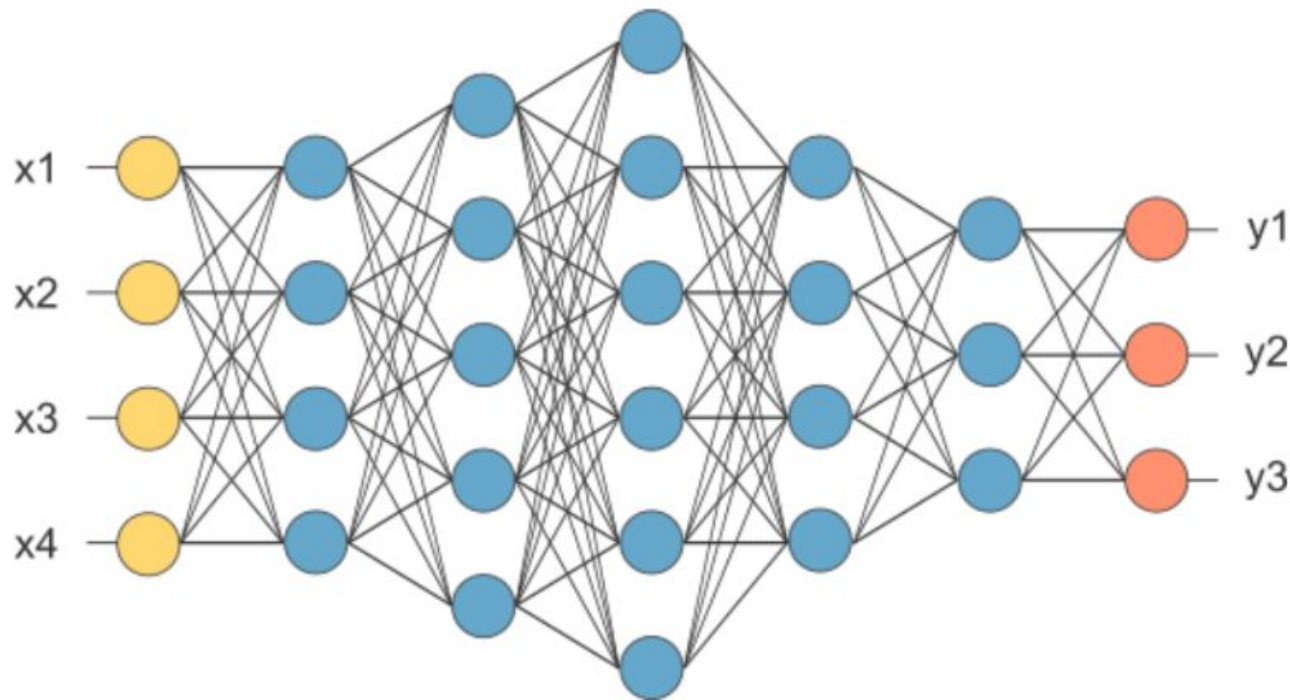


THIS IS CS4045!

GCR:dxuxugo

Fully Connected Network



This is our fully connected network. If $x_1 \dots x_n$, n is very large and growing, this network would become too large. We now will input **one x_i at a time**, and **re-use the same edge weights**.

Examples of sequence data

Speech recognition



"The quick ^ybrown fox jumped
over the lazy dog."

Music generation



Sentiment classification

"There is nothing to like
in this movie."



DNA sequence analysis

AGCCCCTGTGAGGAACTAG



AGCCCCTGTGAGGAACTAG

Machine translation

Voulez-vous chanter avec
moi?



Do you want to sing with
me?

Video activity recognition



Running

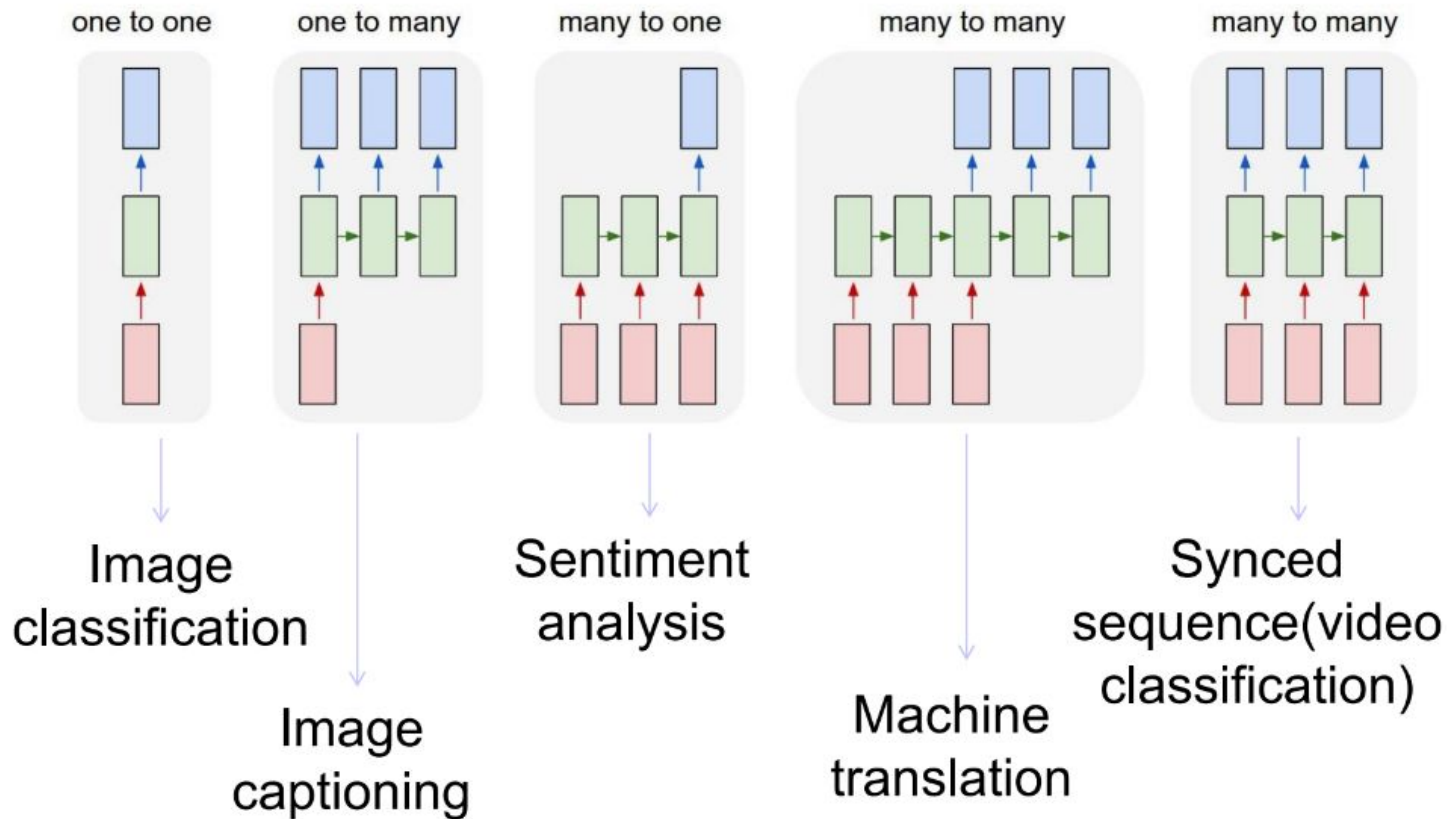
Name entity recognition

Yesterday, Harry Potter
met Hermione Granger.



Yesterday, **Harry Potter**
met **Hermione Granger**.
Andrew Ng

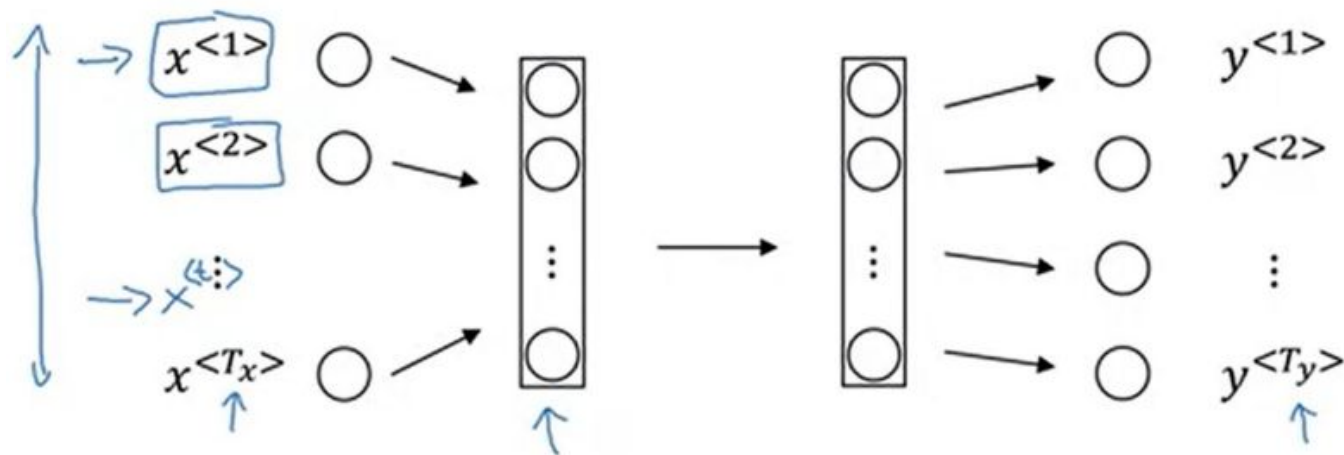
Motivation



Motivation

- Feed forward networks accept a fixed-sized vector as input and produce a fixed-sized vector as output
- fixed amount of computational steps
- recurrent nets allow us to operate over *sequences* of vectors

Why not a standard network?

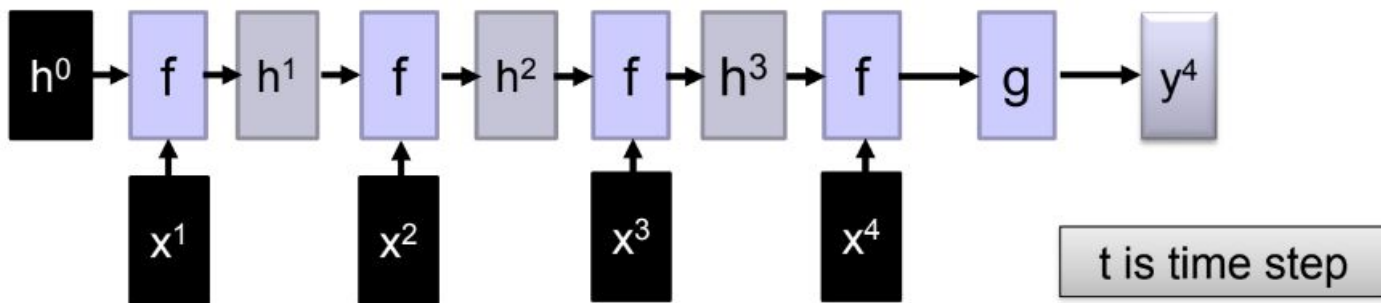
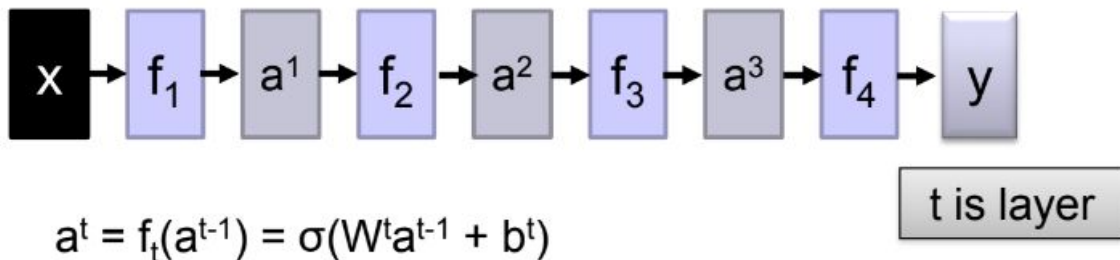


Problems:

- - Inputs, outputs can be different lengths in different examples.
- - Doesn't share features learned across different positions of text.

