



# Language Modeling



SLP3 Ch 3, 9, 10; INLP Ch 6, 18.3

# Probabilistic language modeling (LM)

---

- ▶ **Goal:** compute the probability of a sentence (sequence of words)

$$P(w_1, w_2, w_3, \dots, w_n)$$

- ▶ Applications

- ▶ Machine Translation

- ▶  $P(\text{high winds tonight}) > P(\text{large winds tonight})$

- ▶ Spell Correction

- ▶ “The office is about fifteen **minuets** from my house”
    - ▶  $P(\text{about fifteen minutes from}) > P(\text{about fifteen minuets from})$



# Probabilistic language modeling (LM)

---

- ▶ **Goal:** compute the probability of a sentence (sequence of words)

$$P(w_1, w_2, w_3, \dots, w_n)$$

- ▶ Applications

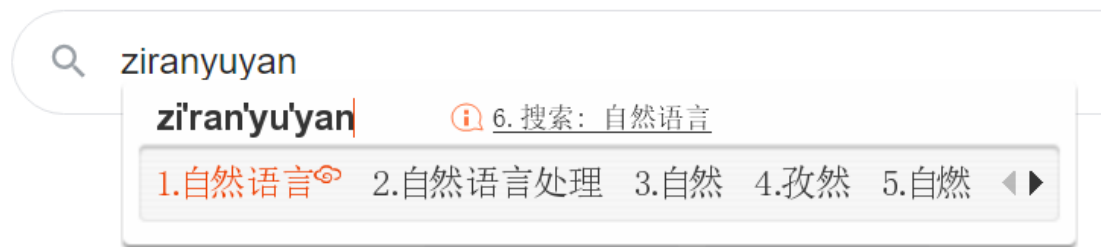
- ▶ Speech Recognition

- ▶  $P(\text{I saw a van}) \gg P(\text{eyes awe of an})$

- ▶ Chinese IME

- ▶  $P(\text{自然语言}) \gg P(\text{孜然鱼雁})$

- ▶ ...



# A trivial LM

---

- ▶ Estimate sentence probability from its frequency in a corpus

$$P(w_1, w_2, w_3, \dots, w_n) = \frac{\text{count}(w_1, \dots, w_n)}{N}$$

- ▶ Will this work?
  - ▶ No! Impossible for a corpus to cover all possible sentences



# Applying chain rule

---

- ▶ The Chain Rule applied to compute joint probability of words in sentence

$$P(w_1, w_2, \dots, w_n) = \prod_i P(w_i | w_1, w_2, \dots, w_{i-1})$$

- ▶ Ex. Calculate the probability of the sentence:

“its water is so transparent”

- ▶  $P(\text{its water is so transparent}) =$   
 $P(\text{its}) \times P(\text{water}|\text{its}) \times P(\text{is}|\text{its water}) \times$   
 $P(\text{so}|\text{its water is}) \times P(\text{transparent}|\text{its water is so})$



# Applying chain rule

---

- ▶ How to estimate these conditional probabilities?
- ▶ Could we just count and divide?
  - ▶  $P(\text{the}|\text{its water is so transparent that}) = \frac{\text{Count}(\text{its water is so transparent that the})}{\text{Count}(\text{its water is so transparent that})}$
- ▶ Will this work?
  - ▶ No! Once again, impossible for a corpus to cover all possible texts



# Methods

---

- ▶ n-gram LM
  - ▶ Probability of each word is conditioned on the preceding  $n-1$  words.
- ▶ Recurrent neural networks
  - ▶ Probability of each word is conditioned on a hidden vector summarizing all the preceding words
- ▶ Transformers
  - ▶ Probability of each word is computed by attending to preceding words
- ▶ Other methods
  - ▶ Ex: grammar-based



# Problem: unknown words

---

- ▶ Often we don't know all the words in advance.
  - ▶ How to deal with the Out Of Vocabulary (OOV) words?
- ▶ Create an unknown word token <UNK>
  - ▶ Training of <UNK> probabilities
    - ▶ Create a fixed lexicon L
    - ▶ During text normalization, any word not in L changed to <UNK>
    - ▶ Now we can estimate its probabilities like a normal word
  - ▶ At test time
    - ▶ Use <UNK> probabilities for any word not in L
- ▶ Alternative
  - ▶ Build a language model at the character level
  - ▶ ...or at the subword level





# Evaluation

---

- ▶ Extrinsic evaluation
  - ▶ Performance on downstream tasks
- ▶ Problems
  - ▶ Time-consuming
  - ▶ Results from different tasks may differ



