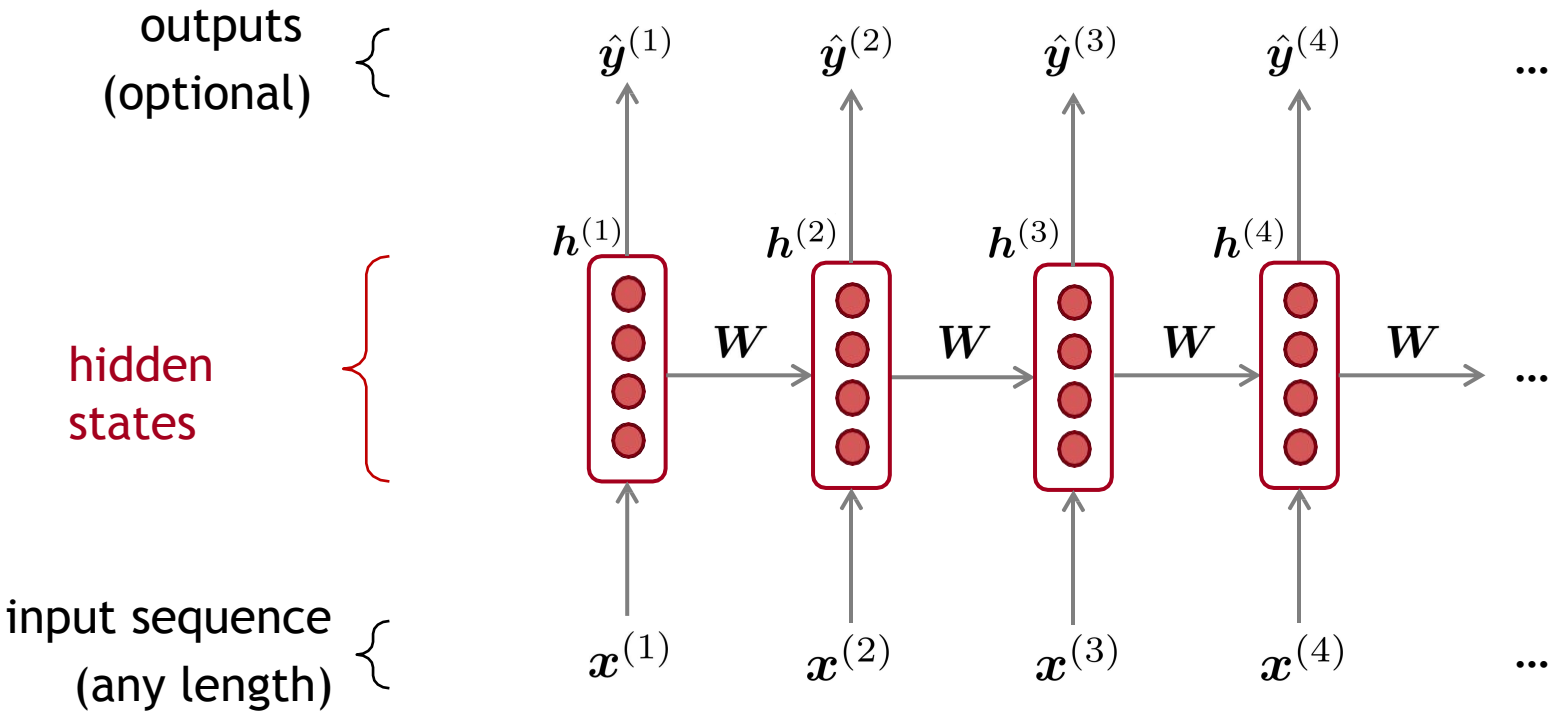


Recurrent Neural Networks (RNN)

A family of neural architectures

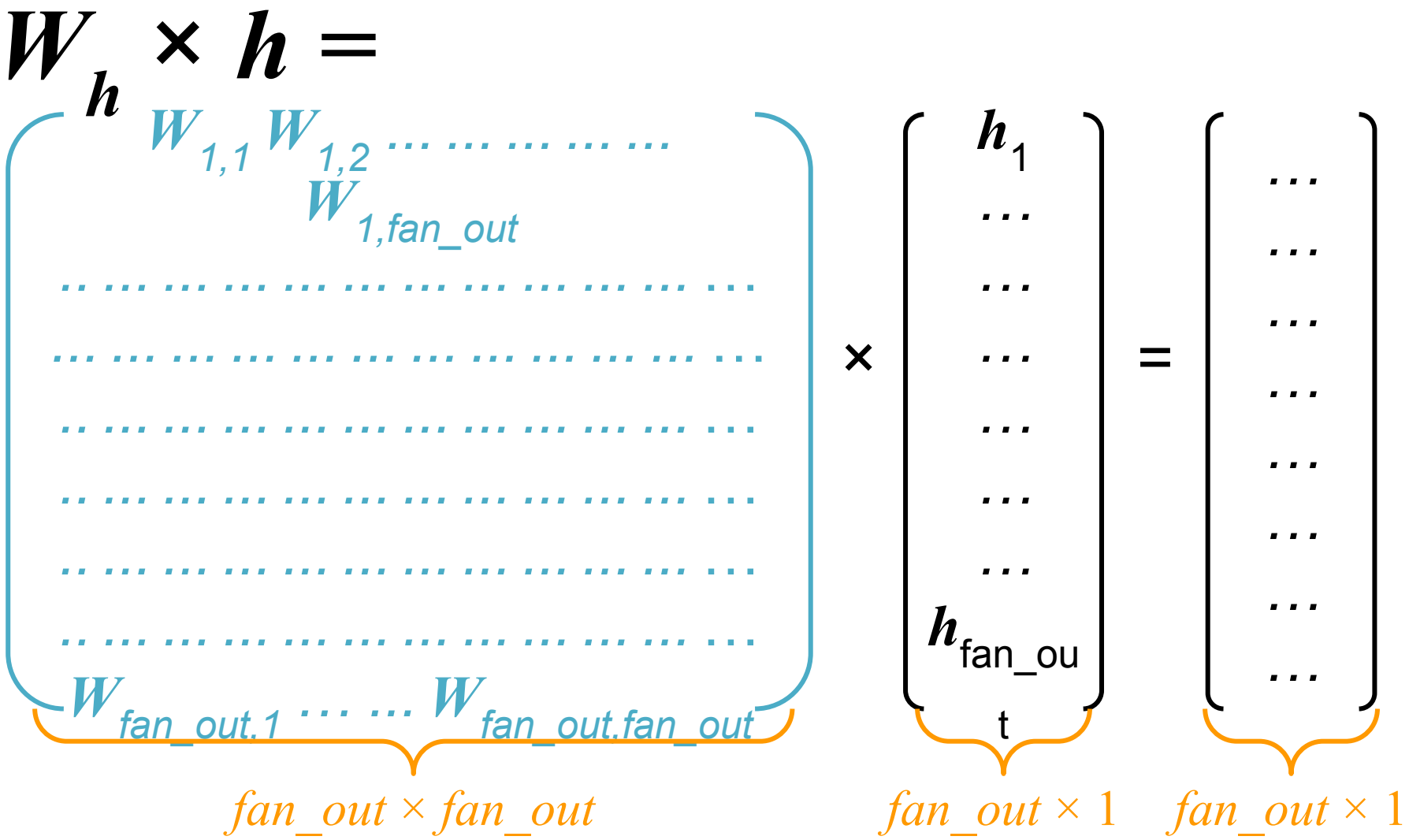
Core idea: Apply the same weights $\boldsymbol{W}$ *repeatedly*

visualized as feedforward networks "unrolled across time":



Recurrent Neural Networks (RNN)

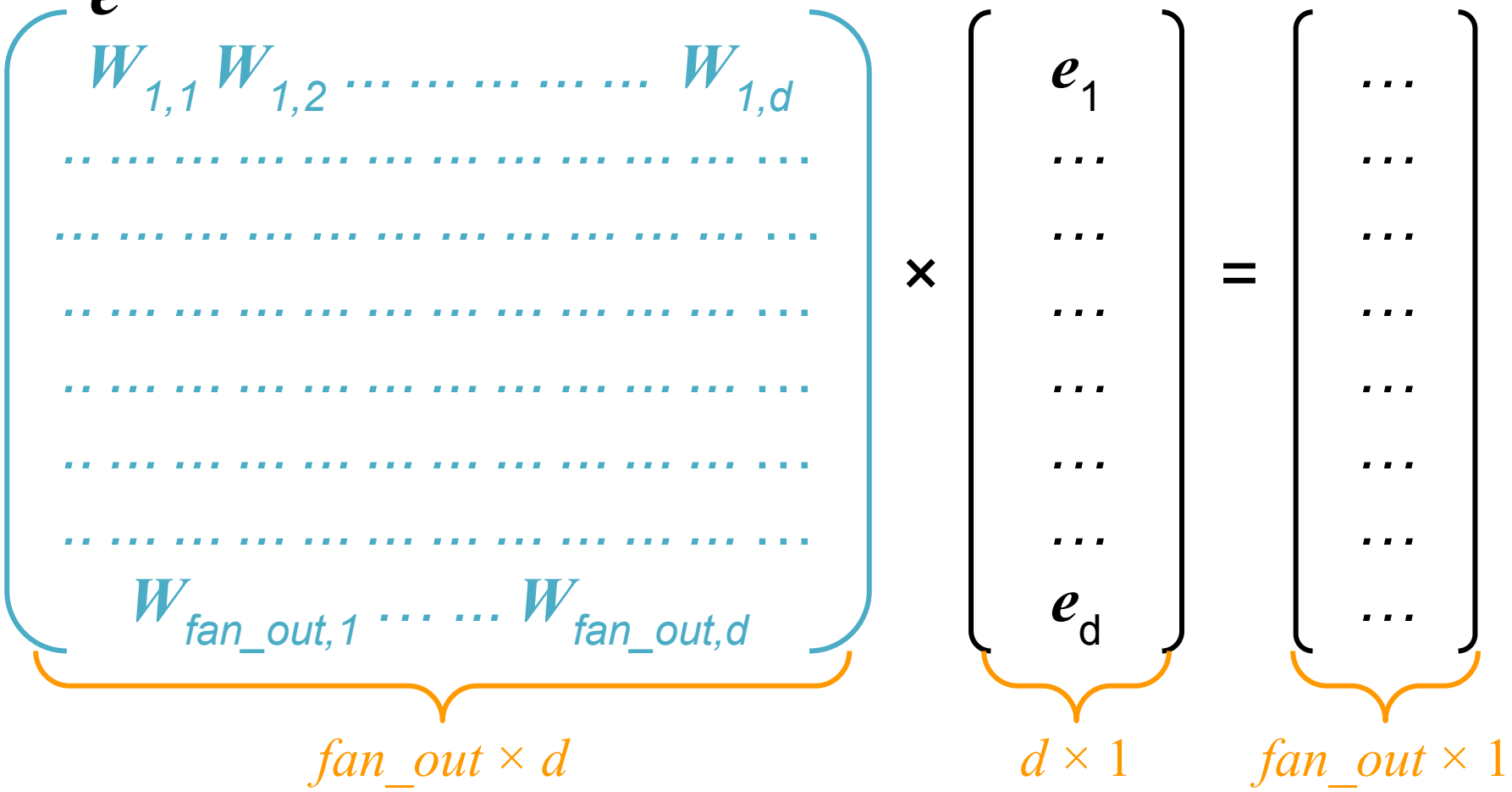
How does our weight matrix W_h look like?