Feed-forward vs Recurrent Network

1. Feedforward network does not have input at each step
2. Feedforward network has different parameters for each layer



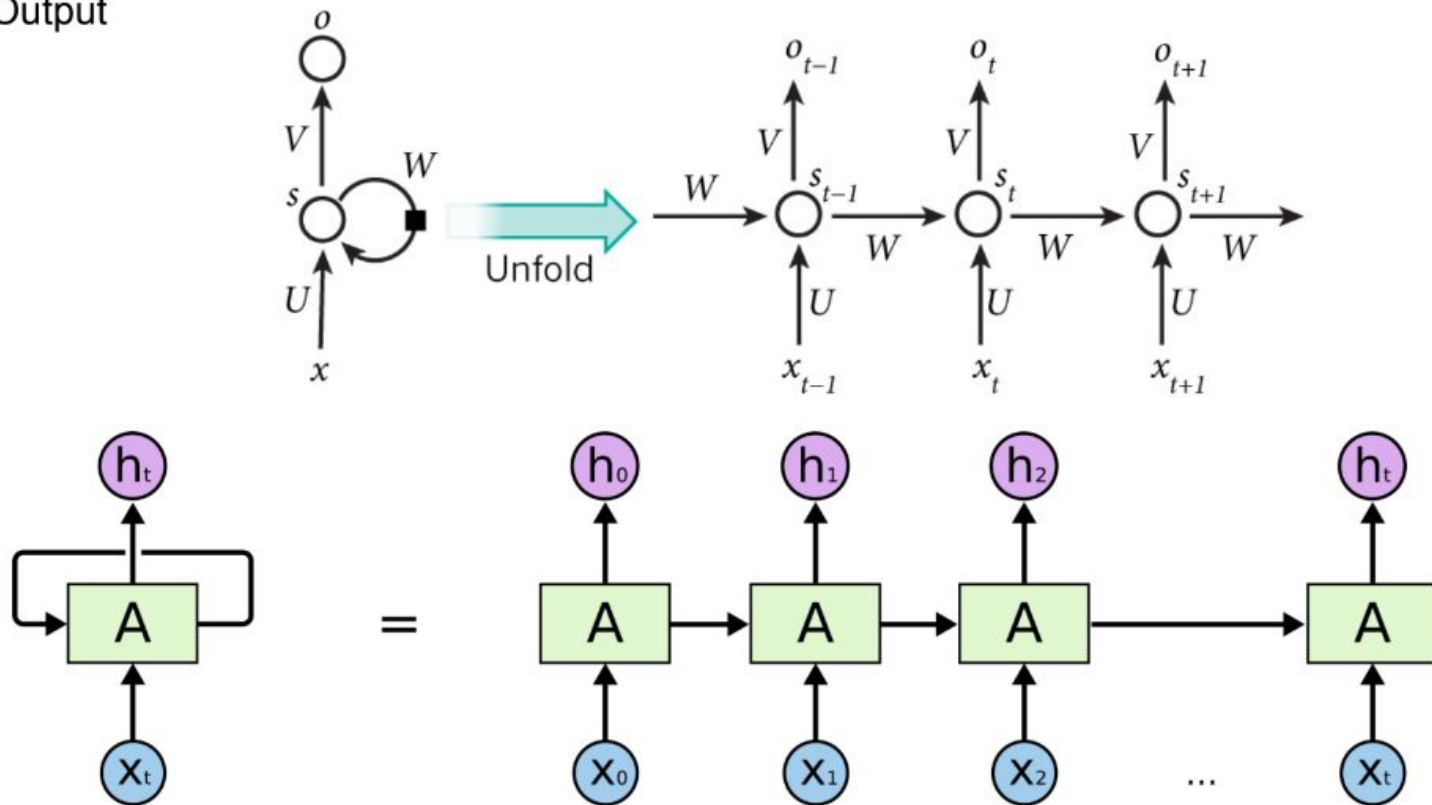
$$h^t = f(h^{t-1}, x^t) = \sigma(W^h h^{t-1} + W^i x^t + b^i)$$

Recurrent Neural Network

Input \rightarrow Hidden \rightarrow Output

it becomes

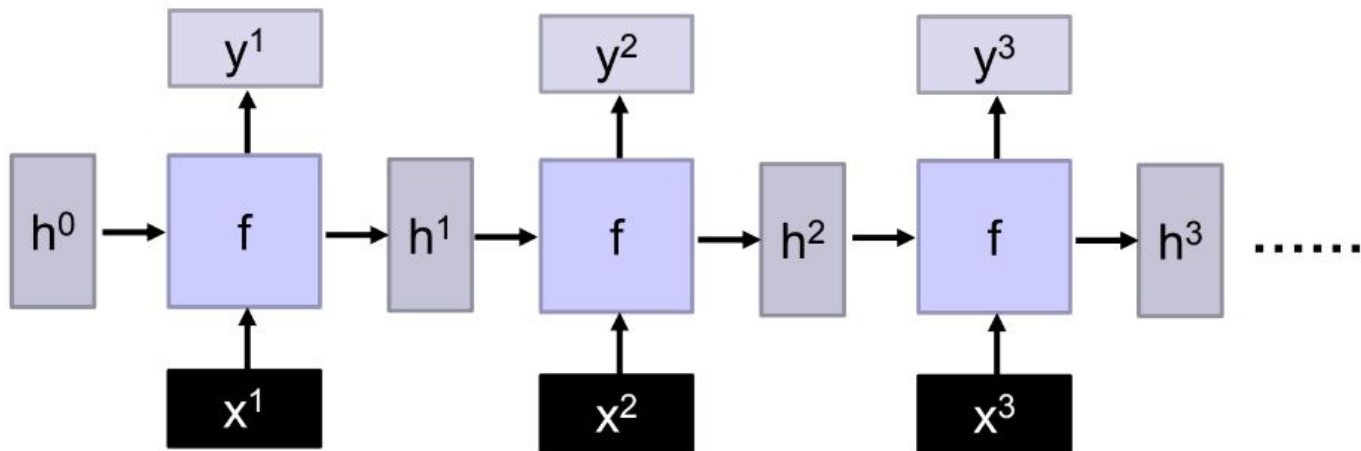
Input + Previous Hidden \rightarrow Hidden \rightarrow
Output



How does RNN reduce complexity?

- Given function $f: h', y = f(h, x)$

h and h' are vectors with the same dimension



No matter how long the input/output sequence is, we only need one function f . If f 's are different, then it becomes a feedforward NN. This may be treated as another compression from fully connected network.

APPLICATION

Generating Text

Machine Translation

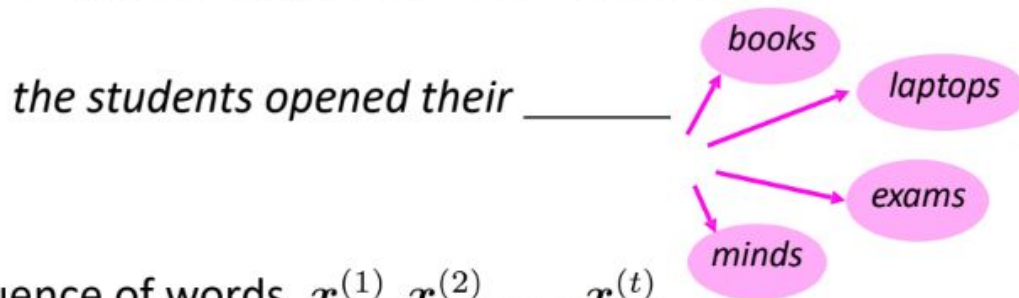
Speech Recognition

Generating Image Descriptions

Chatbots

Language Modeling

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



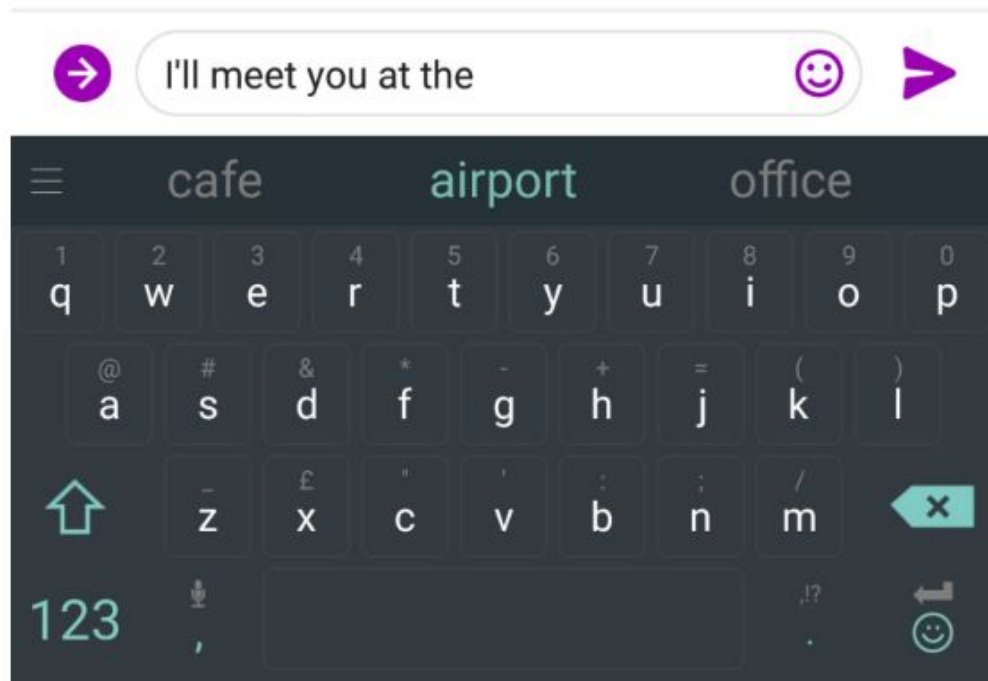
- More formally: given a sequence of words $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$, compute the probability distribution of the next word $\mathbf{x}^{(t+1)}$

$$P(\mathbf{x}^{(t+1)} \mid \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$$

where $\mathbf{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\mathbf{w}_1, \dots, \mathbf{w}_{|V|}\}$


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

You use Language Models every day!



You use Language Models every day!



what is the | 

what is the **weather**
what is the **meaning of life**
what is the **dark web**
what is the **xfl**
what is the **doomsday clock**
what is the **weather today**
what is the **keto diet**
what is the **american dream**
what is the **speed of light**
what is the **bill of rights**

A fixed-window neural Language Model

output distribution

$$\hat{y} = \text{softmax}(Uh + b_2) \in \mathbb{R}^{|V|}$$

hidden layer

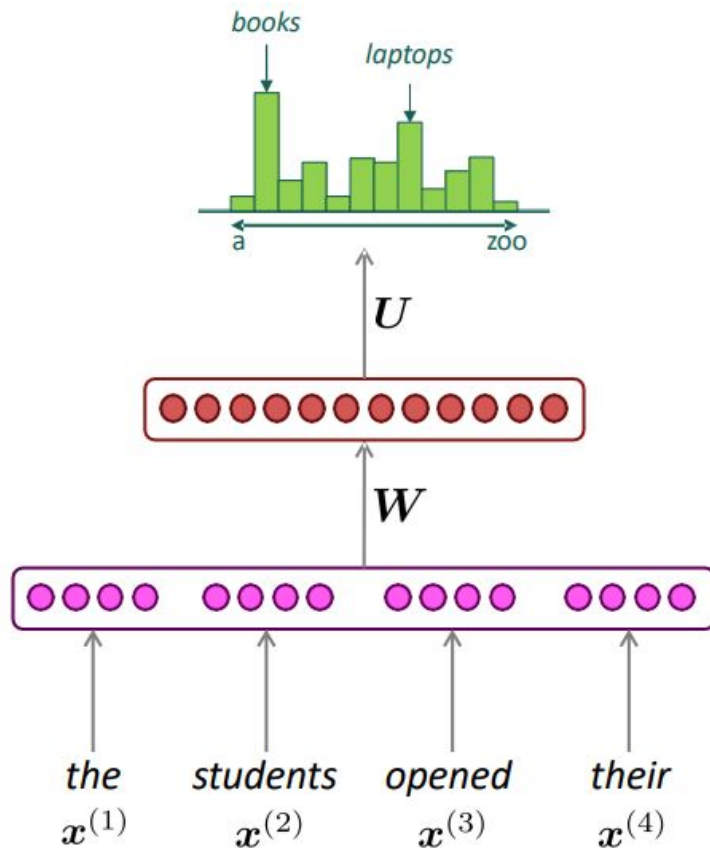
$$h = f(We + b_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



A fixed-window neural Language Model

Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over n -gram LM:

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

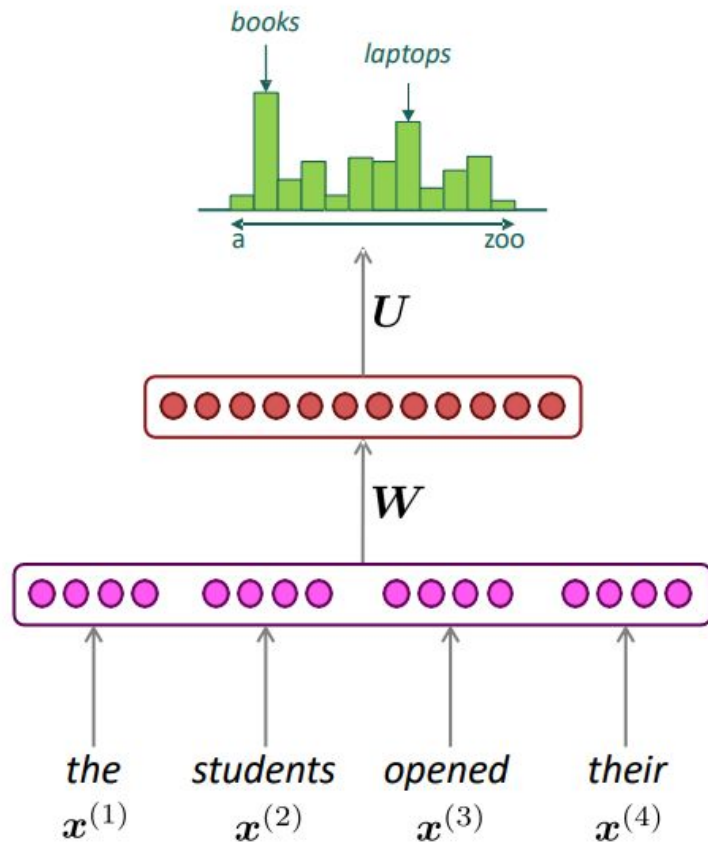
Remaining **problems**:

- Fixed window is **too small**
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(i)}$ are multiplied by

completely different weights in W .