# Evaluation

---

- ▶ Intrinsic evaluation: perplexity
  - ▶ Intuitively, language models should assign high probability to real language they have not seen before.
- ▶ For test data  $\bar{\mathbf{x}}_{1:m}$  ( $m$  sentences)
  - ▶ Average log-probability per word of  $\bar{\mathbf{x}}_{1:m}$  is

$$l = -\frac{1}{M} \sum_{i=1}^m \log_2 p(\bar{\mathbf{x}}_i)$$

if  $M = \sum_{i=1}^m |\bar{\mathbf{x}}_i|$  (total number of words in the corpus)

- ▶ This is the average number of bits required to encode each word
  - ▶ Perplexity (relative to  $\bar{\mathbf{x}}_{1:m}$ ) is  $2^l$ 
    - ▶ Lower is better



# Evaluation

---

$$2^{-\frac{1}{M} \sum_{i=1}^m \log_2 p(\bar{x}_i)}$$

- ▶ Assign probability of 1 to the test data  $\Rightarrow$  perplexity = 1
- ▶ Assign probability of  $\frac{1}{|\mathcal{V}|}$  to every word  $\Rightarrow$  perplexity =  $|\mathcal{V}|$
- ▶ Assign probability of 0 to anything  $\Rightarrow$  perplexity =  $\infty$



# Evaluation

---

- ▶ Caution: never compare the perplexities of LMs with different vocabularies
- ▶ Extreme example
  - ▶ If a LM treats all words as UNK, then its perplexity would be a perfect 1





# N-gram Models



# Markov Assumption

---

- ▶ We approximate each component in the product

$$P(w_i | w_1, w_2, \dots, w_{i-1}) \approx P(w_i | w_{i-k}, \dots, w_{i-1})$$

- ▶ So:

$$\begin{aligned} P(w_1, w_2, \dots, w_n) &= \prod_i P(w_i | w_1, w_2, \dots, w_{i-1}) \\ &\approx \prod_i P(w_i | w_{i-k}, \dots, w_{i-1}) \end{aligned}$$



# N-gram models

---

- ▶ Simplest case: Unigram model

- ▶  $P(w_1, w_2, \dots, w_n) \approx \prod_i P(w_i)$  ← { similar to naïve Bayes }
- ▶ The words' occurrence is independent from each other.

- ▶ Bigram model

- ▶ Condition on the previous word:

$$P(w_i | w_1, w_2, \dots, w_{i-1}) \approx P(w_i | w_{i-1})$$

- ▶ We can extend to trigrams, 4-grams, 5-grams



# N-gram models

---

- ▶ In principle, this is an insufficient model of language
  - ▶ Because language has long-distance dependencies

“The boy who is picking apples is his son.”

- ▶ But n-gram models are surprisingly good for LM





# Estimating bigram probabilities

---

## ▶ Maximum Likelihood Estimate

$$P(w_i | w_{i-n+1}, \dots, w_{i-1}) = \frac{\text{Count}(w_{i-n+1}, \dots, w_{i-1}, w_i)}{\text{Count}(w_{i-n+1}, \dots, w_{i-1})}$$

## ▶ Ex. bigram: $P(w_i | w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

$$P(I | <s>) = \frac{2}{3} = 0.67$$

$$P(</s> | \text{Sam}) = \frac{1}{2} = 0.5$$

$$P(\text{Sam} | <s>) = \frac{1}{3} = 0.33$$

$$P(\text{Sam} | \text{am}) = \frac{1}{2} = 0.5$$

$$P(\text{am} | I) = \frac{2}{3} = 0.67$$

$$P(\text{do} | I) = \frac{1}{3} = 0.33$$



# Problem with MLE

---

- ▶ Data sparseness

- ▶ Some n-grams don't occur in the training set
- ▶ ...but occur in the test set

- ▶ Training set:

- ▶ ... denied the allegations
- ▶ ... denied the reports
- ▶ ... denied the claims
- ▶ ... denied the request

$$P(\text{offer} \mid \text{denied the}) = 0$$

- ▶ Test set:

- ▶ ... denied the offer
- ▶ ... denied the loan

0?!

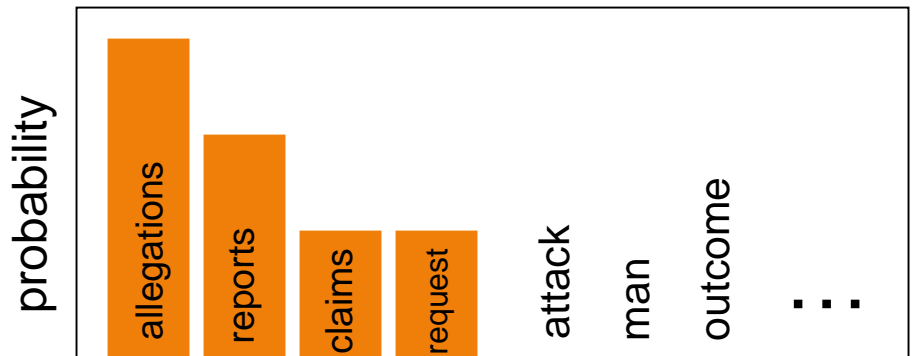


# Smoothing

- ▶ Simple method: add  $\lambda > 0$  to every count (including zero-counts) before normalizing
- ▶ Longstanding champion: modified Kneser-Ney smoothing (Chen and Goodman, 1998)