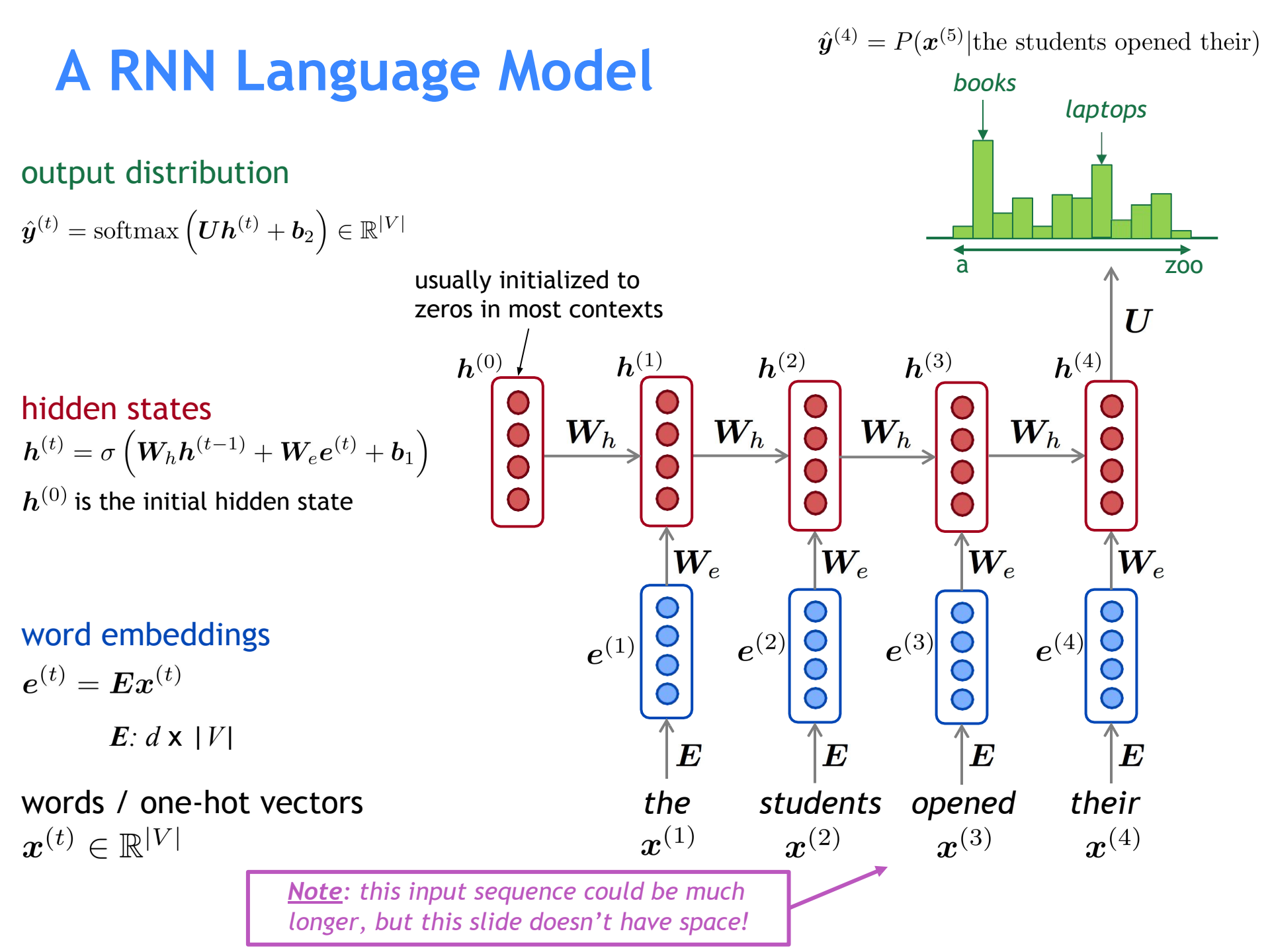
Recurrent Neural Networks (RNN)

How does our weight matrix W_h look like?

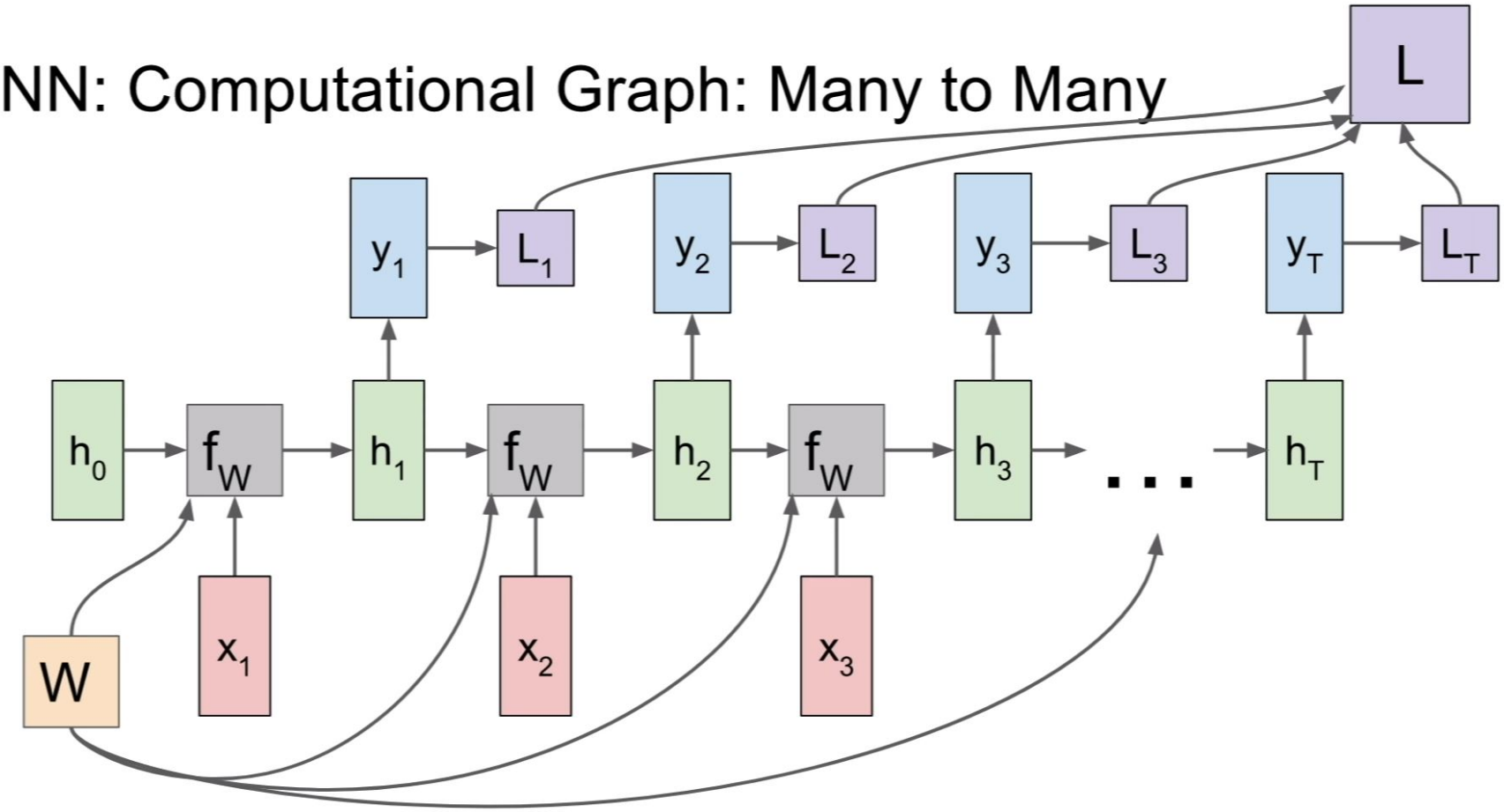
$$W_e \times e =$$



A RNN Language Model



RNN: Computational Graph: Many to Many



A RNN Language Model

Question: So what is *recurrent* about it?

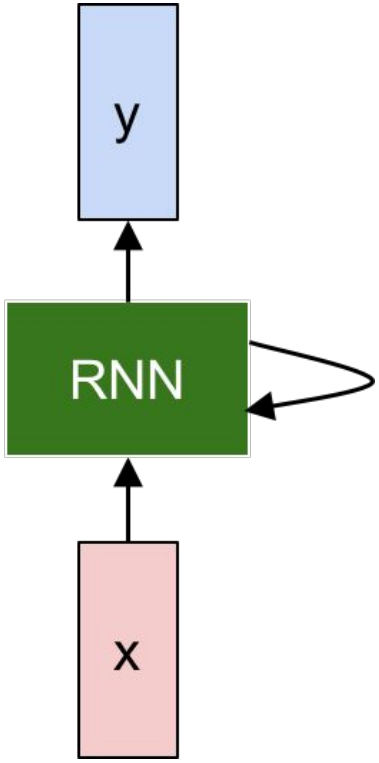
$$h_t = f_W(h_{t-1}, x_t)$$

new state

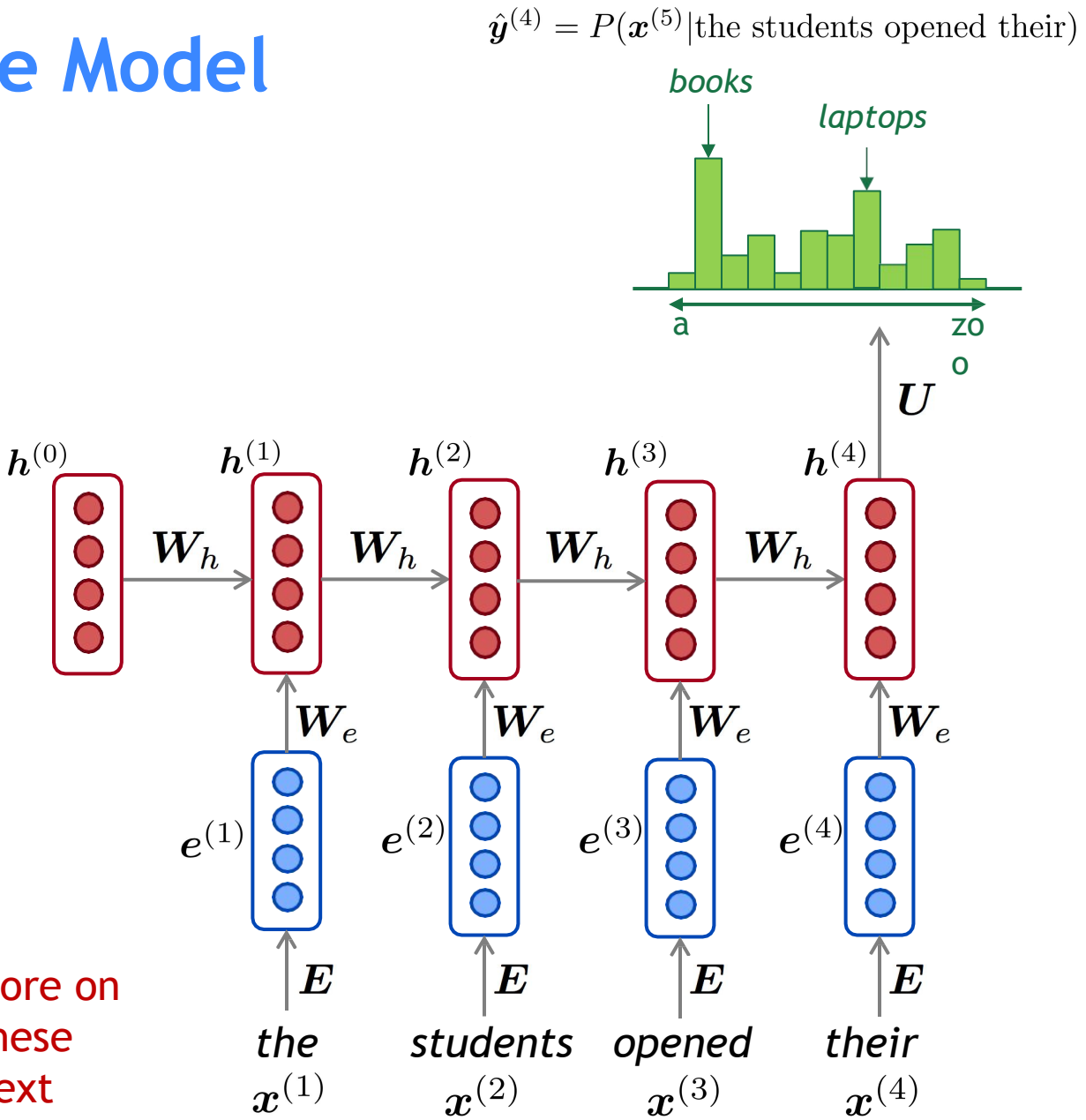
some function
with parameters W

old state

input vector at
some time step



A RNN Language Model



RNN **Advantages**:

- Can process any length input
- Model size doesn't increase for longer input
- Computation for step *t* can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared

RNN **Disadvantages**:

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

More on these next week

Training a RNN Language Model

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

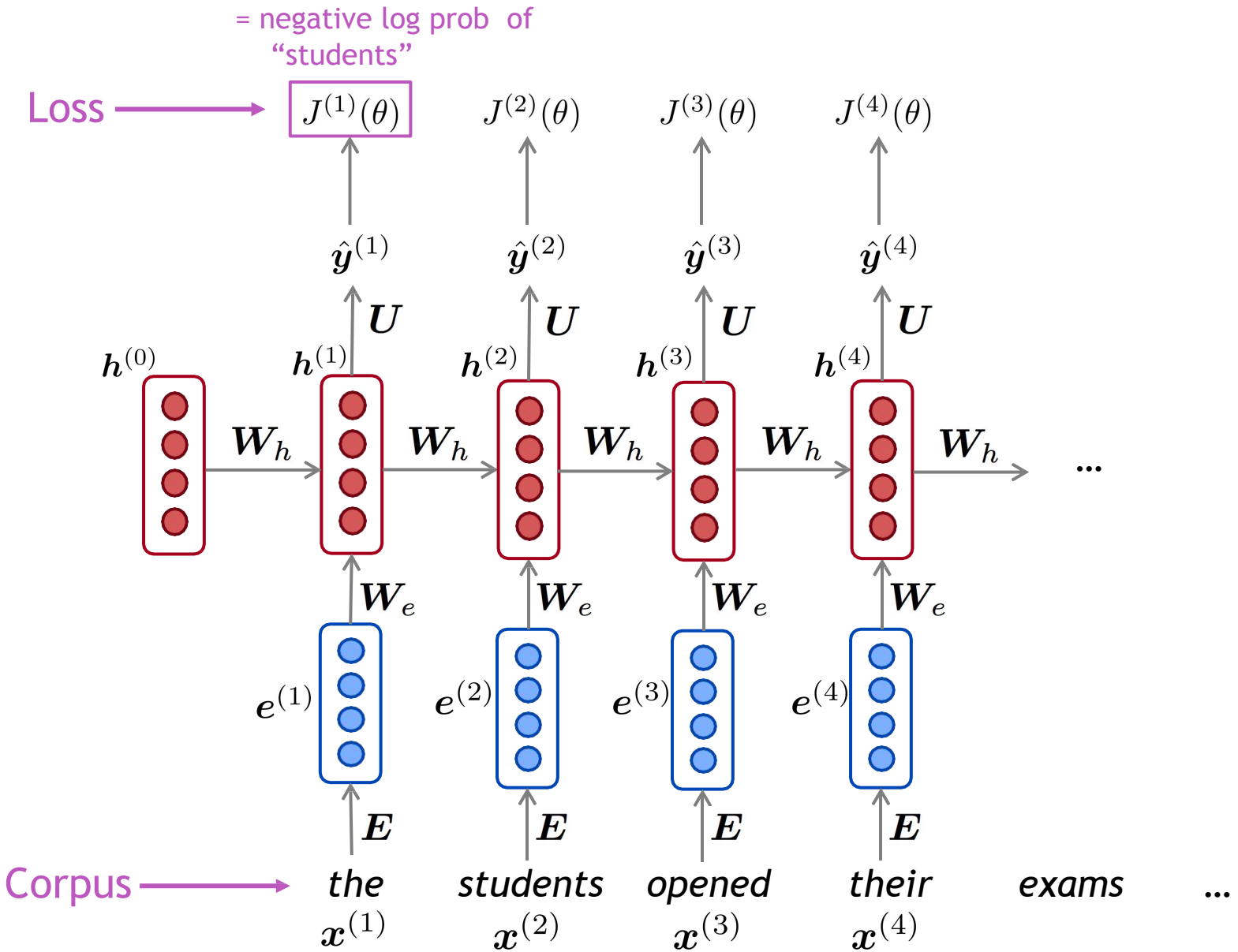
- Loss function on step *t* is usual cross-entropy (CE) between our predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word : $\boldsymbol{y}^{(t)} = \boldsymbol{x}^{(t+1)}$

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$

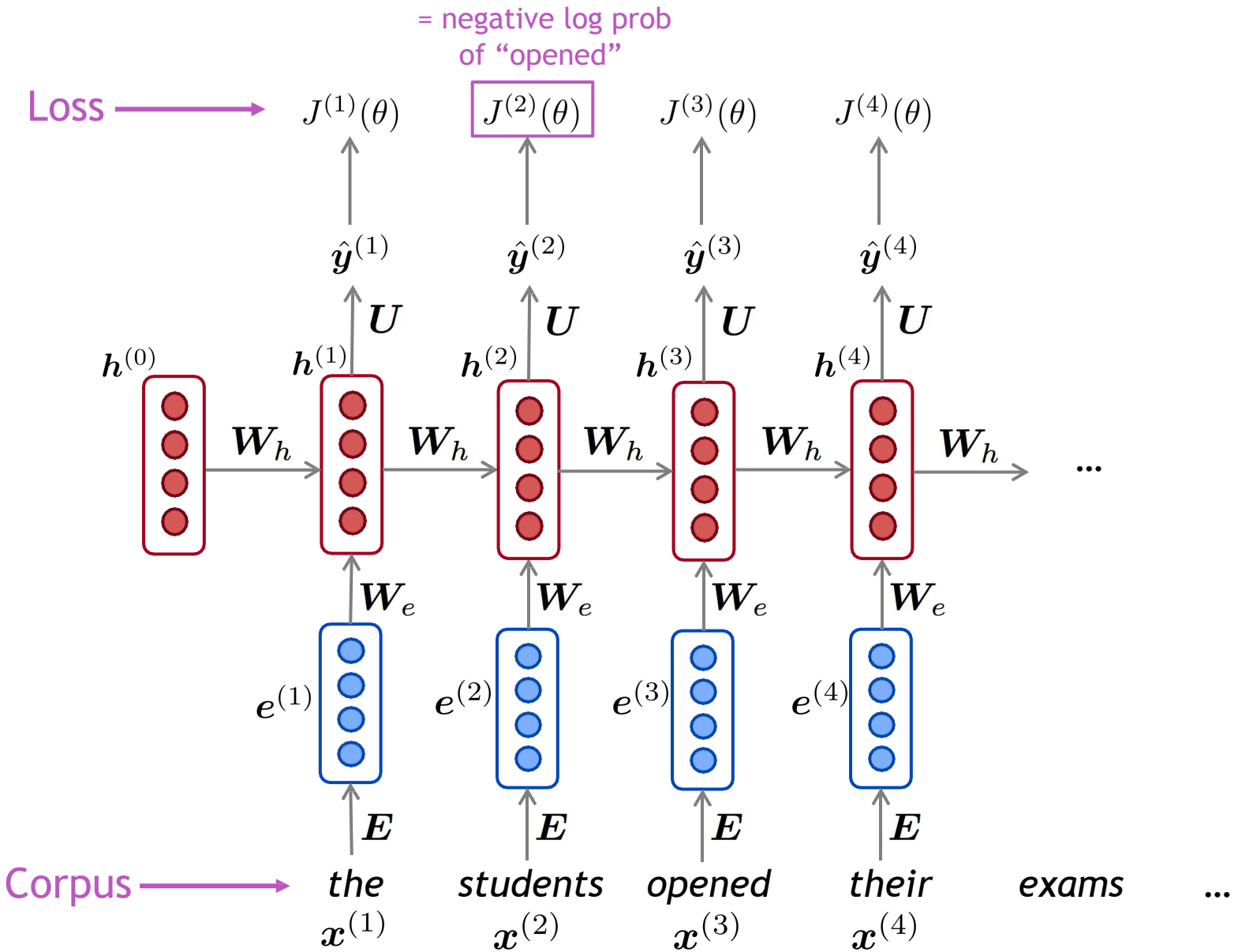
- Average this to get overall loss for entire training set:

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

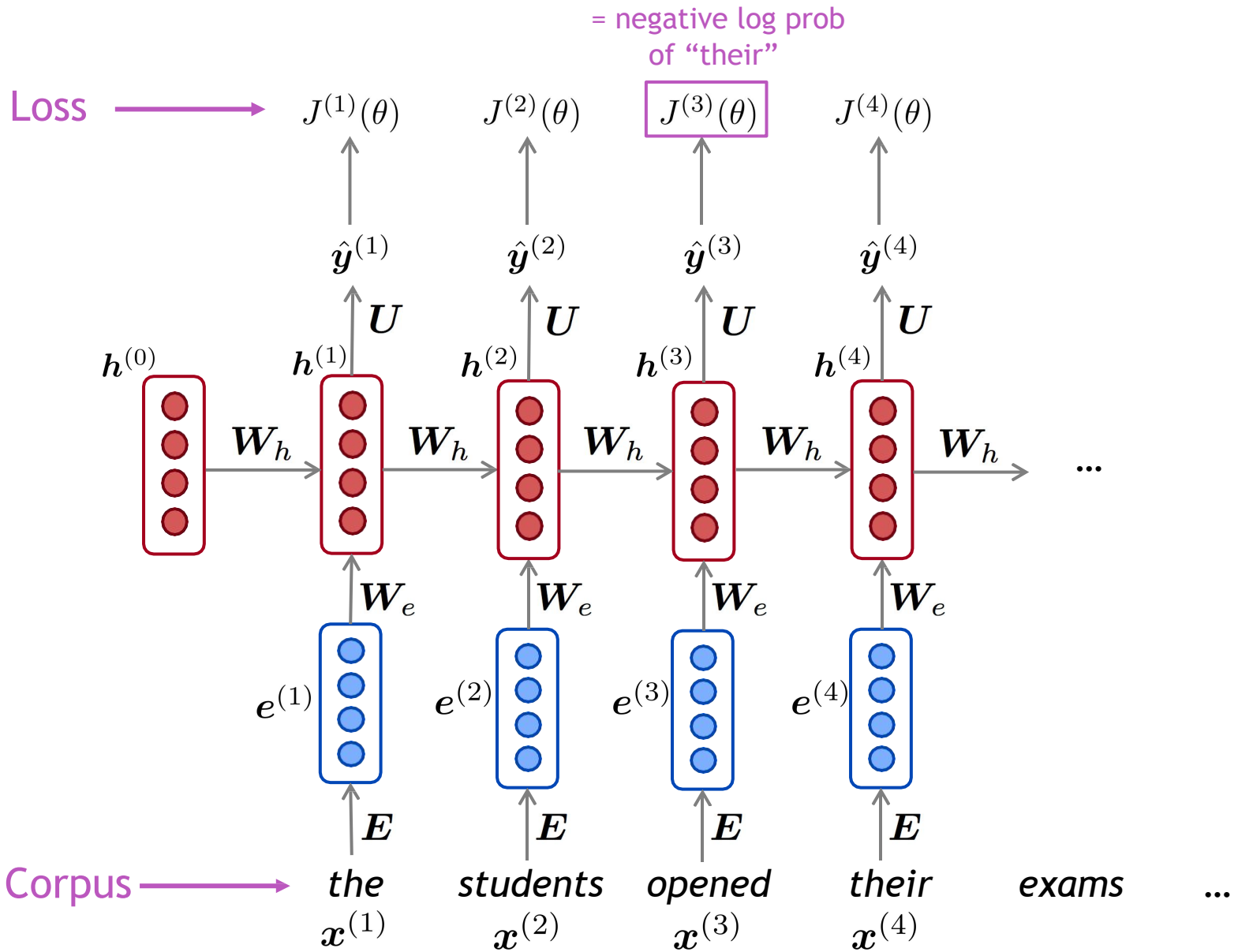
Training a RNN Language Model



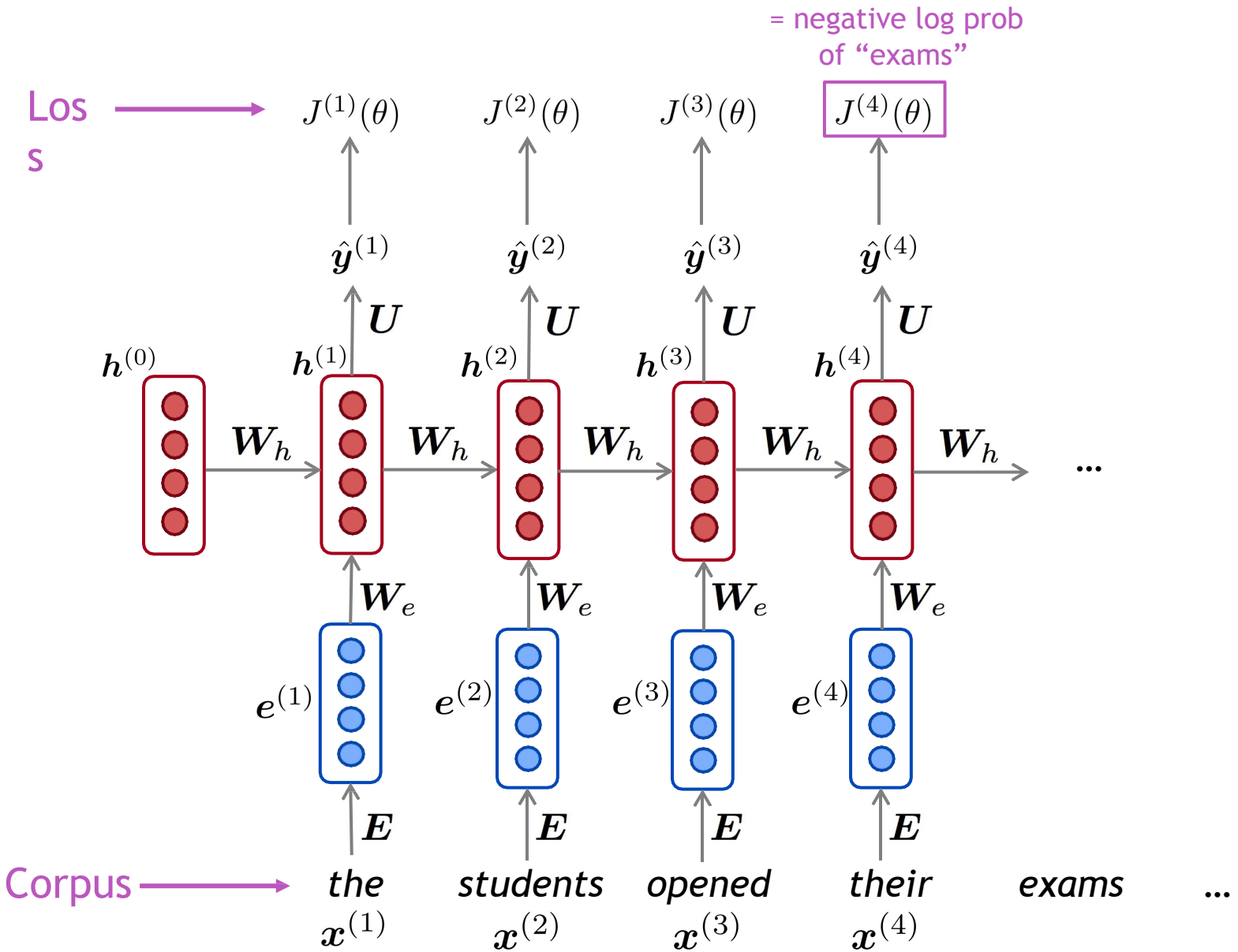
Training a RNN Language Model



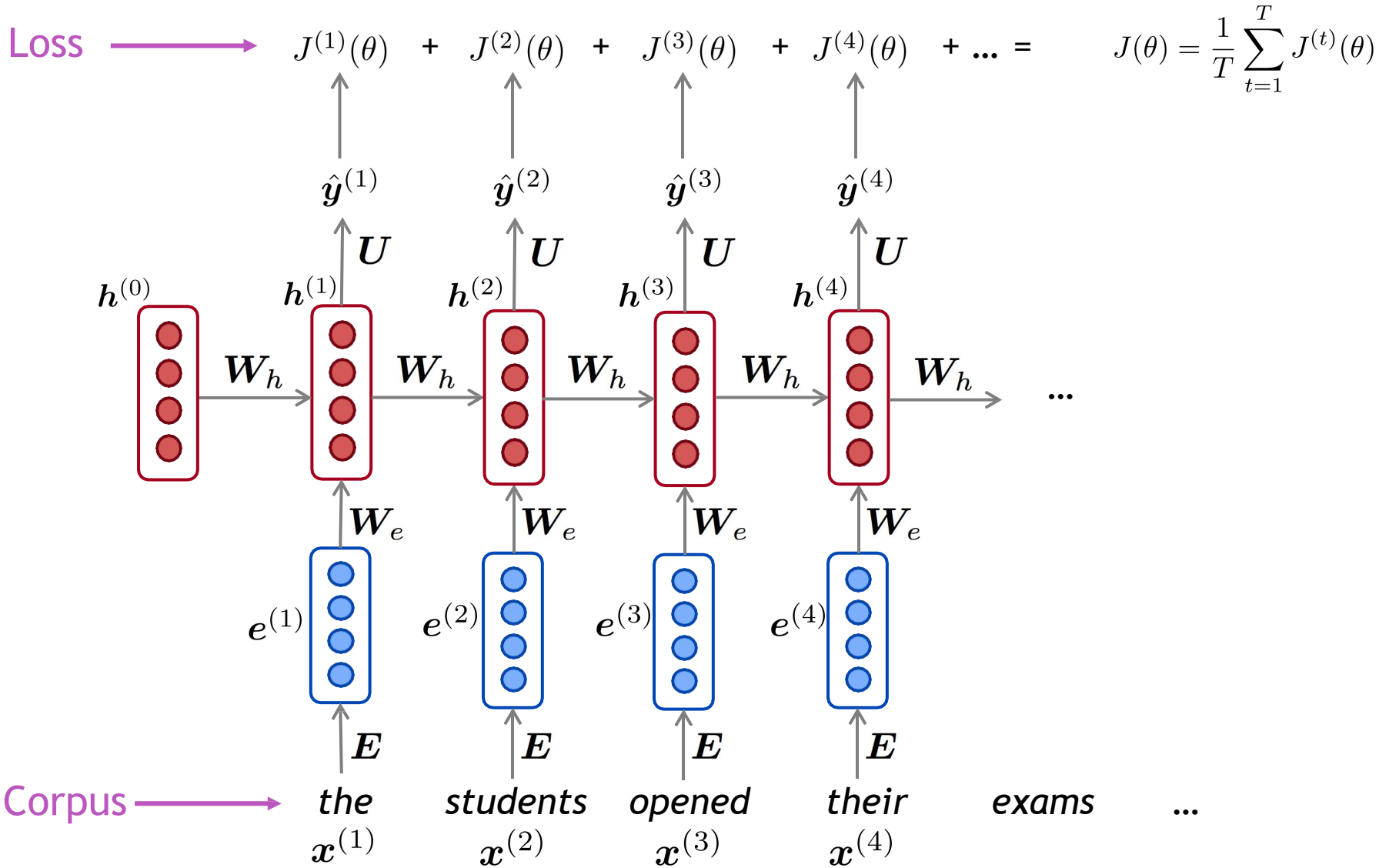
Training a RNN Language Model



Training a RNN Language Model



Training a RNN Language Model



Training a RNN Language Model

- However: Computing loss and gradients across entire corpus is too expensive!
- Recall: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.
- → In practice, consider $x^{(1)}, \ldots, x^{(T)}$ as a sentence

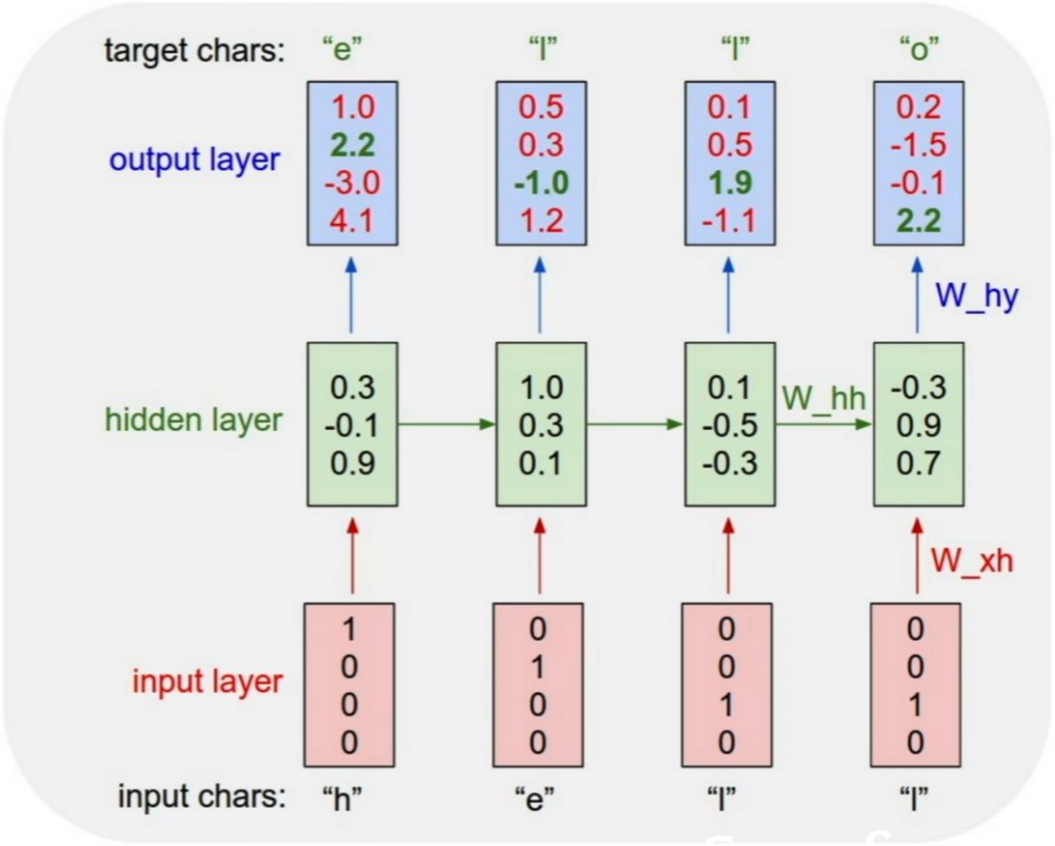
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$$