No symmetry in how the inputs are processed

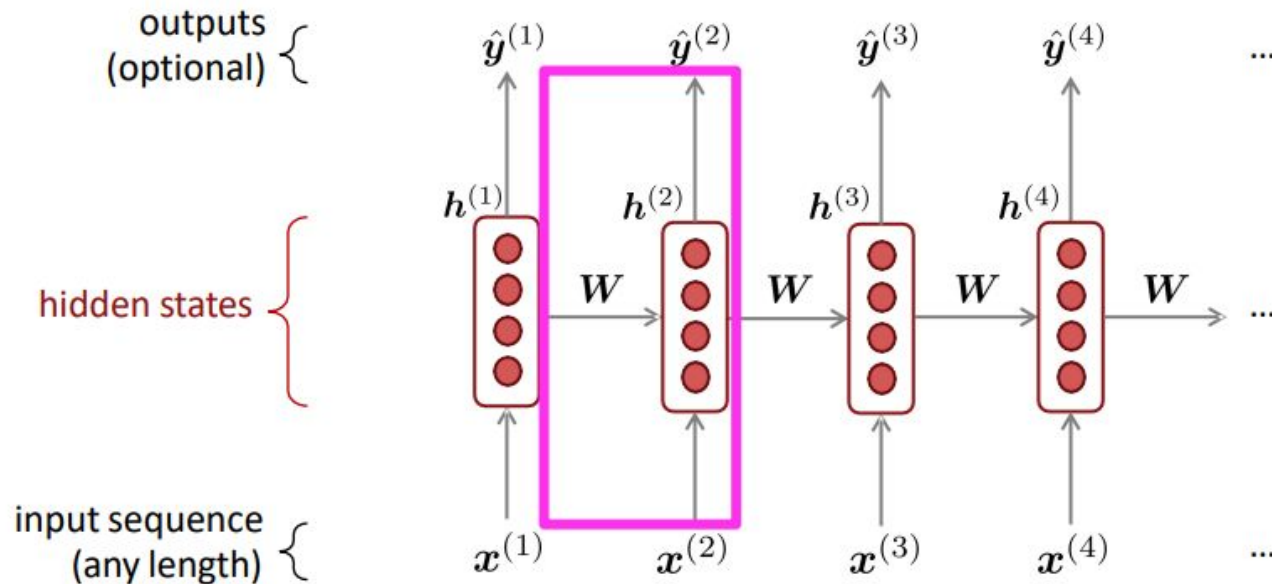
we need a neural architecture
that can process *any length input*



Recurrent Neural Networks (RNN)

A family of neural architectures

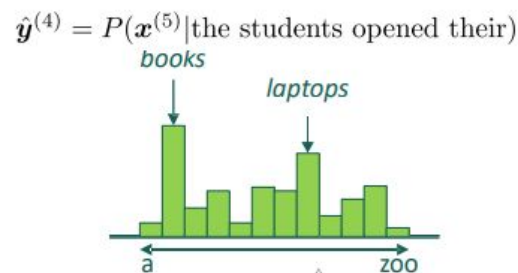
Core idea: Apply
the same weights
 W repeatedly



A Simple RNN Language Model

output distribution

$$\hat{y}^{(t)} = \text{softmax}(U h^{(t)} + b_2) \in \mathbb{R}^{|V|}$$



hidden states

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$$

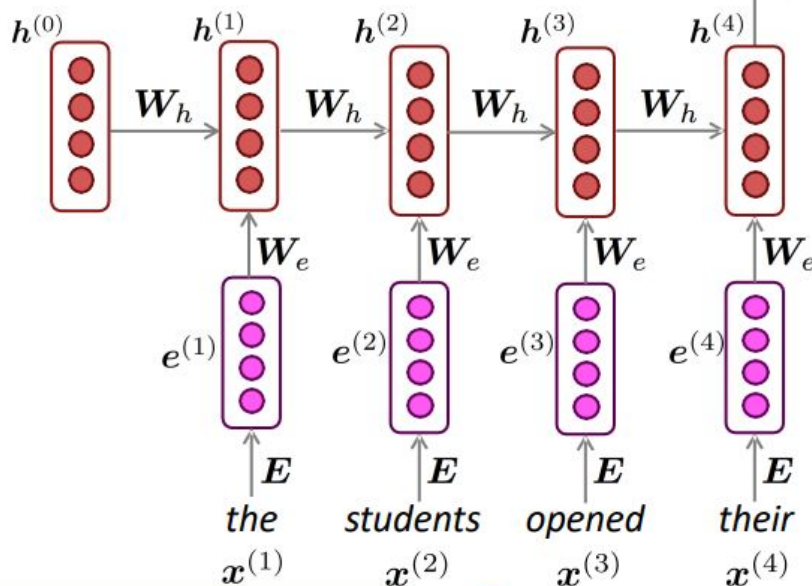
$h^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



Note: this input sequence could be much longer now!

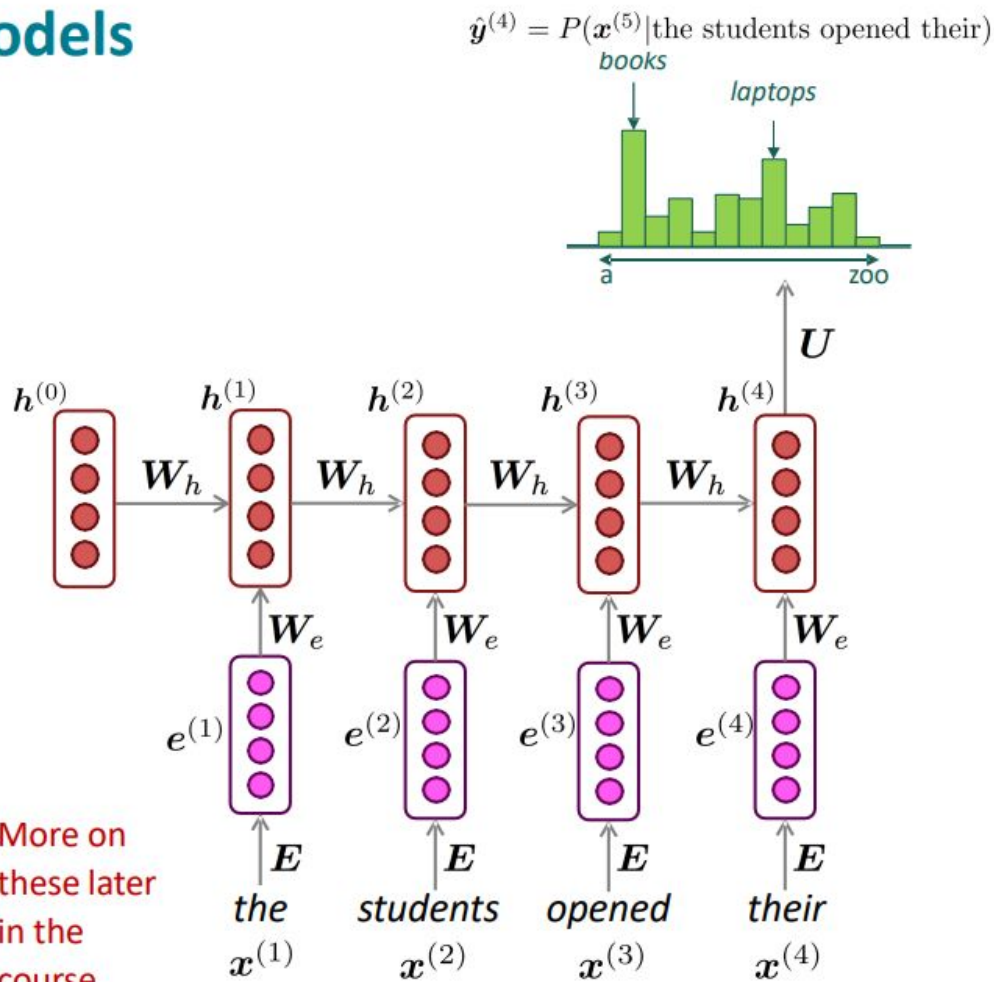
RNN Language Models

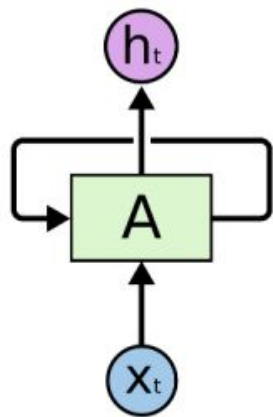
RNN Advantages:

- Can process **any length** input
- Computation for step t can (in theory) use information from **many steps back**
- **Model size doesn't increase** for longer input context
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

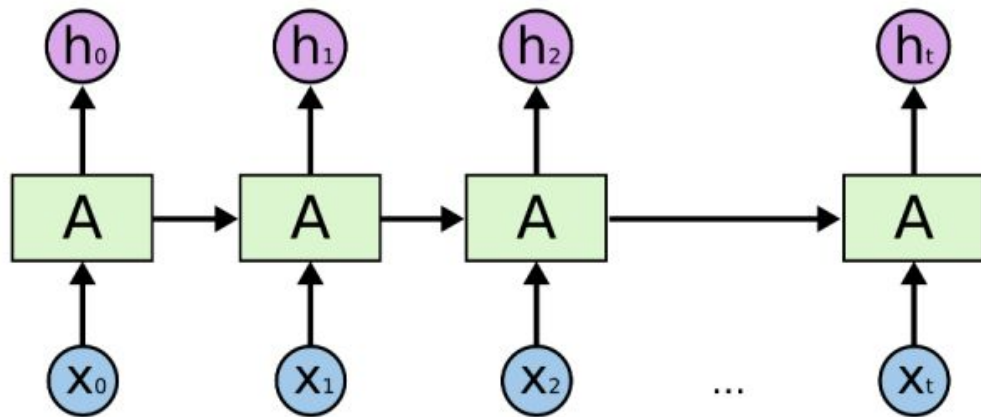
RNN Disadvantages:

- Recurrent computation is **slow**
 - In practice, difficult to access information from **many steps back**
- More on these later in the course



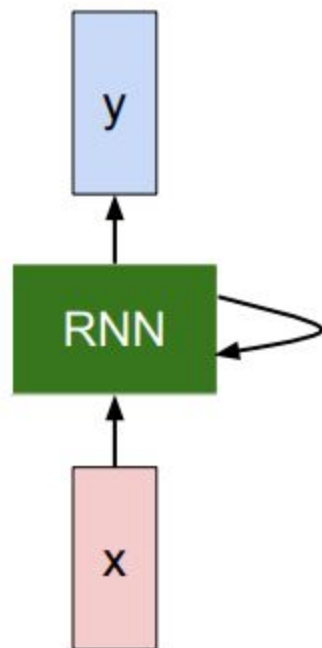


=



(Simple) Recurrent Neural Network

The state consists of a single “hidden” vector h :