When we have sparse statistics:

$P(w \text{denied the})$	
3 allegations	
2 reports	
1 claims	
1 request	
<hr/>	
7 total	



# Smoothing

- ▶ Simple method: add  $\lambda > 0$  to every count (including zero-counts) before normalizing
- ▶ Longstanding champion: modified Kneser-Ney smoothing (Chen and Goodman, 1998)

Steal probability mass to generalize better

$P(w|\text{denied the})$

3.5 allegations

2.5 reports

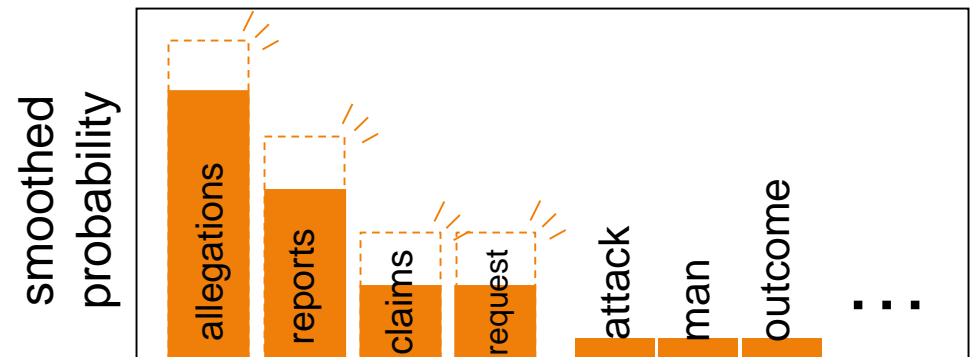
1.5 claims

1.5 request

2.0 other

---

11 total



# Backoff and Interpolation

---

- ▶ General idea
  - ▶ Condition on less context for contexts you haven't learned much about
- ▶ Backoff:
  - ▶ use  $n$ -gram if you have good evidence,
  - ▶ otherwise  $(n-1)$ -gram, otherwise  $(n-2)$ -gram, ...
- ▶ Interpolation:
  - ▶ mix unigram, bigram, trigram, ...
- ▶ Interpolation works better



# Linear Interpolation

---

- ▶ Simple interpolation

- ▶ 
$$\hat{P}(w_n | w_{n-2} w_{n-1}) = \lambda_1 P(w_n | w_{n-2} w_{n-1}) + \lambda_2 P(w_n | w_{n-1}) + \lambda_3 P(w_n)$$

- ▶ 
$$\sum_i \lambda_i = 1$$

- ▶ Lambdas conditional on context

- ▶ 
$$\hat{P}(w_n | w_{n-2} w_{n-1}) = \lambda_1(w_{n-2} w_{n-1}) P(w_n | w_{n-2} w_{n-1}) + \lambda_2(w_{n-2} w_{n-1}) P(w_n | w_{n-1}) + \lambda_3(w_{n-2} w_{n-1}) P(w_n)$$