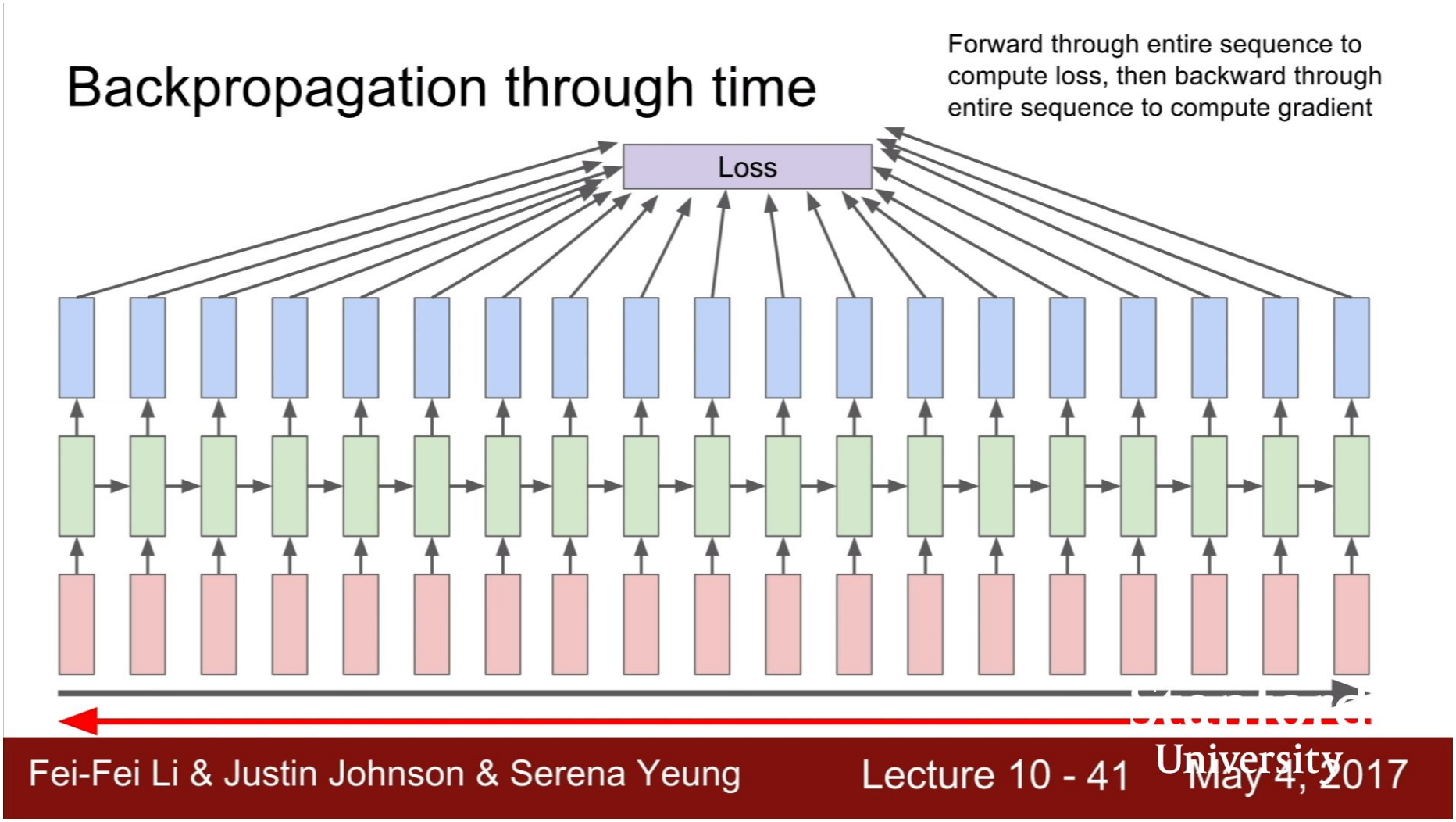
- Compute loss $J(\theta)$ for a sentence (actually usually a batch of sentences), compute gradients and update weights. Repeat.

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

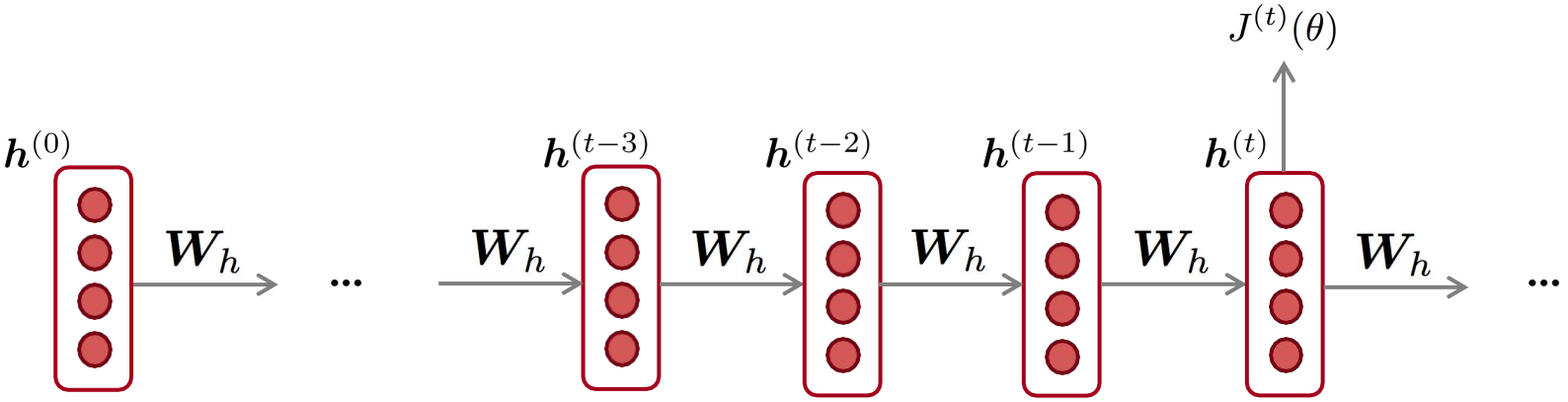
Example training sequence:
“hello”





Imagine training the entire wikipedia corpus to just get 1 gradient update...

Backpropagation for RNNs



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

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

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

Why?

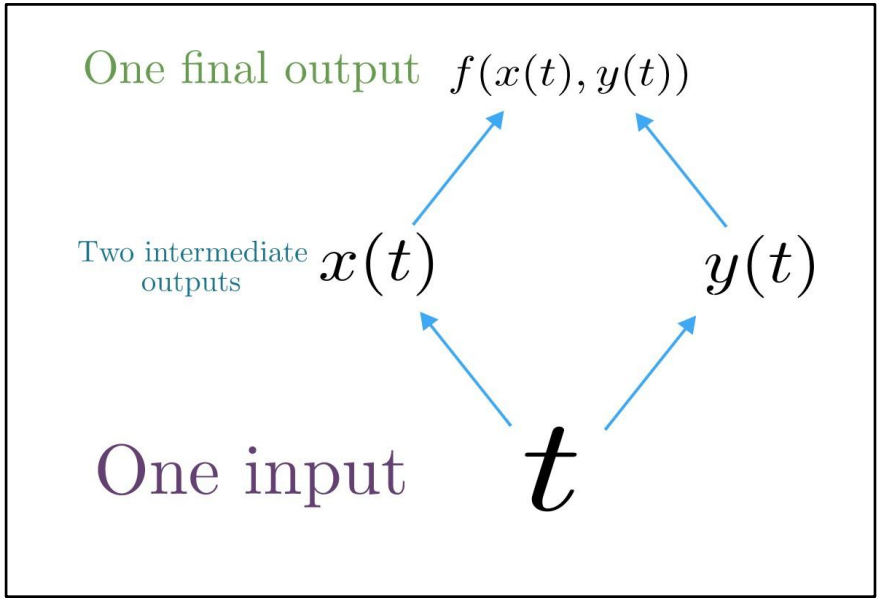
Backpropagation Review

http://cs231n.github.io/optimization-2/#staged

Multivariable Chain Rule

- 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:

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



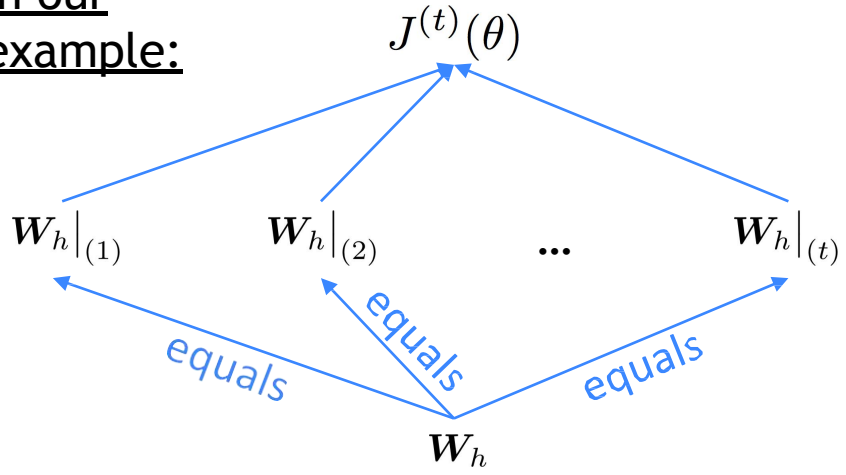
Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs: Proof sketch

- 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:



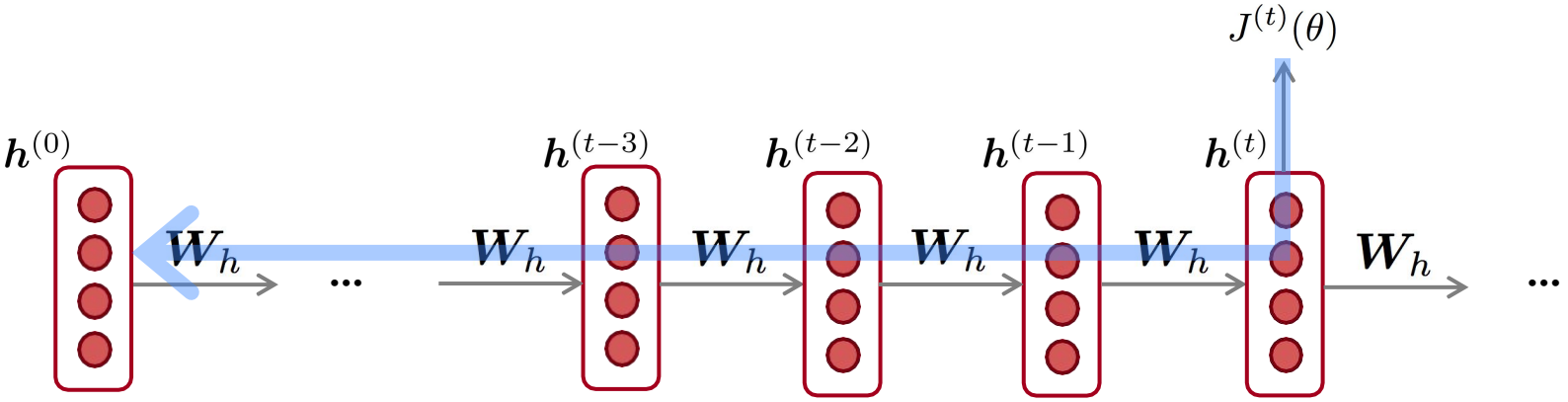
Apply the multivariable chain rule:

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

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

Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs



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

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"**

BPTT(backpropagation through time)

$$s_t = tanh(Ux_t + Ws_{t-1})$$

$$\hat{y}_t = softmax(Vs_t)$$

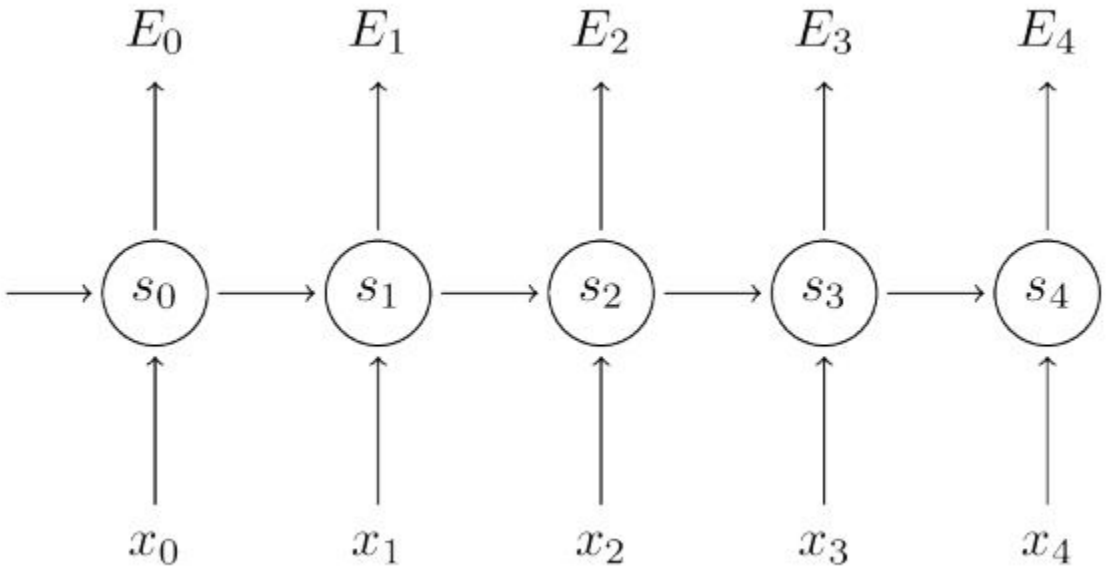
$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

$$= -\sum_t y_t \log \hat{y}_t$$

BPTT(backpropagation through time)

credit to DENNY BRITZ



$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}$$

$$= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}$$

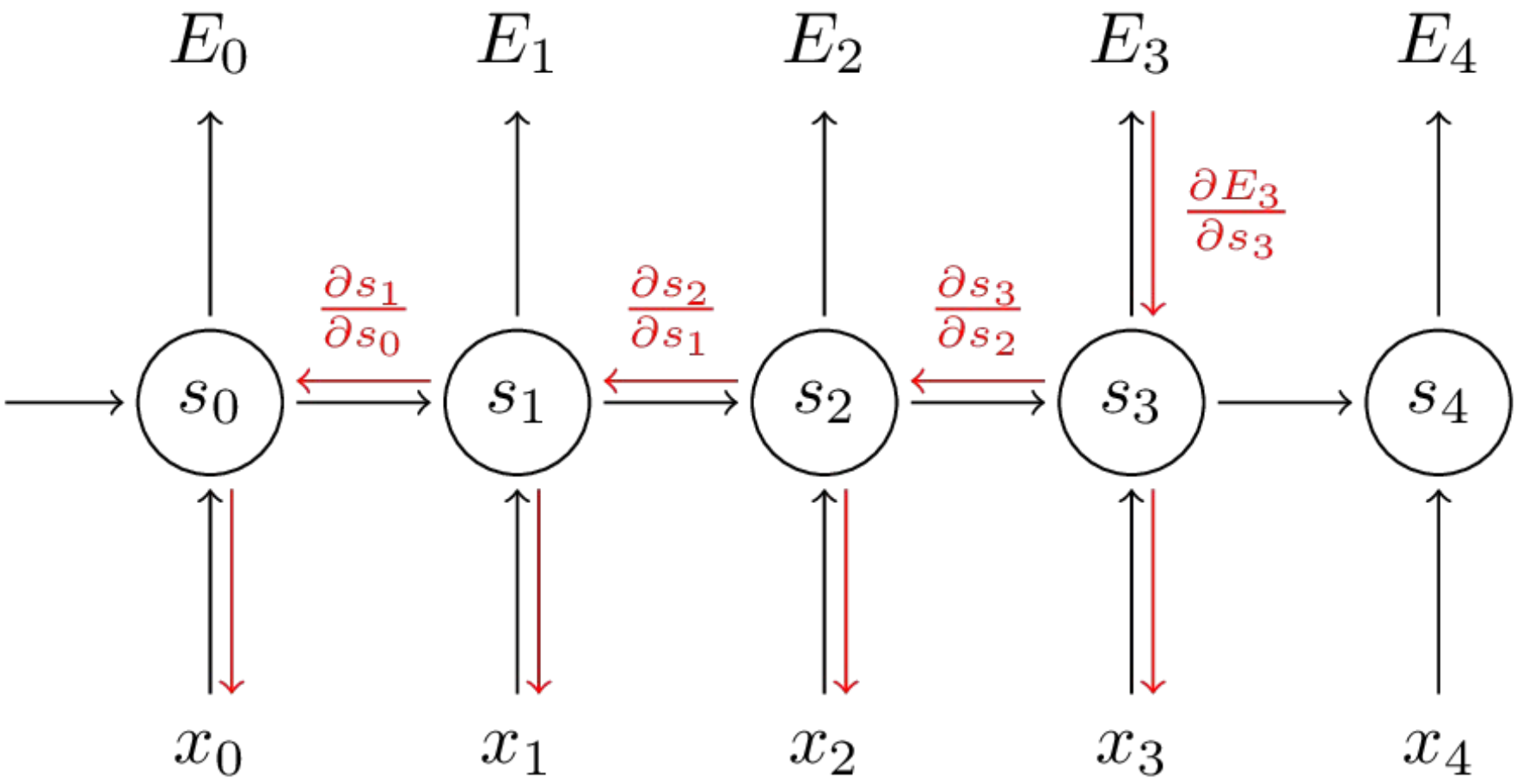
$$= (\hat{y}_3 - y_3) \otimes s_3$$

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3}\frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

BPTT(backpropagation through time)