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



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

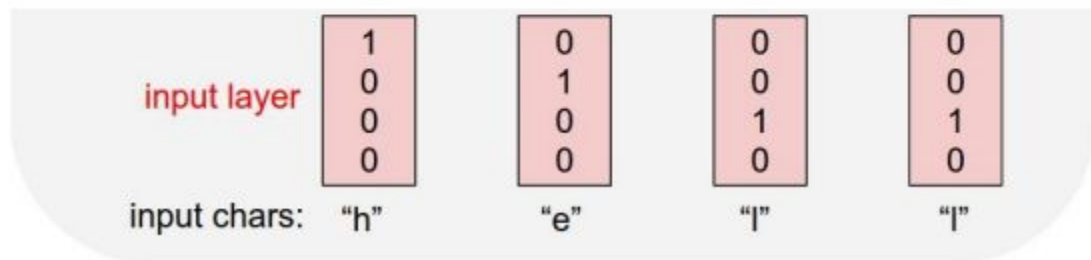
$$y_t = W_{hy}h_t$$

Sometimes called a “Vanilla RNN” or an “Elman RNN” after Prof. Jeffrey Elman

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

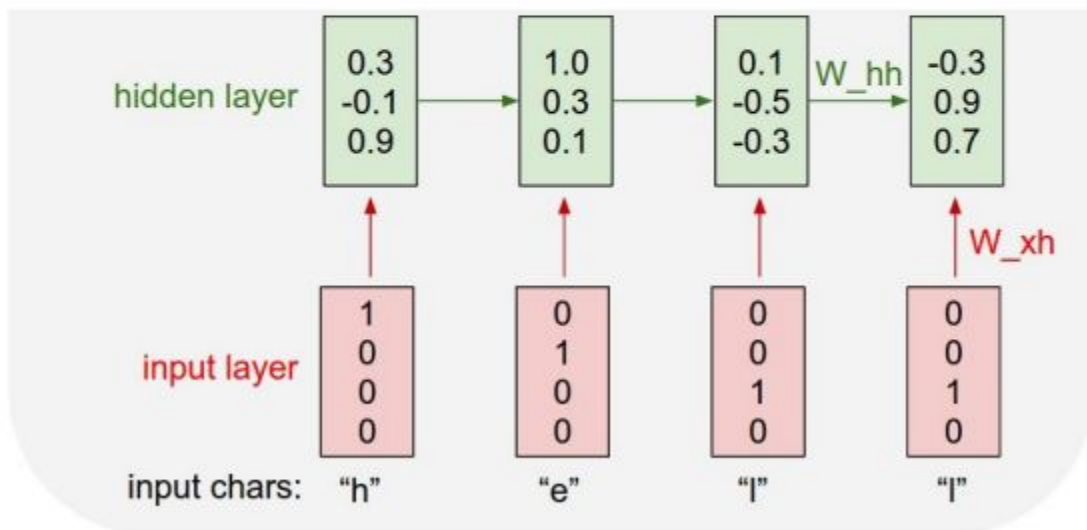


Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

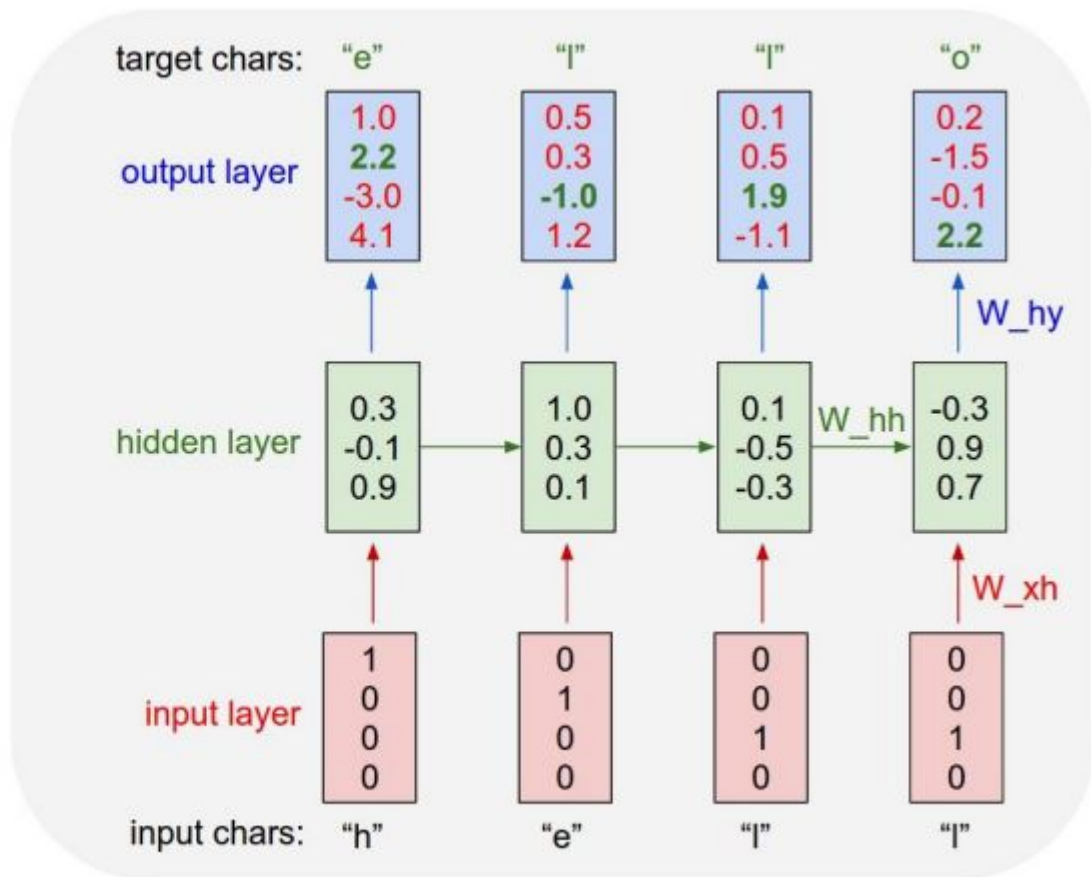
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

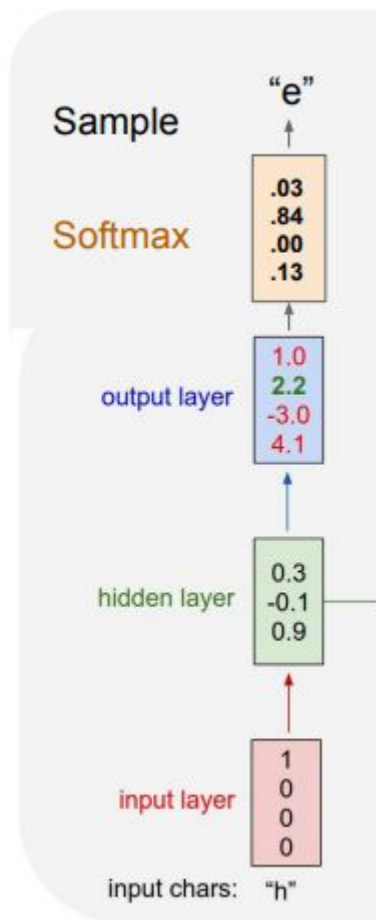
Example training
sequence:
“hello”



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

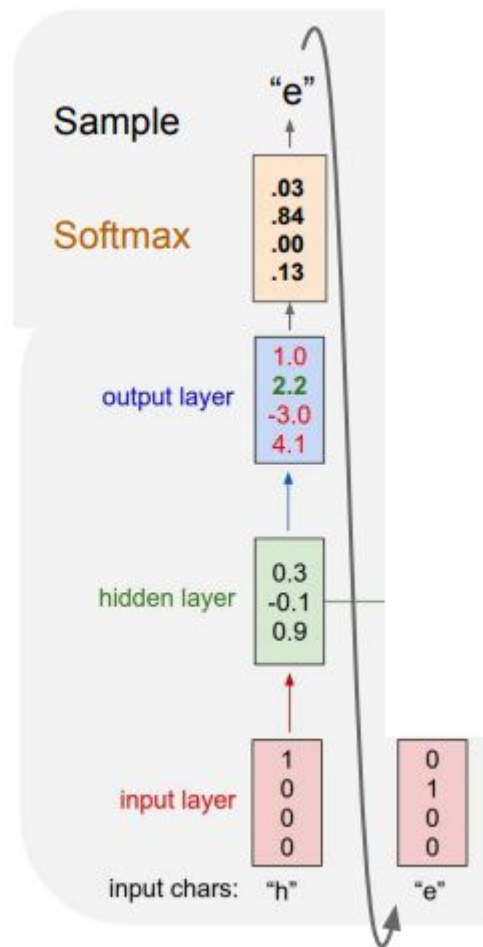
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

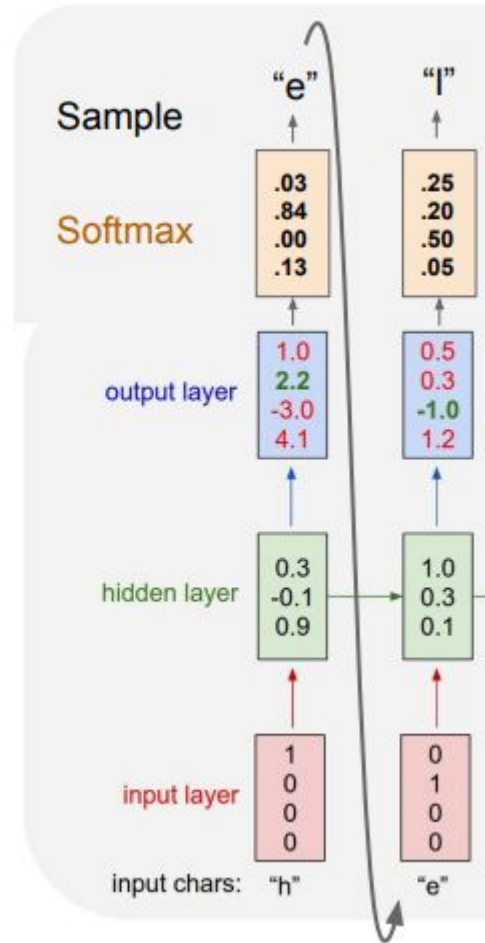
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

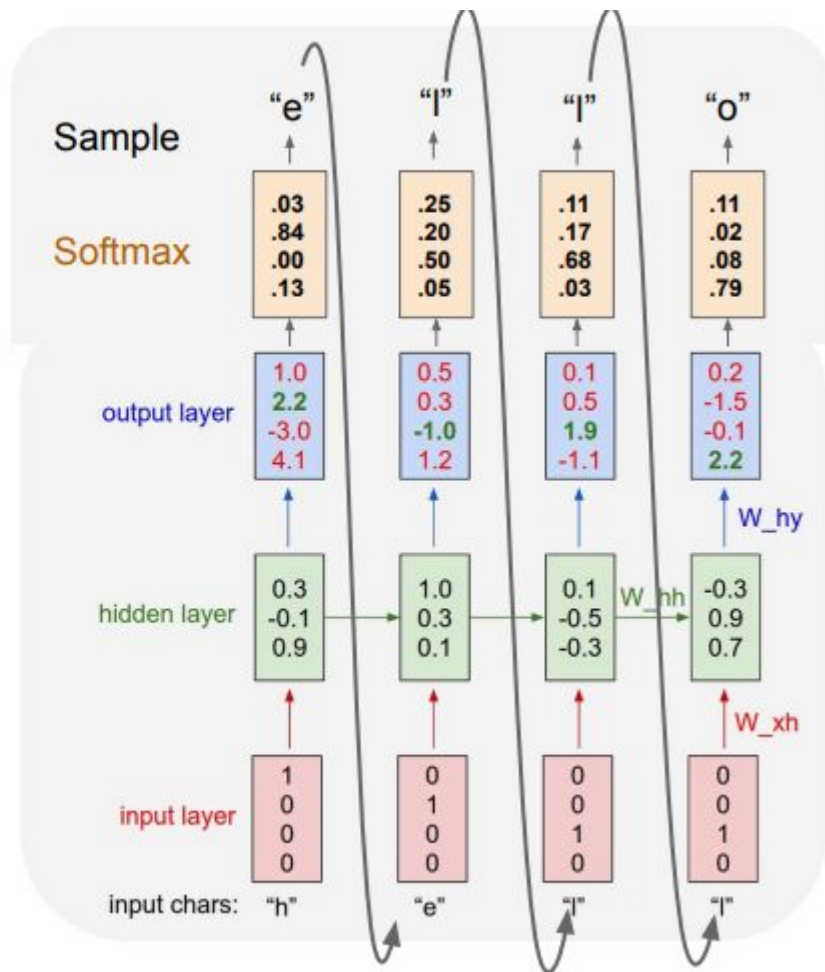
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample
characters one at a time,
feed back to model