# Google n-gram viewer

► <https://books.google.com/ngrams>

Google Books Ngram Viewer

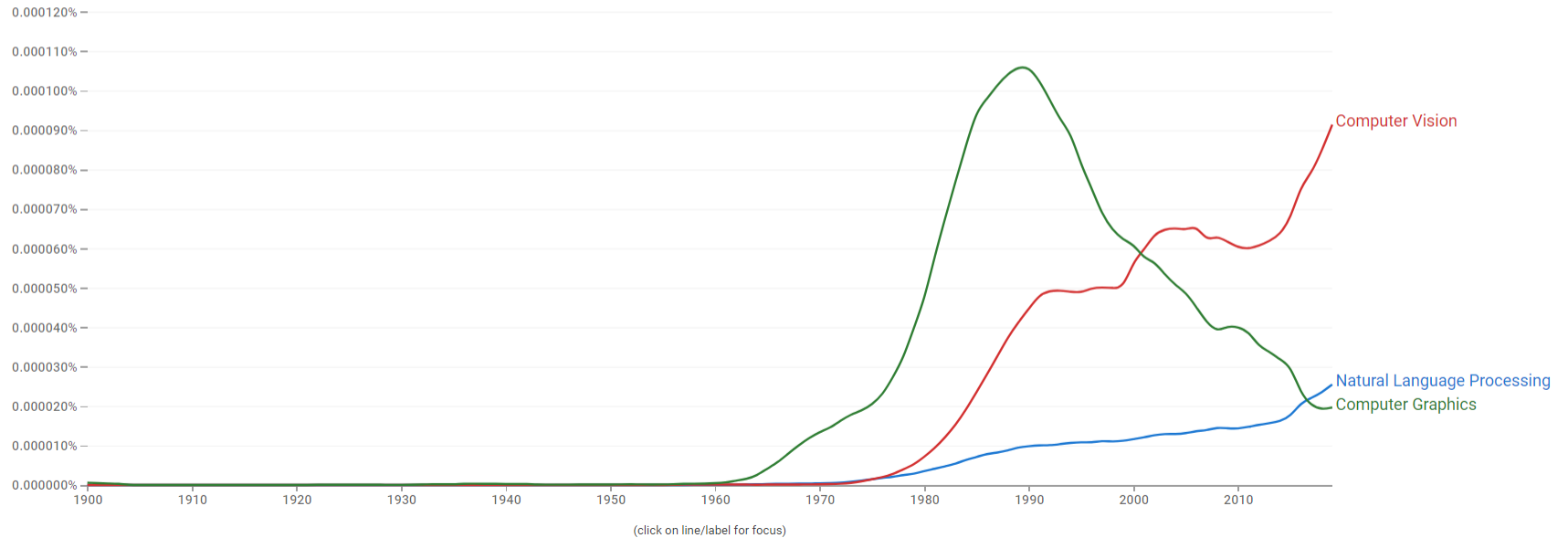
🔍 Natural Language Processing,Computer Vision,Computer Graphics

1900 - 2019

English (2019)