credit to DENNY BRITZ



```python
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

Vanishing/exploding gradient problem

- Multiply the same W at each time step during backprop.
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation. This can become very small or very large quickly, and the locality assumption of gradient descent breaks down. → Vanishing or exploding gradient.

- Gradients can be seen as a measure of influence of the past on the future.
- The vanishing gradient problem can cause problems: When predicting the next word, information from many time steps in the past is not taken into consideration.

Vanishing/exploding gradient problem

- Gradient clipping method
- Initialization and ReLus
- Gated Recurrent Units (GRU) introduced by [Cho et al. 2014] and LSTMs [Hochreiter & Schmidhuber, 1999]

Truncated BPTT

Truncated BPTT processes the sequence one timestep at a time, and every k1 timesteps, it runs BPTT for k2 timesteps, so a parameter update can be cheap if k2 is small.

1. **k1**: The number of forward-pass timesteps between updates. Generally, this influences how slow or fast training will be, given how often weight updates are performed.
2. **k2**: The number of timesteps to which to apply BPTT. Generally, it should be large enough to capture the temporal structure in the problem for the network to learn. Too large a value results in vanishing gradients.

TBPTT (k1, k2)

- **TBPTT(n,n)**: Updates are performed at the end of the sequence across all timesteps in the sequence (e.g. classical BPTT).
- **TBPTT(1,n)**: timesteps are processed one at a time followed by an update that covers all timesteps seen so far (e.g. classical TBPTT by Williams and Peng).
- **TBPTT(k1,1)**: The network likely does not have enough temporal context to learn, relying heavily on internal state and inputs.
- **TBPTT(k1,k2), where k1<k2<n**: Multiple updates are performed per sequence which can accelerate training.
- **TBPTT(k1,k2), where k1=k2**: A common configuration where a fixed number of timesteps are used for both forward and backward-pass timesteps (e.g. 10s to 100s).

Prepare sequence data

The way that you break up your sequence data will define the number of timesteps used in the forward and backward passes of BPTT.

- Use data as-is
- Naive data split
- Domain-specific data split
  - In natural language processing problem, the input sequence could be divided by sentence and then padded to a fixed length, or split according to the average sentence length in the domain.
- Systematic data split (e.g. grid search)
  - perform a grid search over each sub-sequence length and adopt the configuration that results in the best performing model on average.
- Lean heavily on internal states with TBPTT(1, 1)

Pseudo code truncated version of BPTT

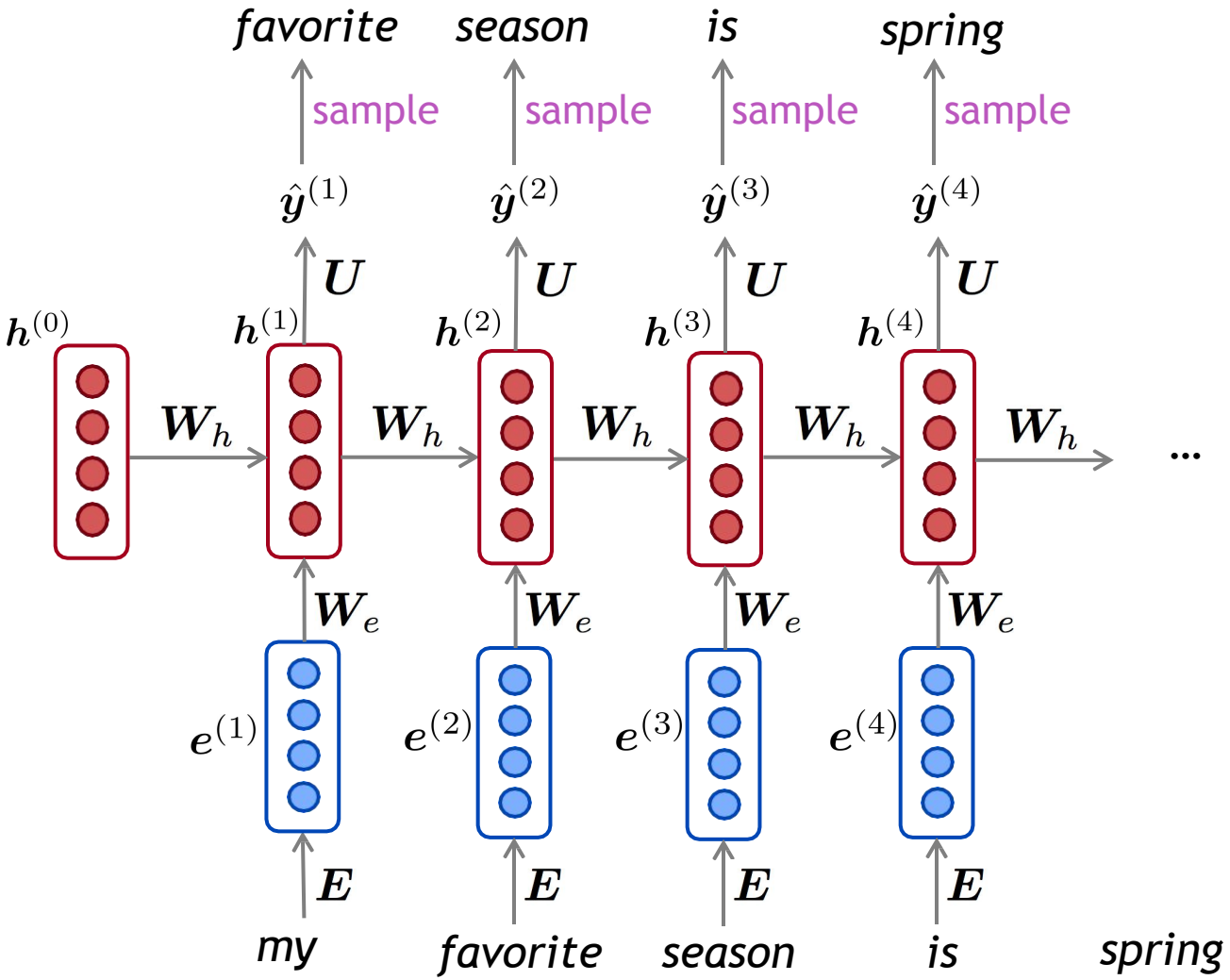
```
Back_Propagation_Through_Time(a, y)   // a[t] is the input at time t. y[t] is the output
    Unfold the network to contain k instances of f
    do until stopping criteria is met:
        x = the zero-magnitude vector;// x is the current context
        for t from 0 to n - k         // t is time. n is the length of the training sequence
            Set the network inputs to x, a[t], a[t+1], ..., a[t+k-1]
            p = forward-propagate the inputs over the whole unfolded network
            e = y[t+k] - p;           // error = target - prediction
            Back-propagate the error, e, back across the whole unfolded network
            Sum the weight changes in the k instances of f together.
            Update all the weights in f and g.
            x = f(x, a[t]);           // compute the context for the next time-step
```

RNN Applications

- Language Model
- Sentiment Classification
- Machine Translation
- Question Answering
- Speech recognition
- Time series prediction

Generating text with a RNN Language Model

Just like a n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output is next step's input.



Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on Obama speeches:

The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.

Source:
https://medium.com/@samim/obama-rnn-machine-generated-political-speeches-c8abd18a2ea0

Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:

"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

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

Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Seinfeld* scripts:

He slams his hand on the door. KRAMER enters dancing with garbage.

KRAMER
Hey hey hey, great idea for a big sponge: Make it so large you think it's got a fat clock in the middle.

JERRY
(takes off his bones)
Kramer, do you have a fun flashback to do?

Source:
https://www.avclub.com/a-bunch-of-comedy-writers-teamed-up-with-a-computer-to-1818633242

Generating text with a RNN Language Model

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- (character-level) RNN-LM trained on paint colors:

Ghasty Pink 231 137 165	Sand Dan 201 172 143
Power Gray 151 124 112	Grade Bat 48 94 83
Navel Tan 199 173 140	Light Of Blast 175 150 147
Bock Coe White 221 215 236	Grass Bat 176 99 108
Horble Gray 178 181 196	Sindis Poop 204 205 194
Homestar Brown 133 104 85	Dope 219 209 179
Snader Brown 144 106 74	Testing 156 101 106
Golder Craam 237 217 177	Stoner Blue 152 165 159
Hurky White 232 223 215	Burble Simp 226 181 132
Burf Pink 223 173 179	Stanky Bean 197 162 171
Rose Hork 230 215 198	Turdly 190 164 116

Source:
http://aiweirdness.com/post/160776374467/new-paint-colors-invented-by-neural-network

Evaluating Language Models

- The traditional evaluation metric for Language Models is perplexity.

$$\mathrm{PP} = \underbrace{\prod_{t=1}^{T} \left(\frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)}_{\text{Inverse probability of dataset}}^{1/T}$$

Normalized by number of words

- Lower is better!
- minimizing perplexity and minimizing the loss function are equivalent.
- $log(PP) = J(\theta)$

RNNs have greatly improved perplexity

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)

Source: https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

Why should we care about Language Modeling?

- Language Modeling is a subcomponent of other NLP systems:
  - Speech recognition
    - Use a LM to generate transcription, conditioned on audio
  - Machine Translation
    - Use a LM to generate translation, conditioned on original text
  - Summarization
    - Use a LM to generate summary, conditioned on original text

These systems are called *conditional Language Models*

- Language Modeling is a benchmark task that helps us measure our progress on understanding language

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

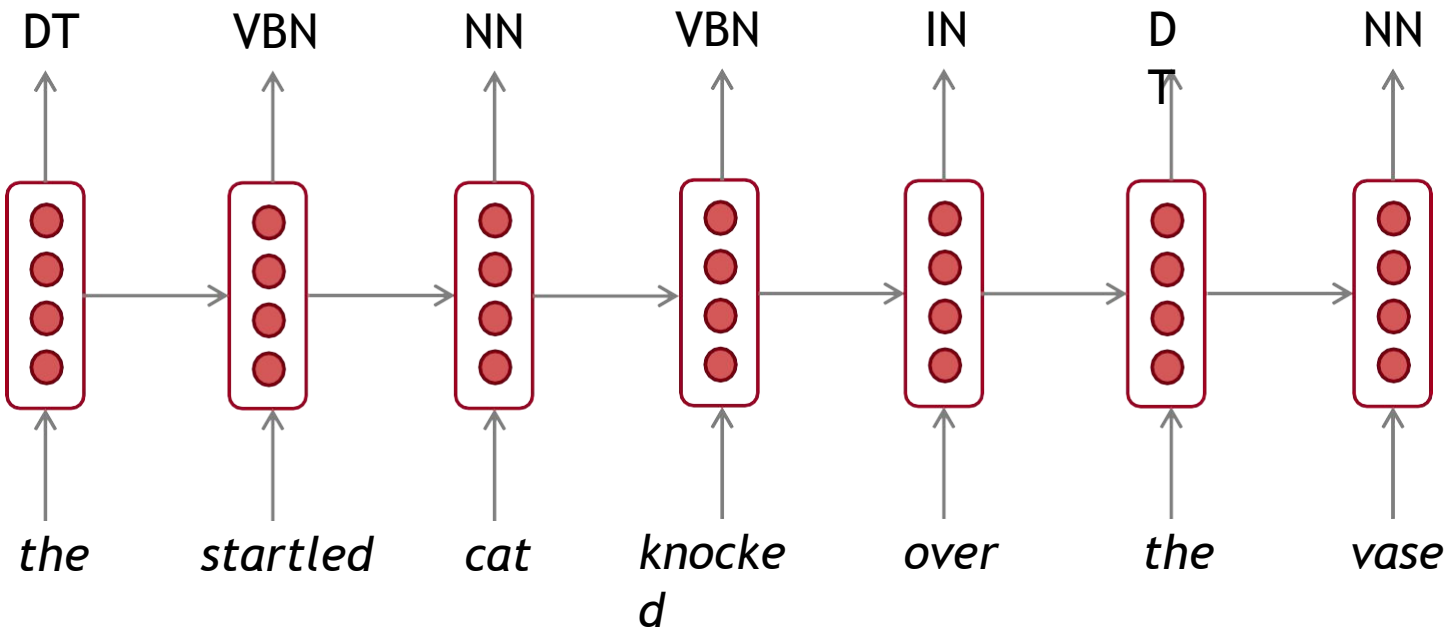
- Recurrent Neural Network ≠ Language Model

- We've shown that RNNs are a great way to build a LM.

- But RNNs are useful for much more!

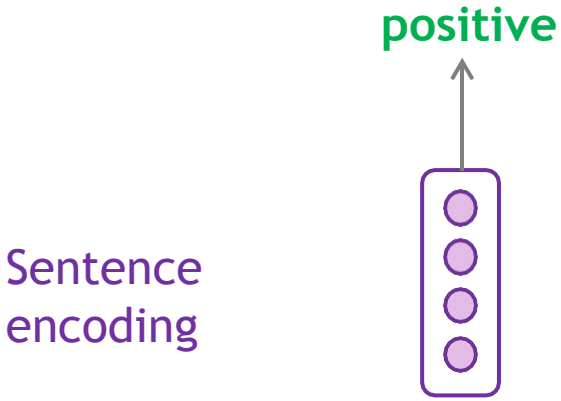
RNNs can be used for tagging

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

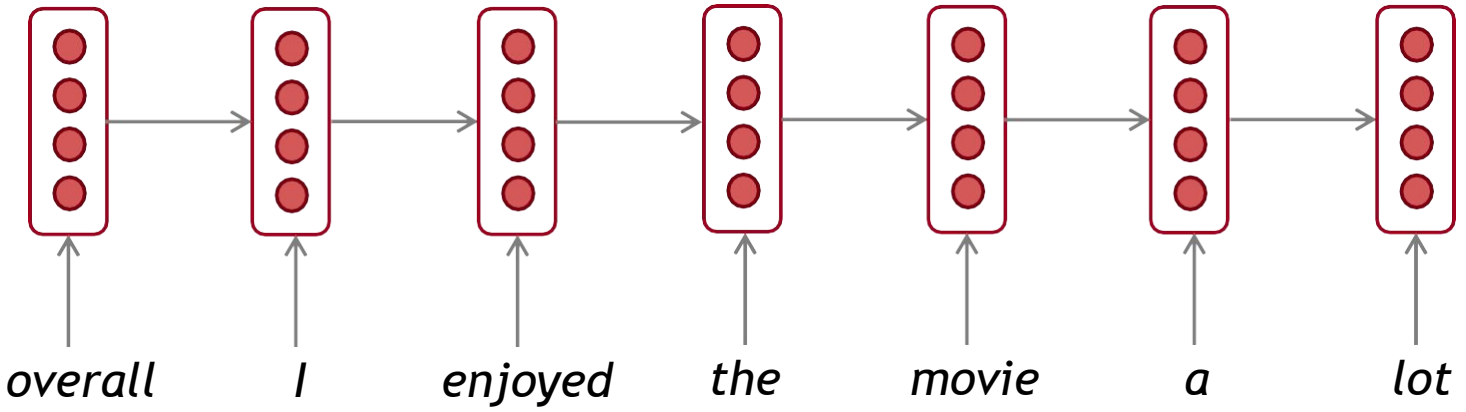


RNNs can be used for sentence classification

e.g. sentiment classification

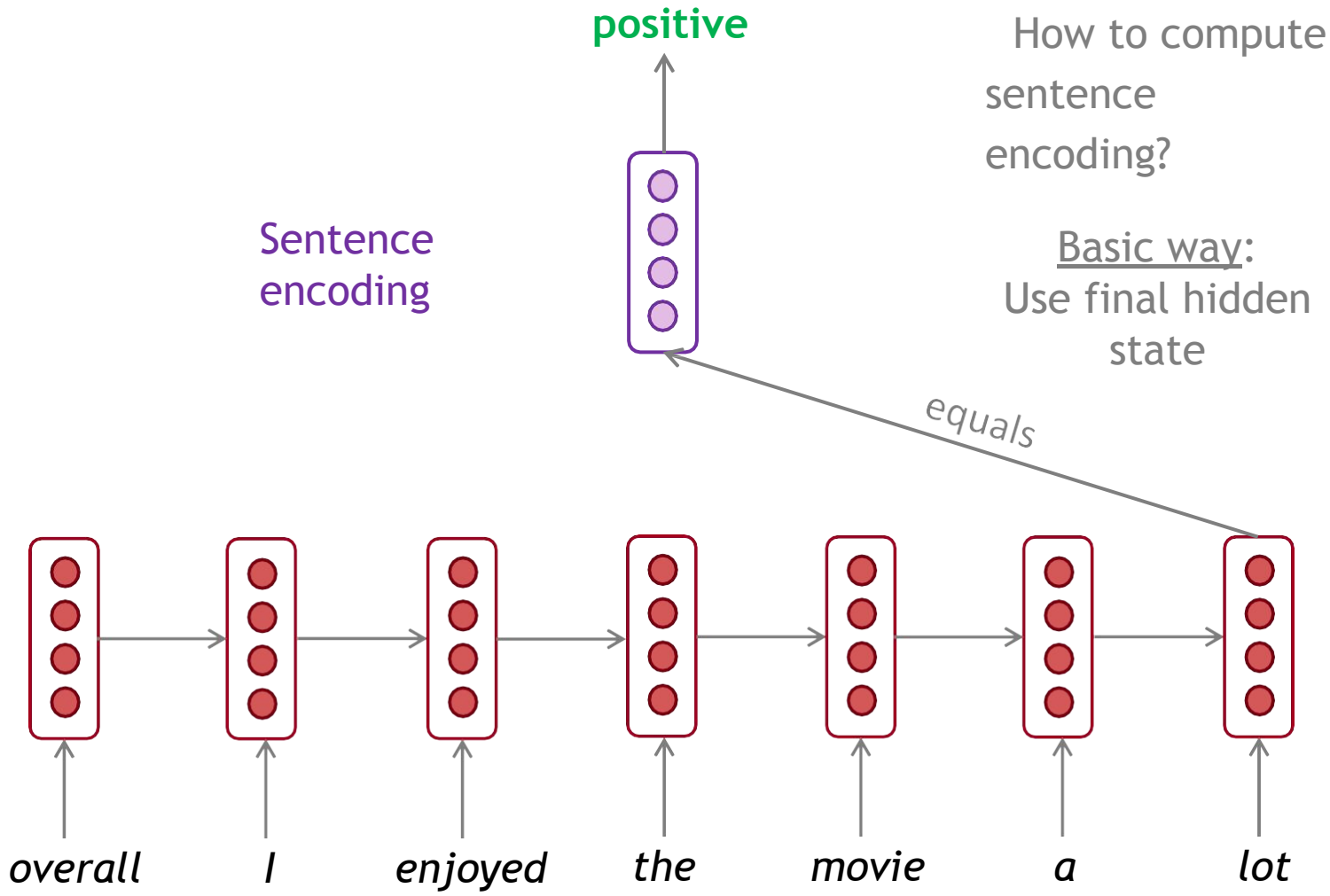


How to compute sentence encoding?



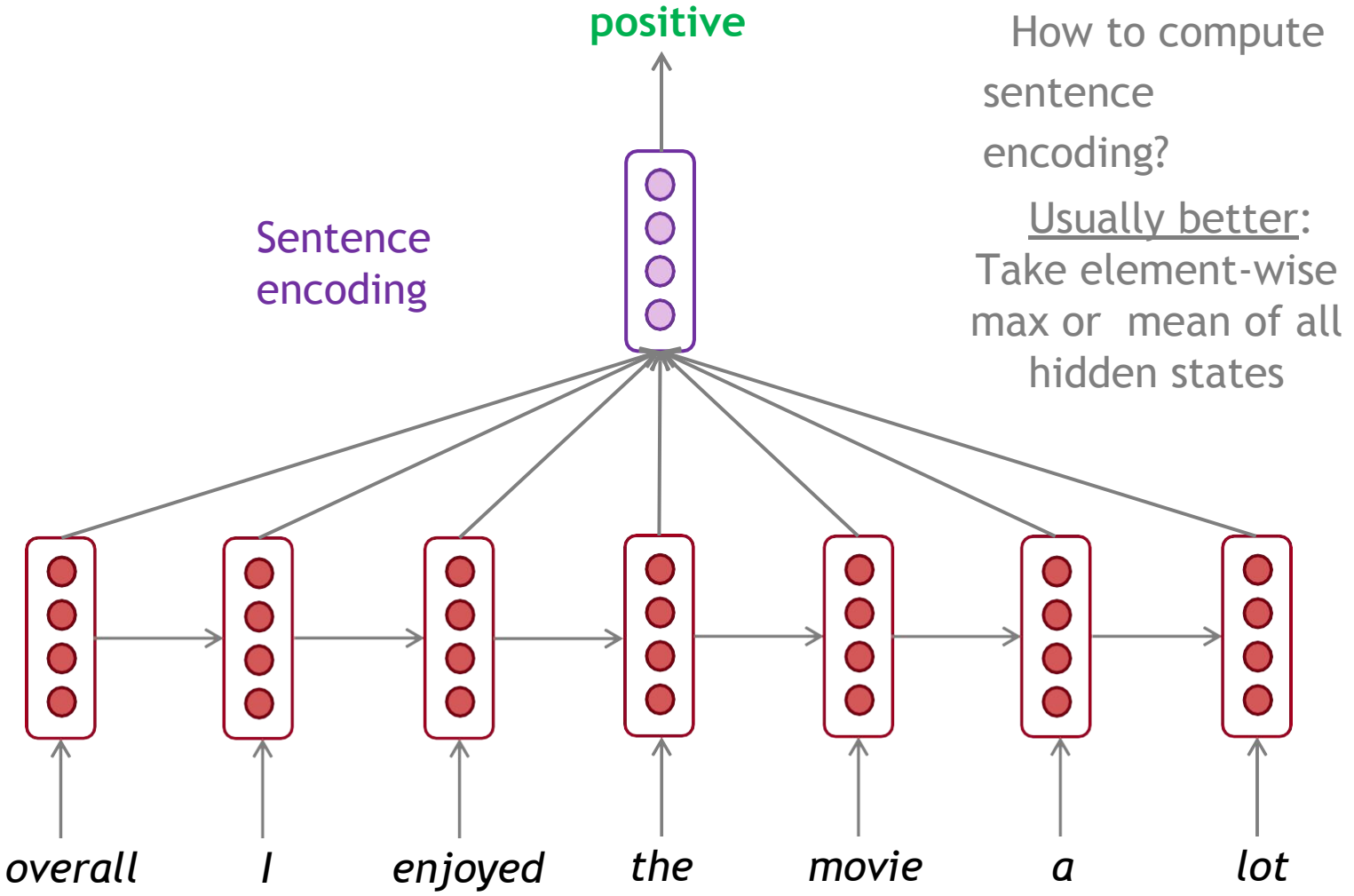
RNNs can be used for sentence classification

e.g. sentiment classification



RNNs can be used for sentence classification

e.g. sentiment classification

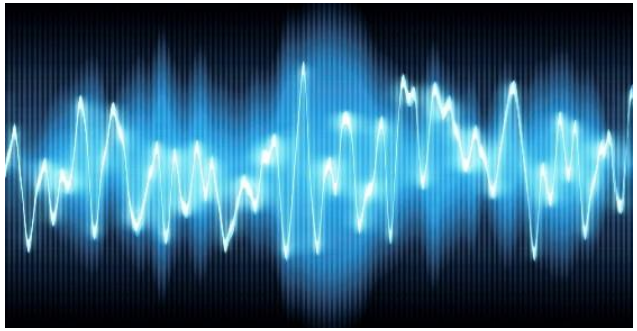


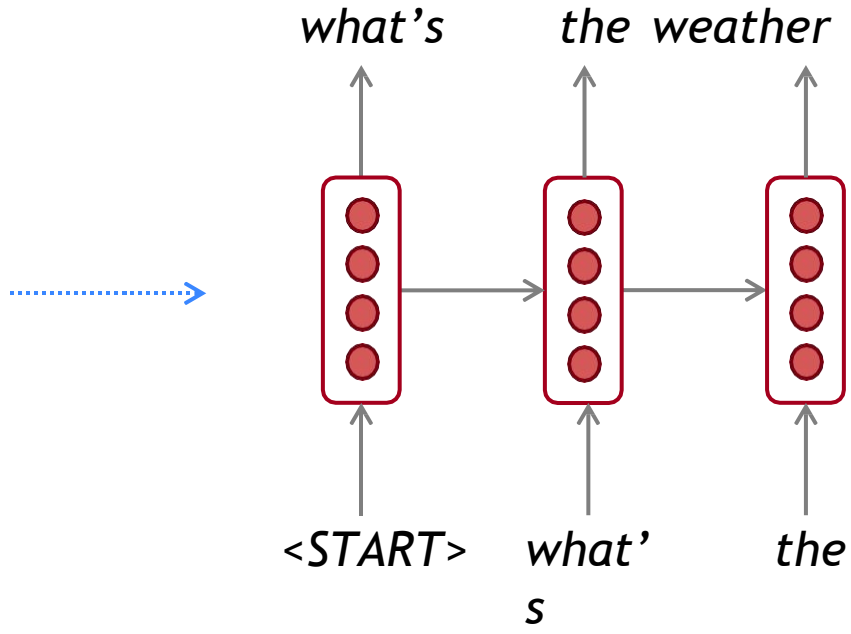
How to compute sentence encoding?

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

RNNs can be used to generate text

e.g. speech recognition, machine translation, summarization

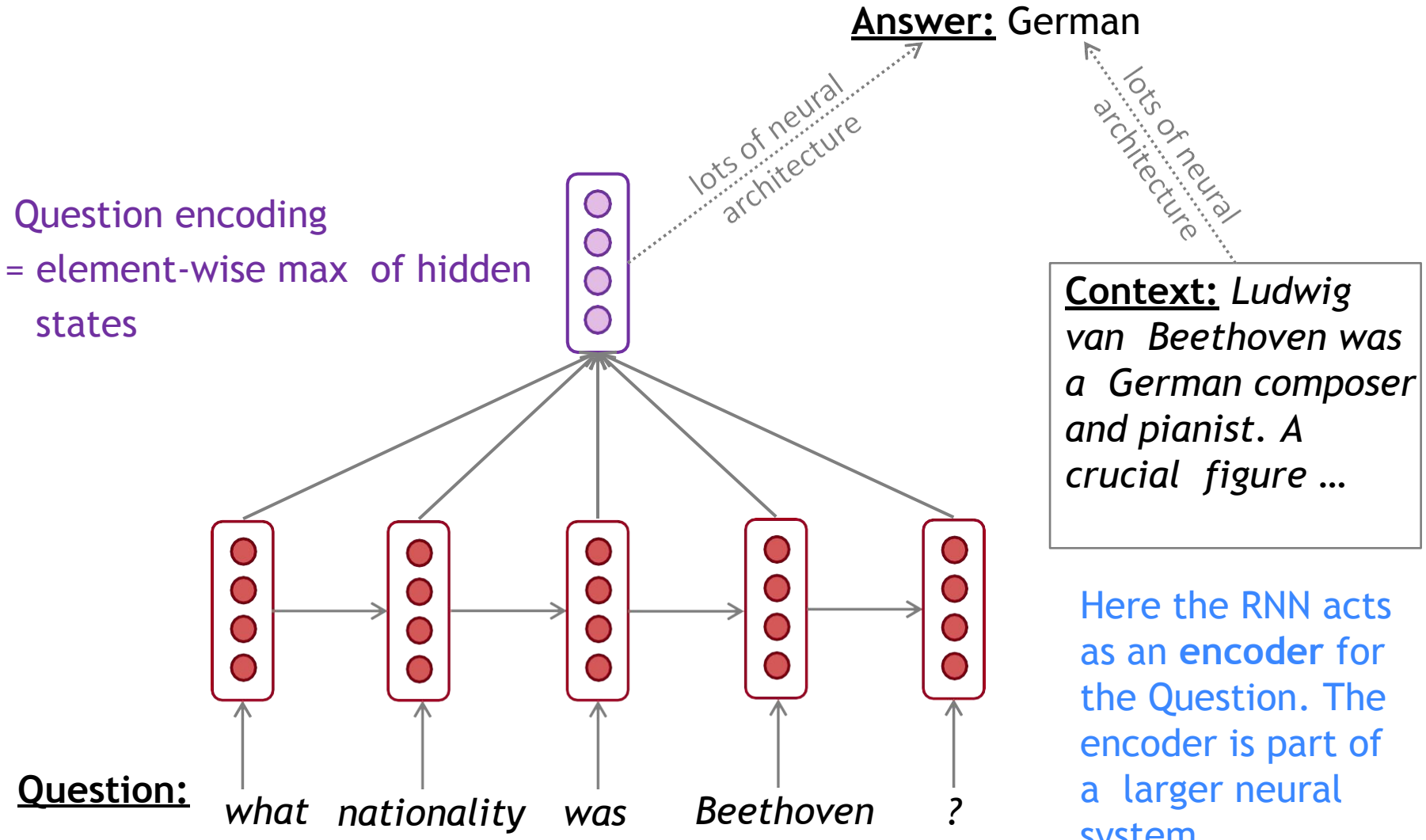




Remember: these are called "conditional language models". We'll see Machine Translation in much more detail later.

RNNs can be used as an encoder module

e.g. question answering, machine translation



Here the RNN acts as an **encoder** for the Question. The encoder is part of a larger neural system.

Thank you