CODE

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
texts = ["I love this product", "This is terrible", "Awesome!", "Waste of money"]
labels = [1, 0, 1, 0] # 1 for positive, 0 for negative
# Tokenize the text data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

```
# Padding sequences to have the same length
max_sequence_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_sequence_length, padding='post')
```

CODE

```
model = Sequential()  
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=16,  
input_length=max_sequence_length))  
model.add(SimpleRNN(8, activation='tanh'))  
model.add(Dense(1, activation='sigmoid'))
```

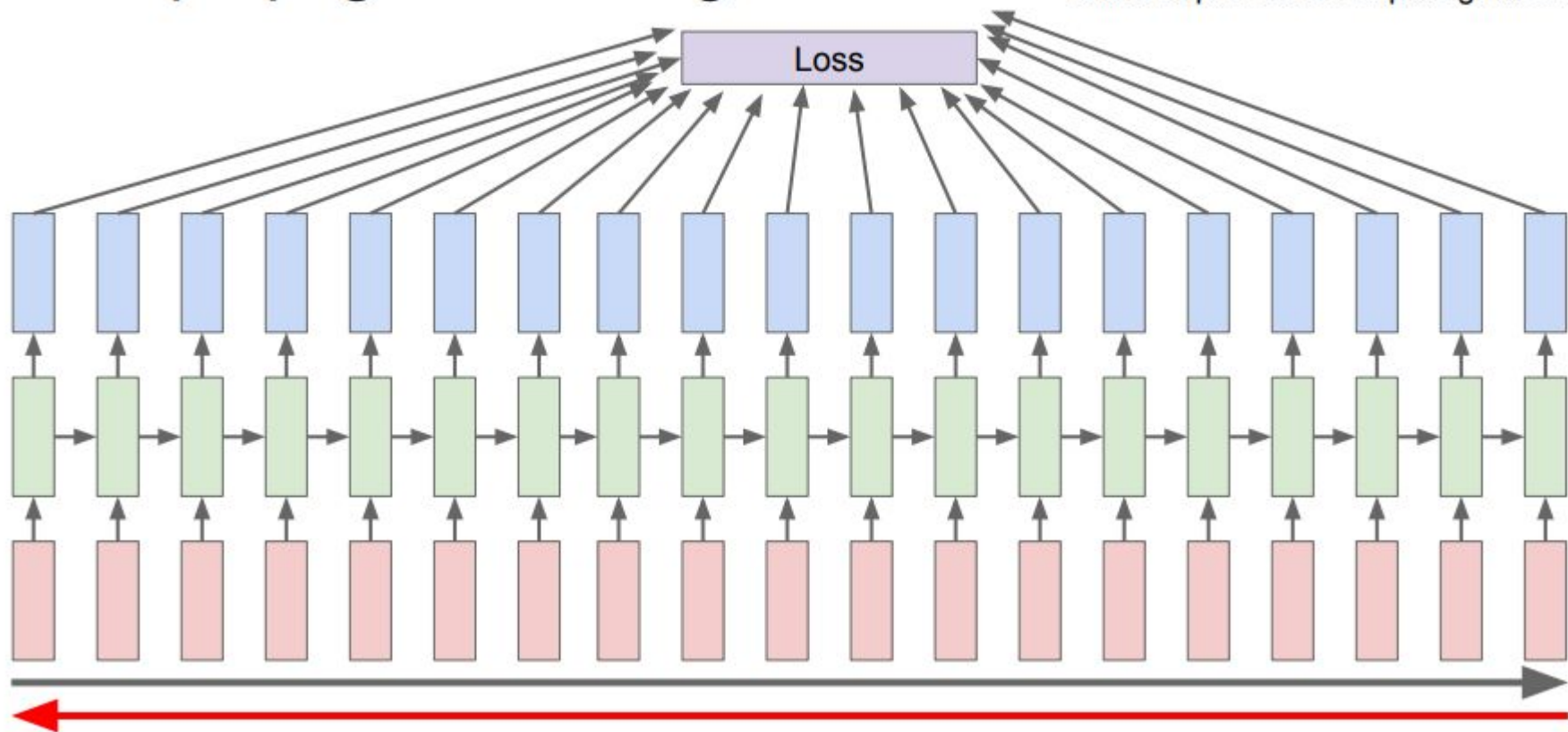
```
model.compile(optimizer='adam', loss='binary_crossentropy',  
metrics=['accuracy'])
```

```
labels = np.array(labels)  
model.fit(sequences, labels, epochs=10, batch_size=2)
```

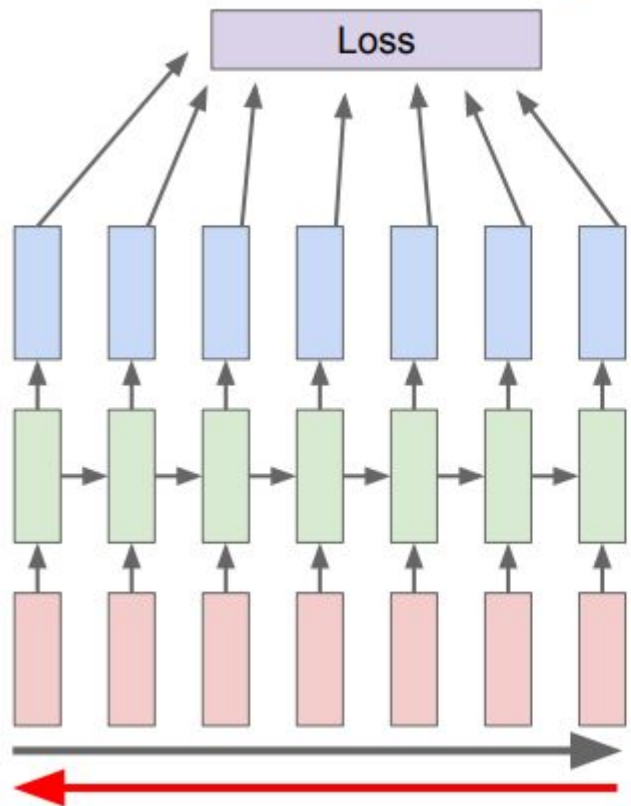
```
test_text = ["I hate it", "Amazing product"]  
test_sequences = tokenizer.texts_to_sequences(test_text)  
test_sequences = pad_sequences(test_sequences, maxlen=max_sequence_length,  
padding='post')  
predictions = model.predict(test_sequences)  
print(predictions)
```

Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

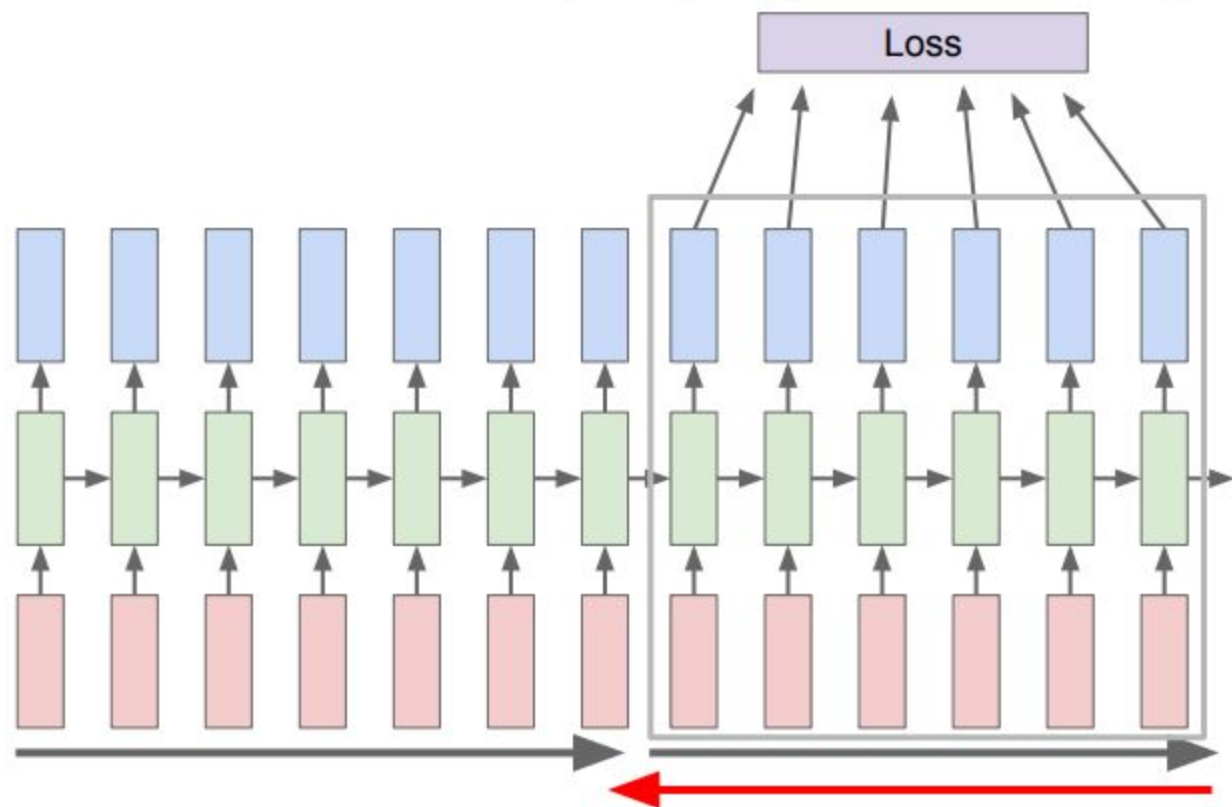


Truncated Backpropagation through time



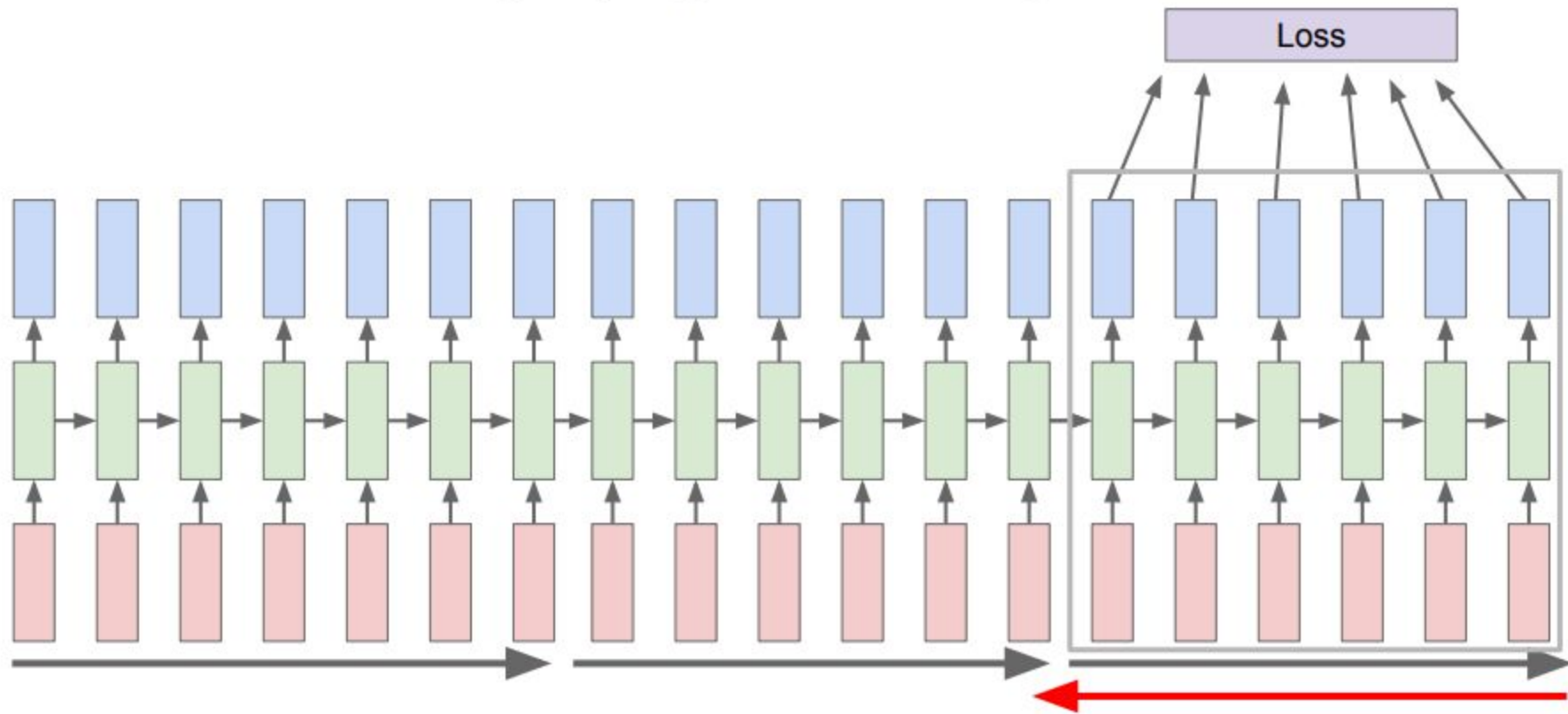
Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation through time