Case-Insensitive

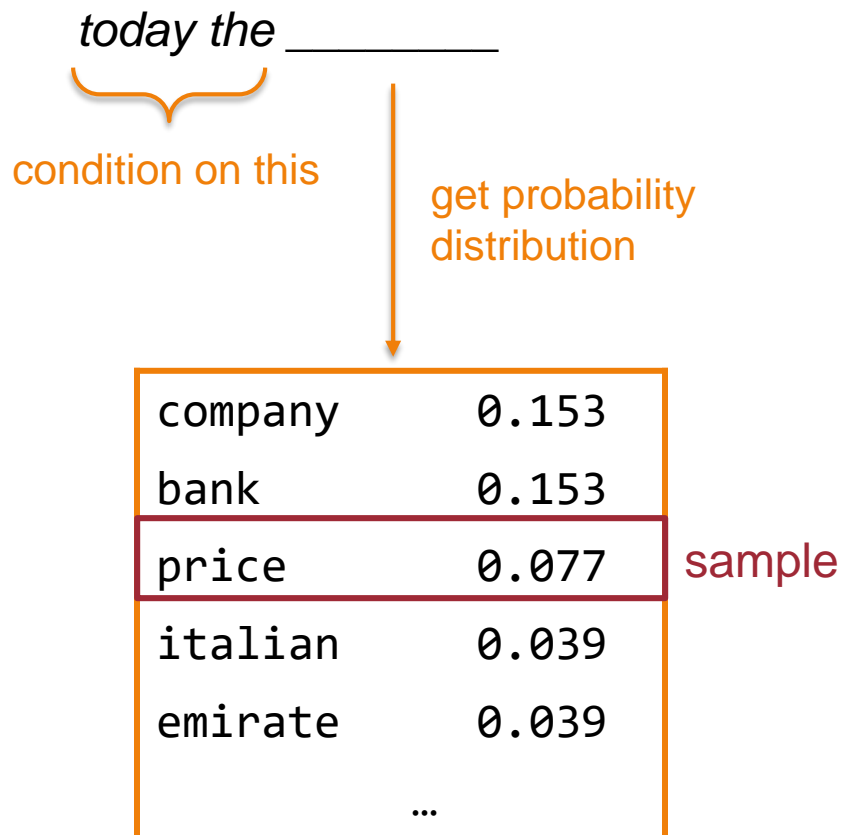
Smoothing



# Generating with a trigram model

---

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

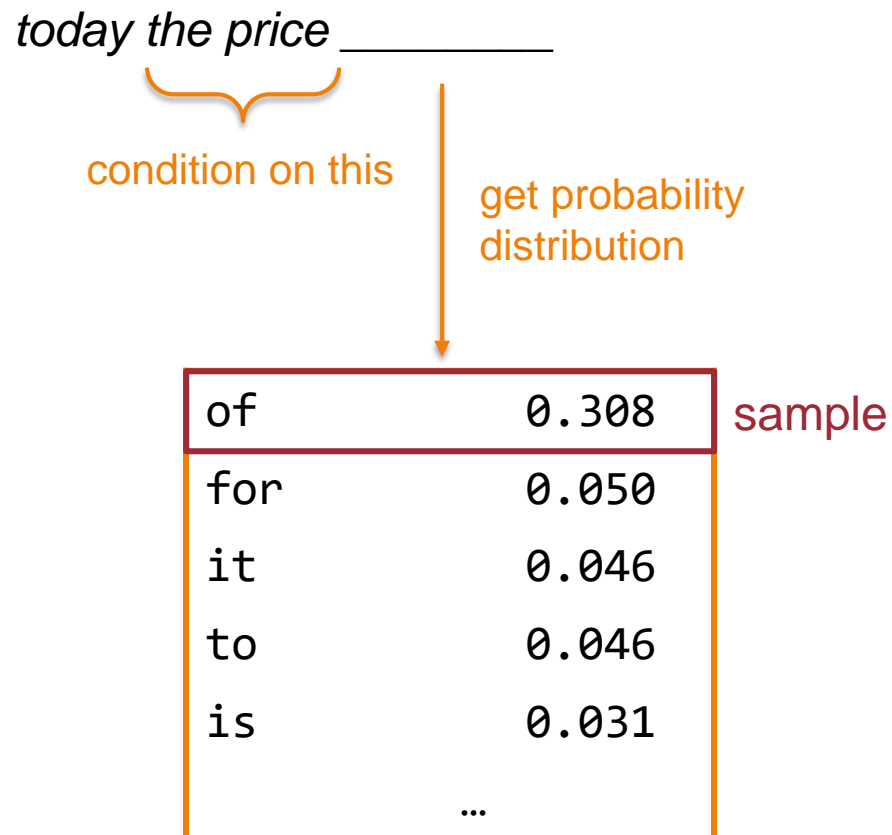




# Generating with a trigram model

---

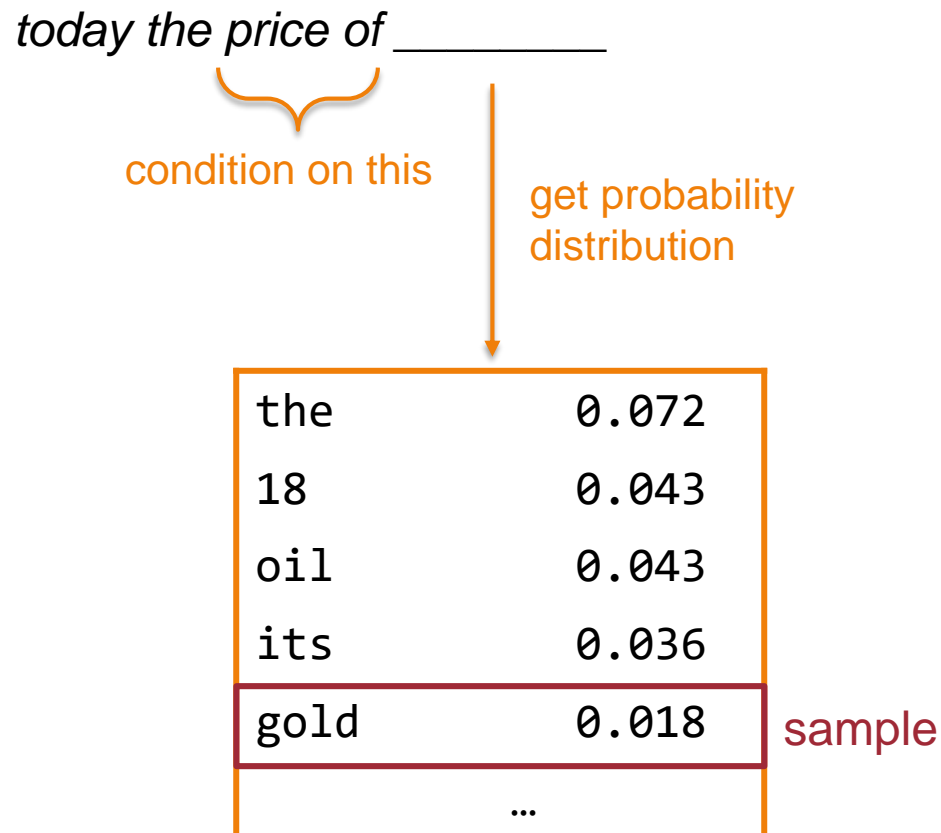
- ▶ You can also use a Language Model to **generate text**.



# Generating with a trigram model

---

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



# Generating with a trigram model

---

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

*today the price of gold \_\_\_\_\_*



# Generating with a trigram 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 .*

Surprisingly grammatical!

...but **incoherent**. We need to consider more than three words at a time if we want to model language well.

But increasing  $n$  worsens sparsity problem, and increases model size...





# Recurrent neural networks



# A fixed window neural language model

---

~~as the proctor started the clock~~ the students opened their \_\_\_\_\_

discard

fixed window



# A fixed window neural language model

output distribution

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

hidden layer

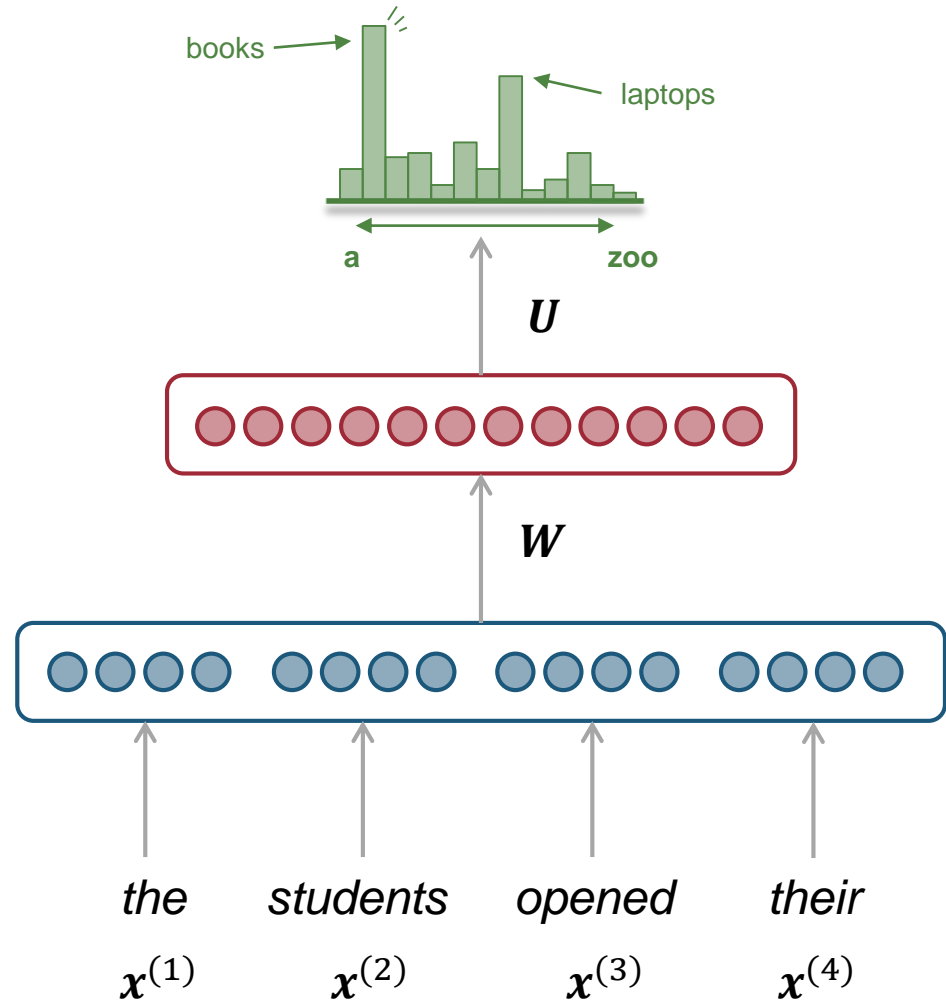
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



# A fixed window neural language model

Improvements over n-gram LM:

- No sparsity problem

Training data:

*I have to make sure that the cat gets fed.*  
(never seen *dog gets fed*)

Test data:

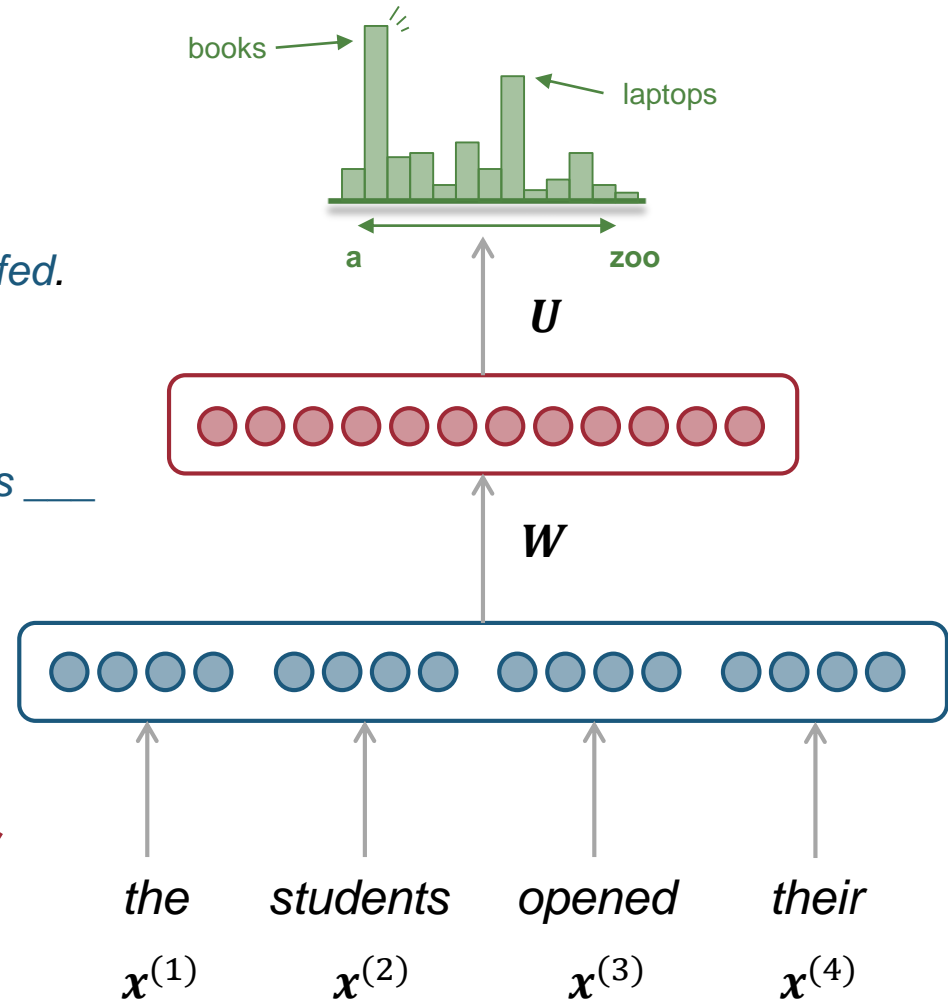
*I forgot to make sure that the dog gets \_\_\_\_*

N-gram LM

What's this... ??

Neural LM

“*cat*” and “*dog*” share similar embeddings... so I'll predict “*fed*” after dog.