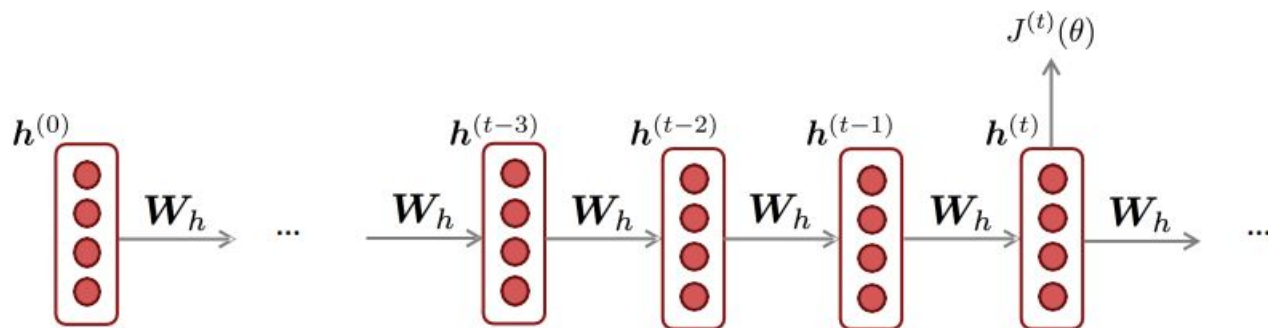


Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Truncated Backpropagation through time



Backpropagation for RNNs



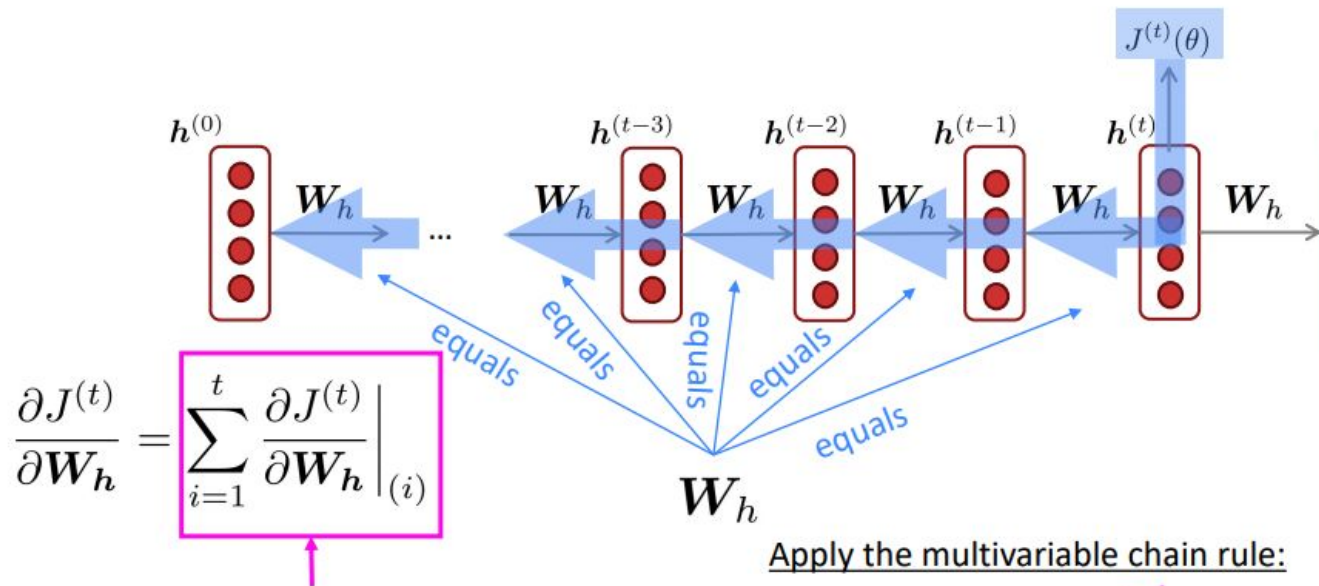
Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the **repeated** weight matrix W_h ?

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

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

Why?

Training the parameters of RNNs: Backpropagation for RNNs



In practice, often "truncated" after ~ 20 timesteps for training efficiency reasons

Question: How do we calculate this?

Answer: Backpropagate over timesteps $i = t, \dots, 0$, summing gradients as you go. This algorithm is called "**backpropagation through time**" [Werbos, P.G., 1988, *Neural Networks 1*, and others]

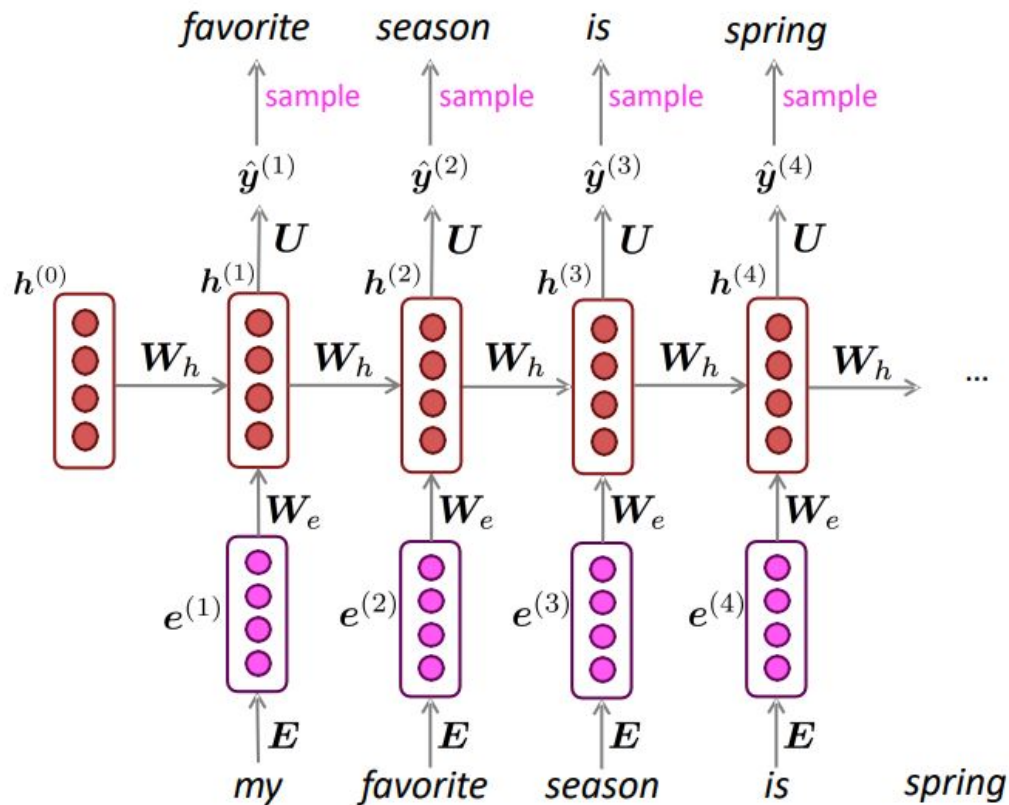
Apply the multivariable chain rule:

$$\begin{aligned} \frac{\partial J^{(t)}}{\partial W_h} &= \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial W_h} \Big|_{(i)} \frac{\partial W_h}{\partial W_h} \Big|_{(i)} \\ &= \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial W_h} \Big|_{(i)} \end{aligned}$$

= 1

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 becomes next step's input.

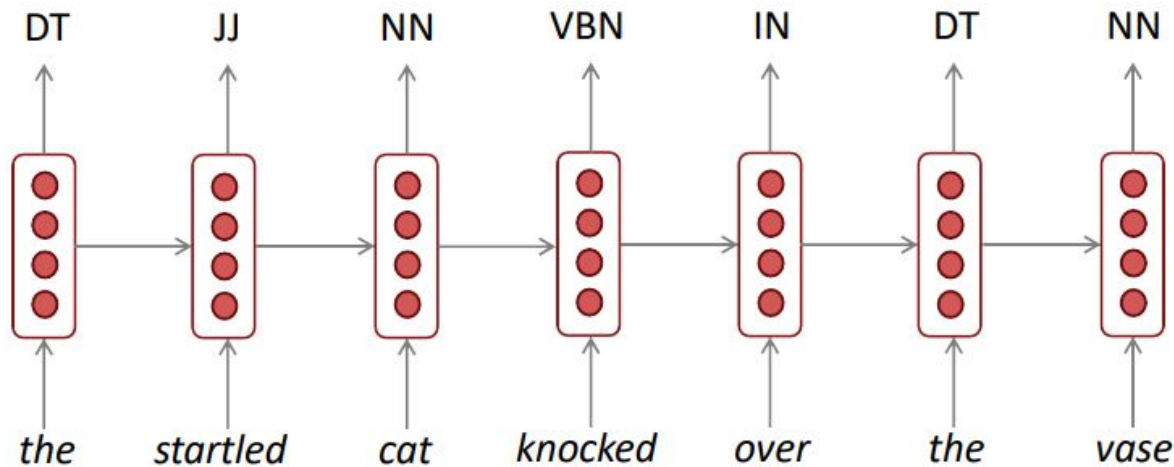


Recap

- **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've shown that RNNs are a great way to build a LM.
- But RNNs are useful for much more!

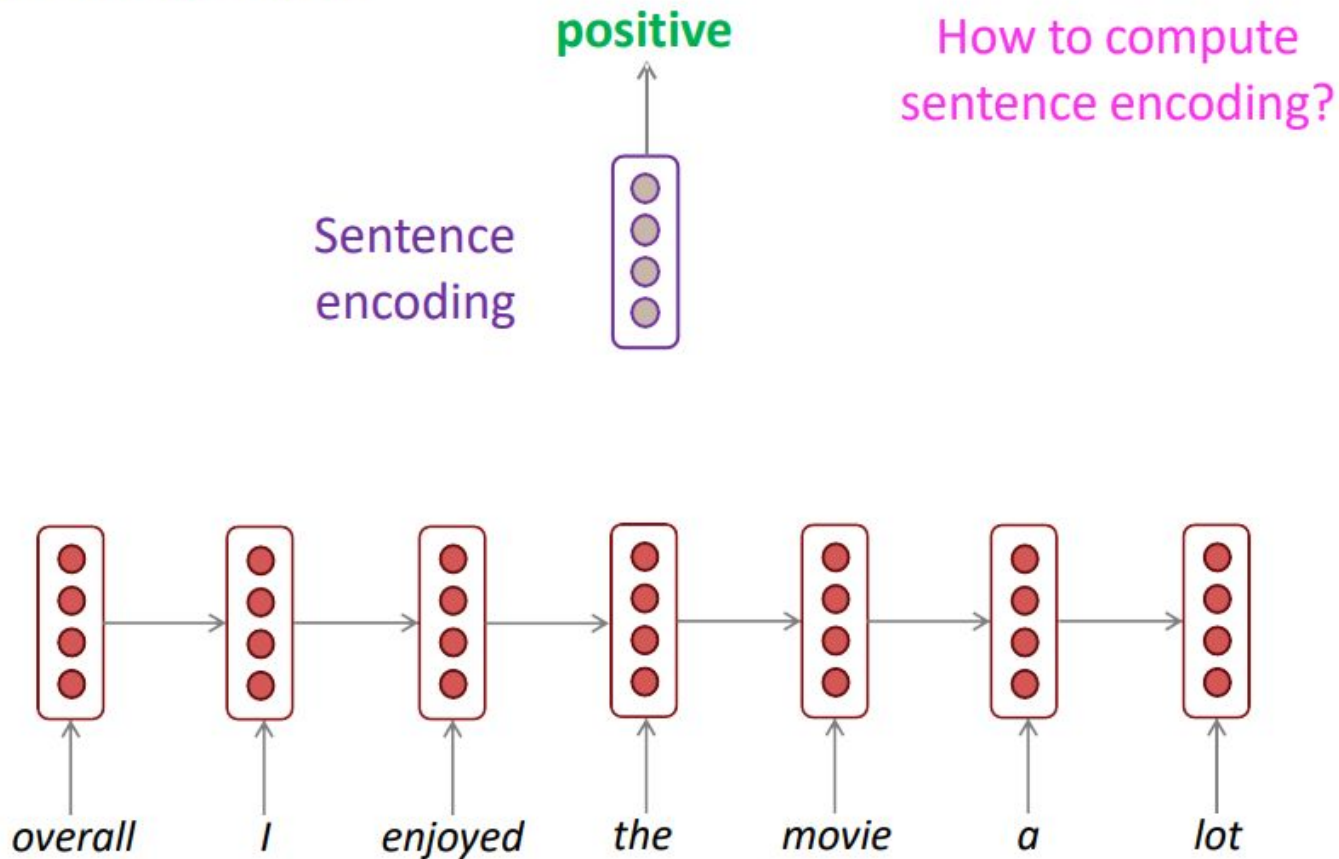
RNNs can be used for tagging

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



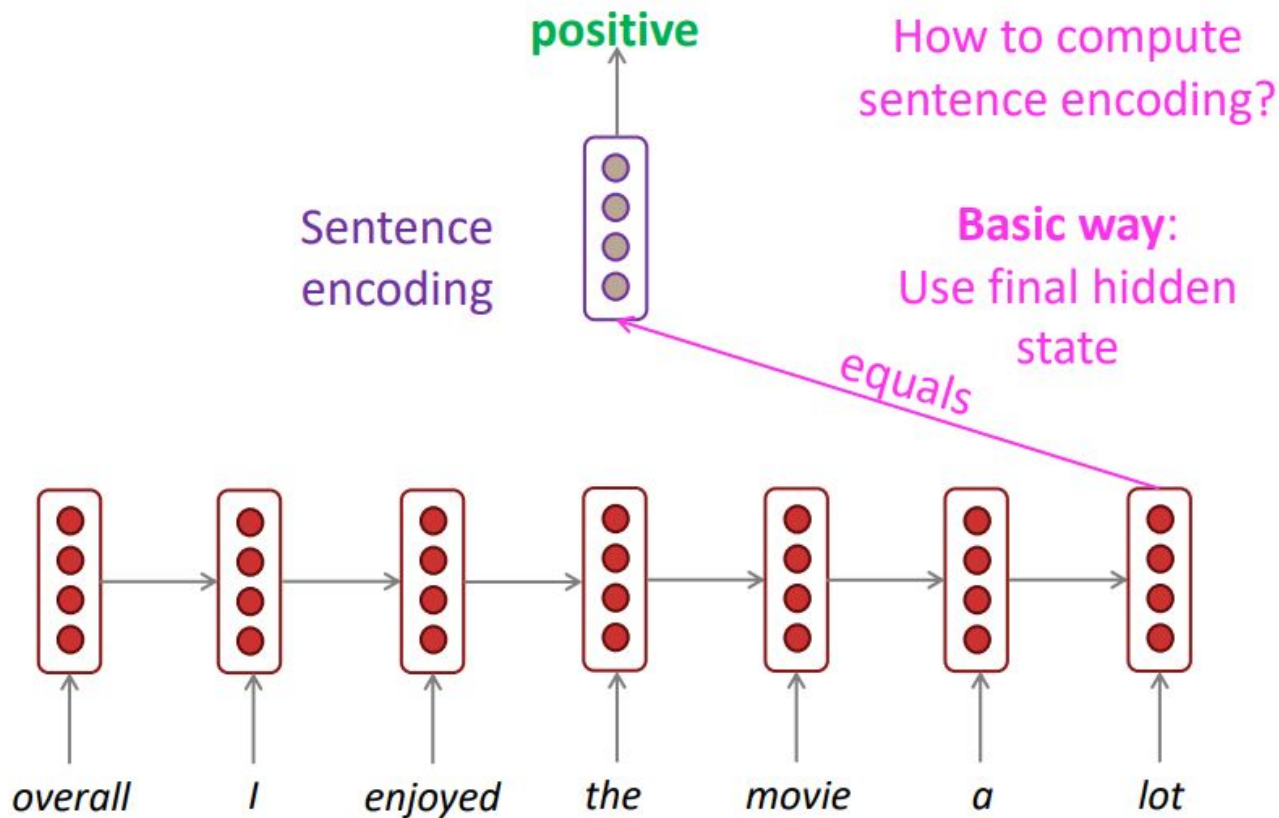
RNNs can be used for sentence classification

e.g., sentiment classification



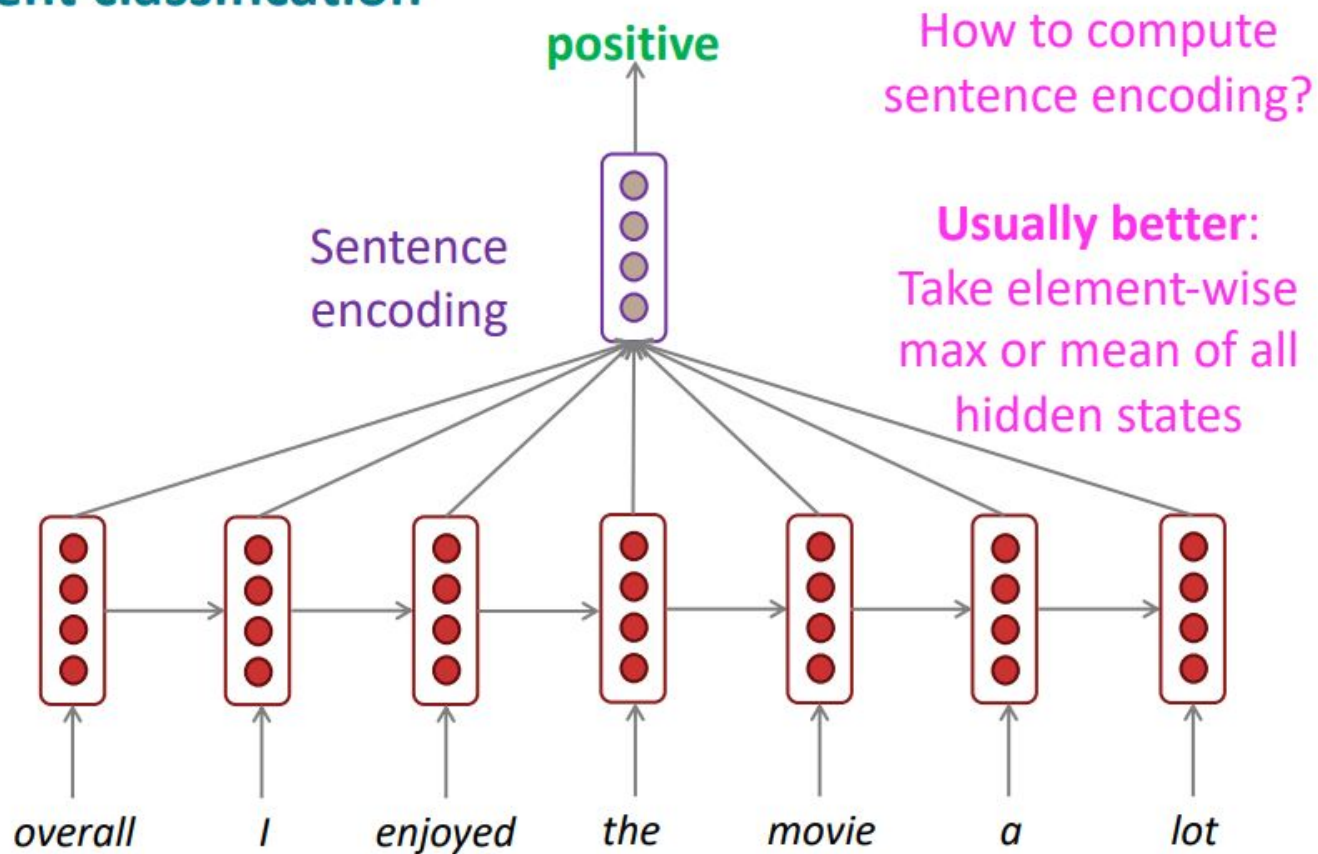
RNNs can be used for sentence classification

e.g., sentiment classification



RNNs can be used for sentence classification

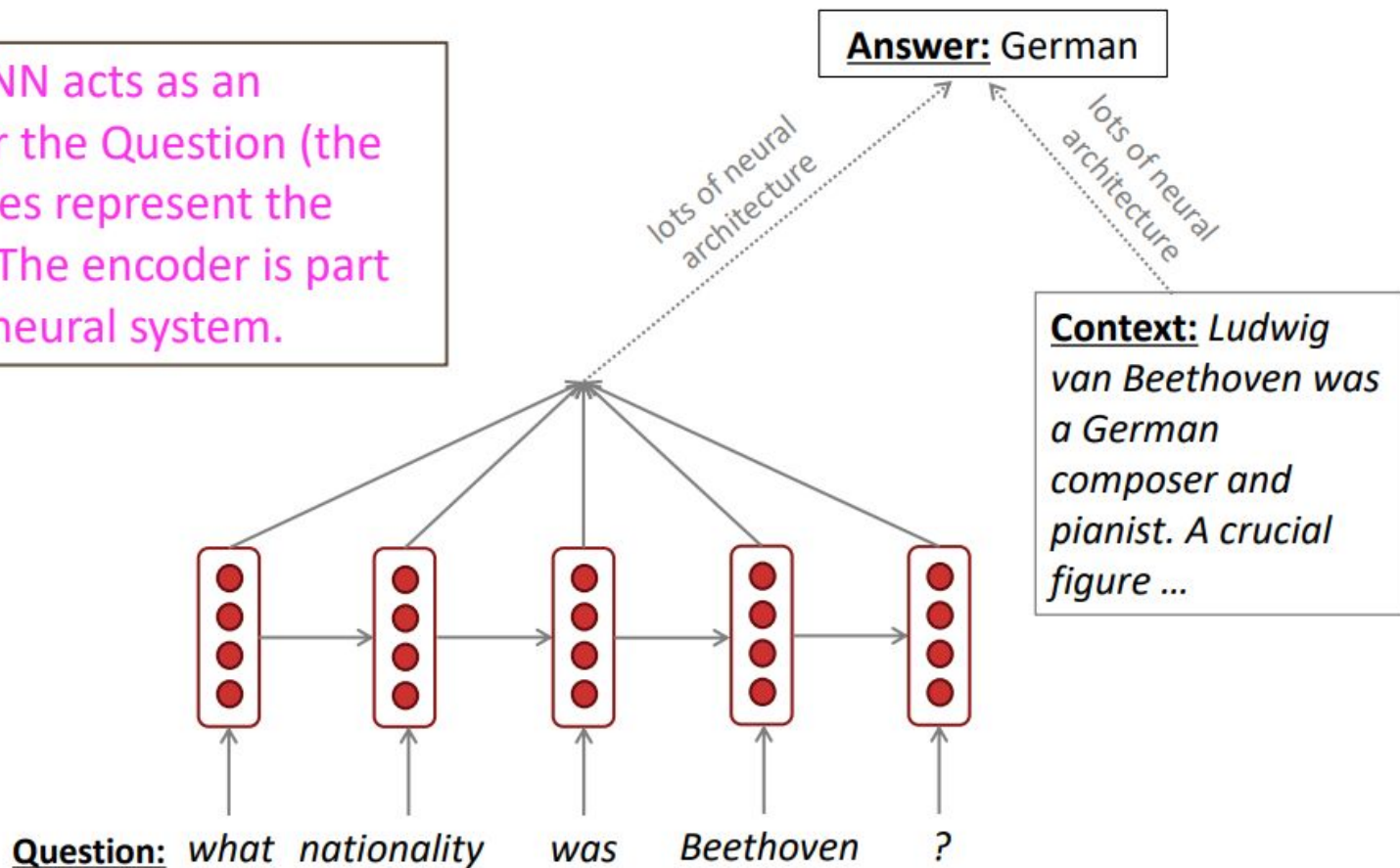
e.g., sentiment classification



RNNs can be used as an encoder module

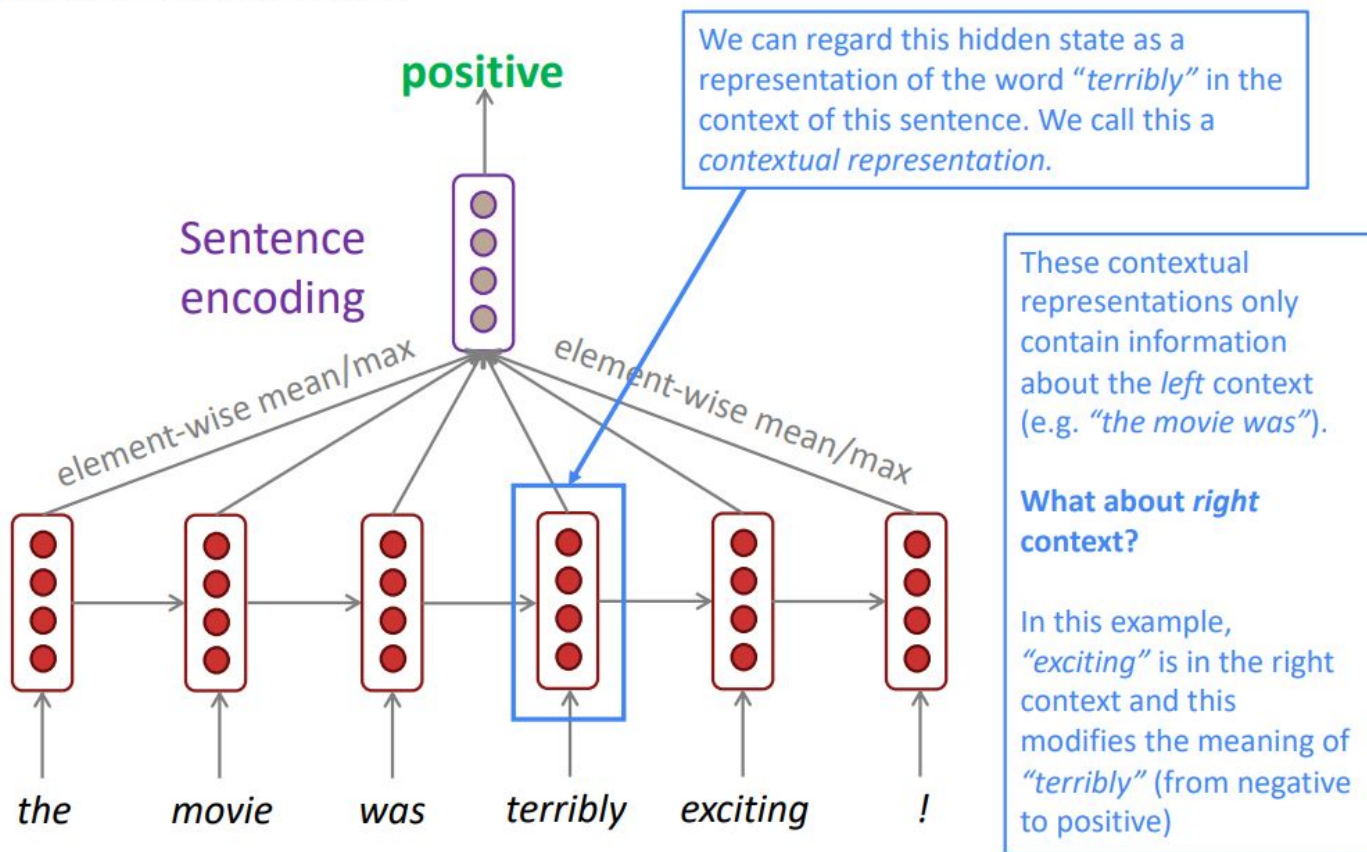
e.g., **question answering**, machine translation, *many other tasks!*

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.



Bidirectional and Multi-layer RNNs: motivation

Task: Sentiment Classification



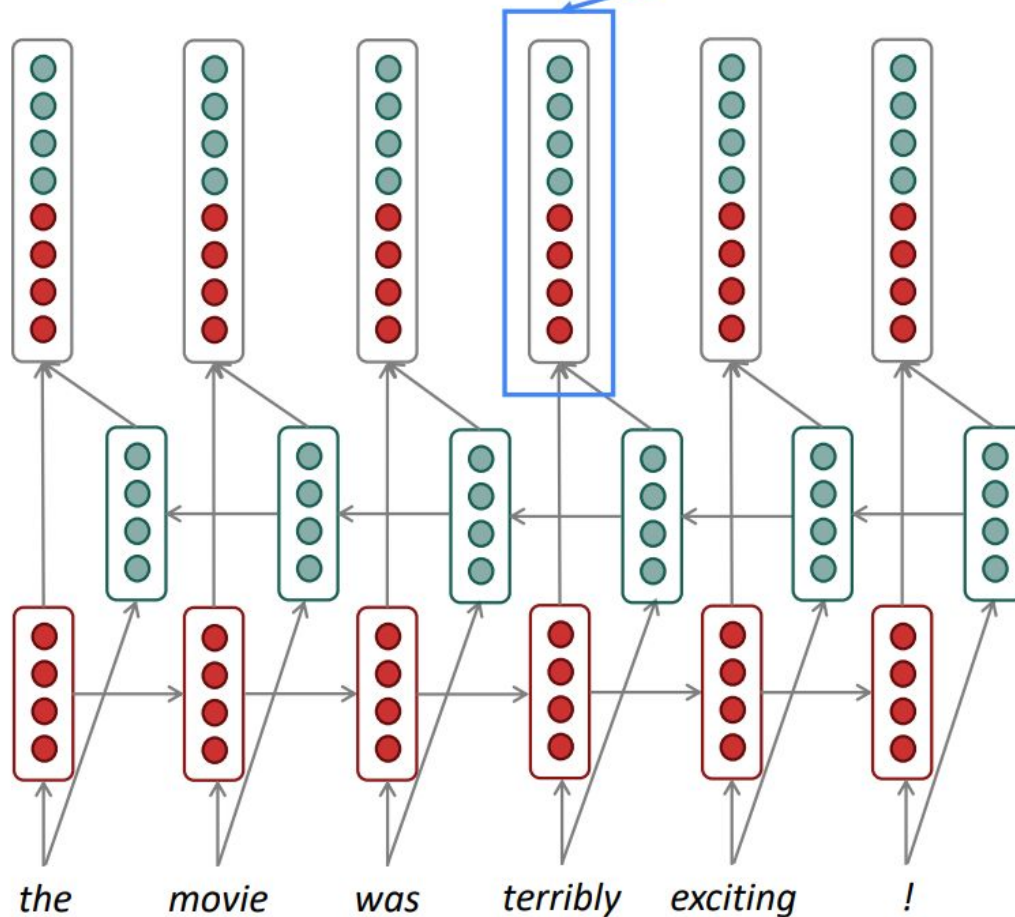
Bidirectional RNNs

This contextual representation of "terribly" has both left and right context!

Concatenated
hidden states

Backward RNN

Forward RNN



Bidirectional RNNs

On timestep t :

This is a general notation to mean “compute one forward step of the RNN” – it could be a simple RNN or LSTM computation.

Forward RNN $\vec{h}^{(t)} = \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)})$

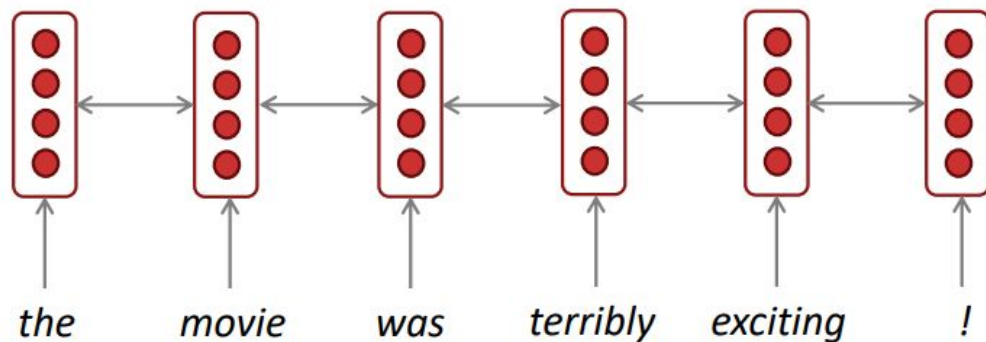
Backward RNN $\overleftarrow{h}^{(t)} = \text{RNN}_{\text{BW}}(\overleftarrow{h}^{(t+1)}, \mathbf{x}^{(t)})$

} Generally, these two RNNs have separate weights

Concatenated hidden states $\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$

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

Bidirectional RNNs: simplified diagram



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

Bidirectional RNNs

- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**
 - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g., any kind of encoding), **bidirectionality is powerful** (you should use it by default).
- For example, **BERT** (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system **built on bidirectionality**.
 - You will learn more about **transformers**, including BERT, in a couple of weeks!

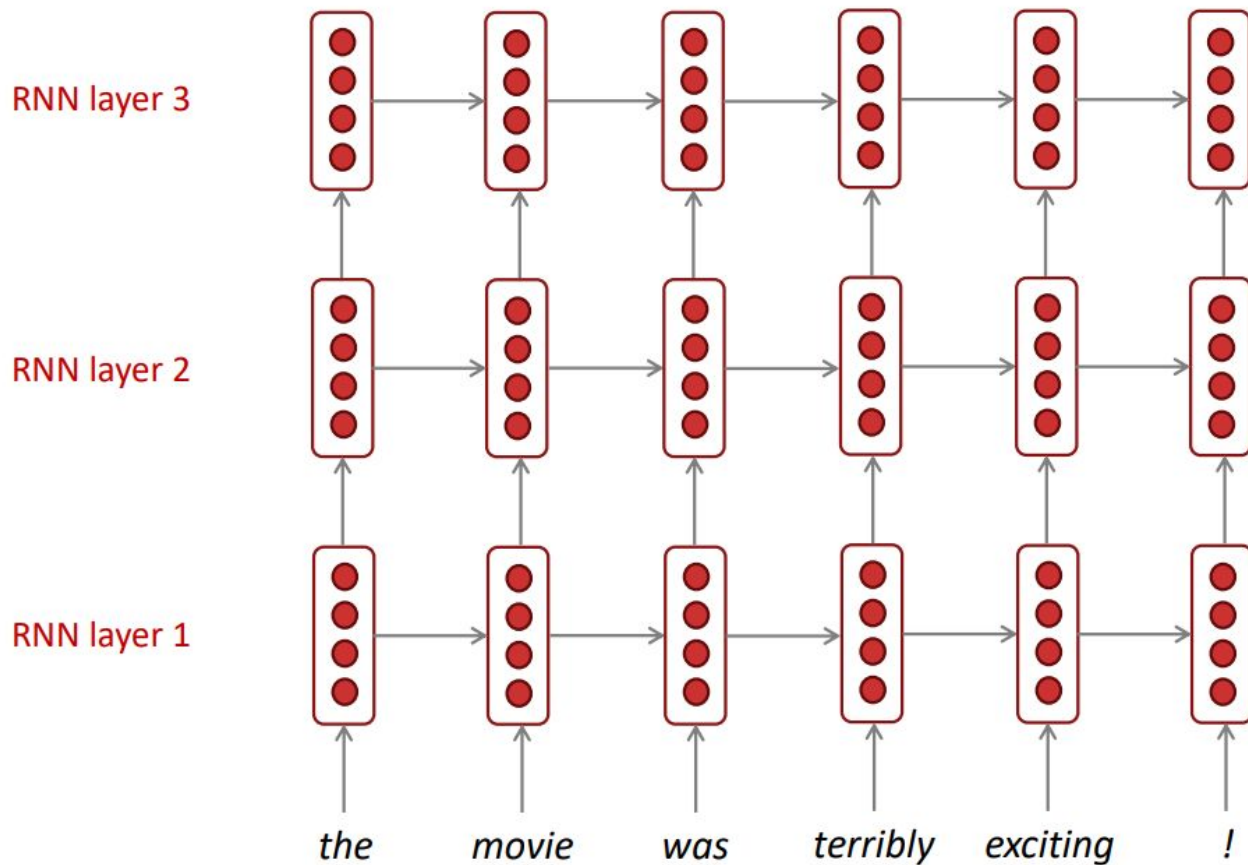
Multi-layer RNNs

- 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 $i+1$



Multi-layer RNNs in practice

- Multi-layer or stacked RNNs allow a network to compute **more complex representations**
 - they work better than just have one layer of high-dimensional encodings!
 - The **lower RNNs** should **compute lower-level features** and the **higher RNNs** should compute **higher-level features**.
- **High-performing RNNs are usually multi-layer** (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al. find that for Neural Machine Translation, **2 to 4 layers** is best for the encoder RNN, and **4 layers** is best for the decoder RNN
 - Often 2 layers is a lot better than 1, and 3 might be a little better than 2
 - Usually, **skip-connections/dense-connections** are needed to train deeper RNNs (e.g., **8 layers**)
- **Transformer-based networks** (e.g., BERT) are usually deeper, like **12 or 24 layers**.
 - You will learn about Transformers later; they have a lot of skipping-like connections

REFERENCES

<https://web.stanford.edu/class/cs224n/slides/cs224n-2023-lecture05-rnnlm.pdf>

http://cs231n.stanford.edu/slides/2020/lecture_10.pdf

<https://web.stanford.edu/class/cs224n/slides/cs224n-2021-lecture06-fancy-rnn.pdf>