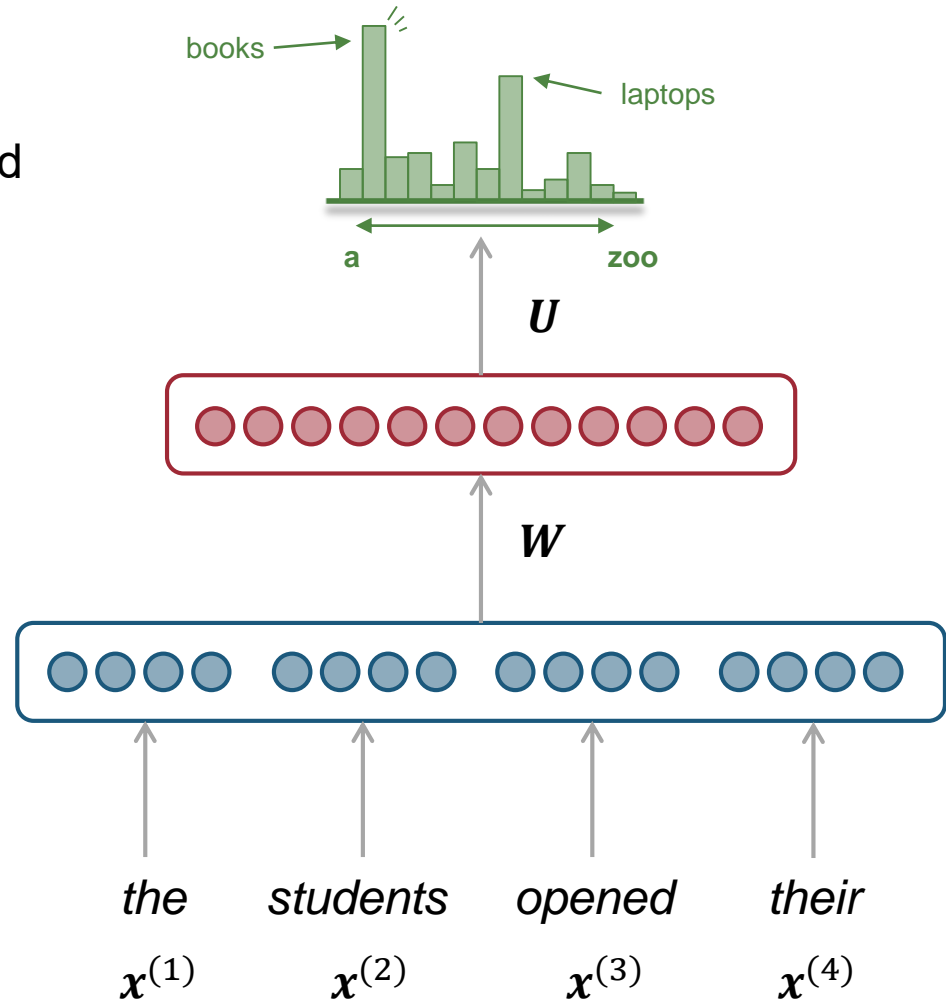
# A fixed window neural language model

## Improvements over n-gram LM:

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

## Remaining problems:

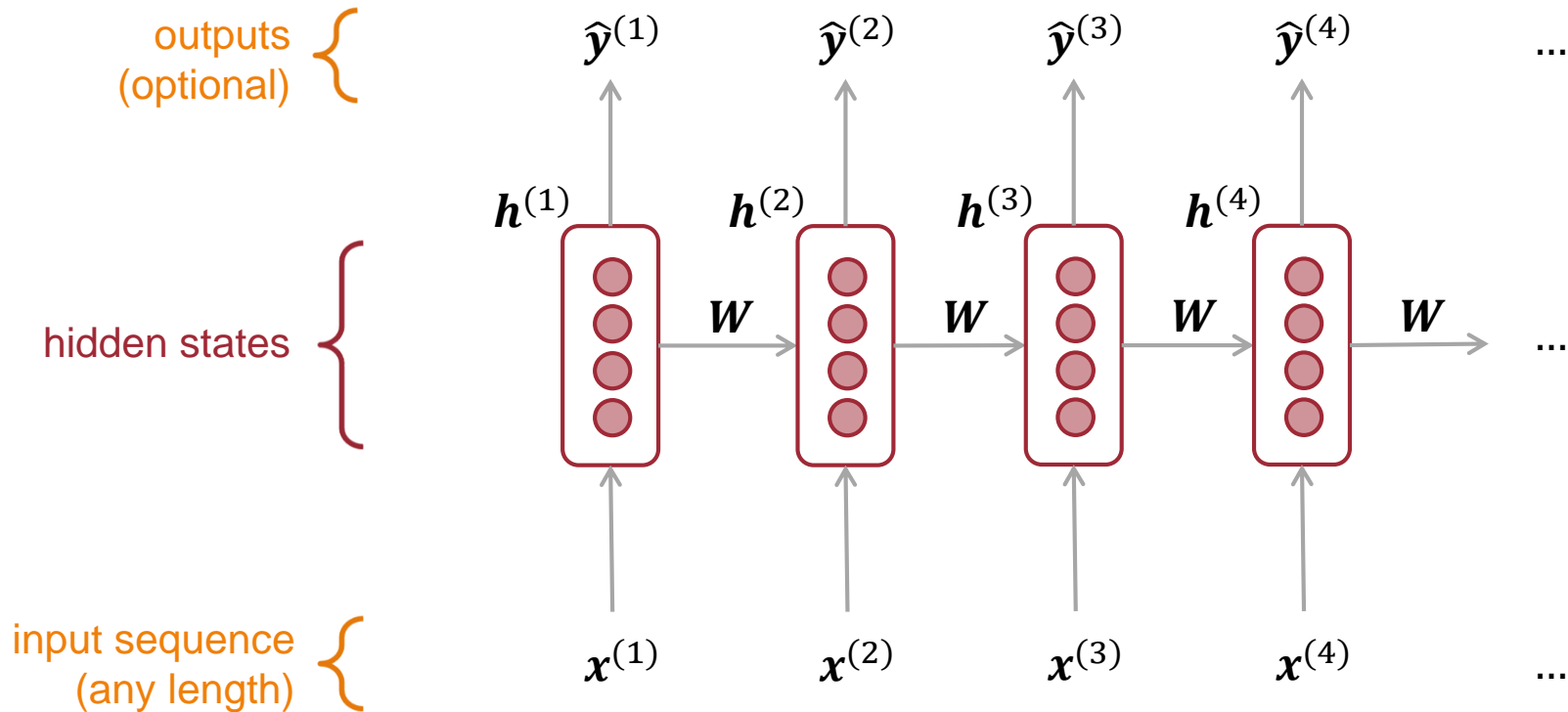
- Fixed window is too small
- Enlarging window enlarges  $W$



# Recurrent Neural Networks (RNN)

---

- ▶ A family of neural architectures

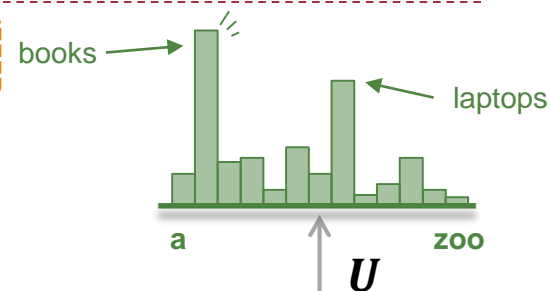


# RNN LM

output distribution

$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$

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



hidden layer

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

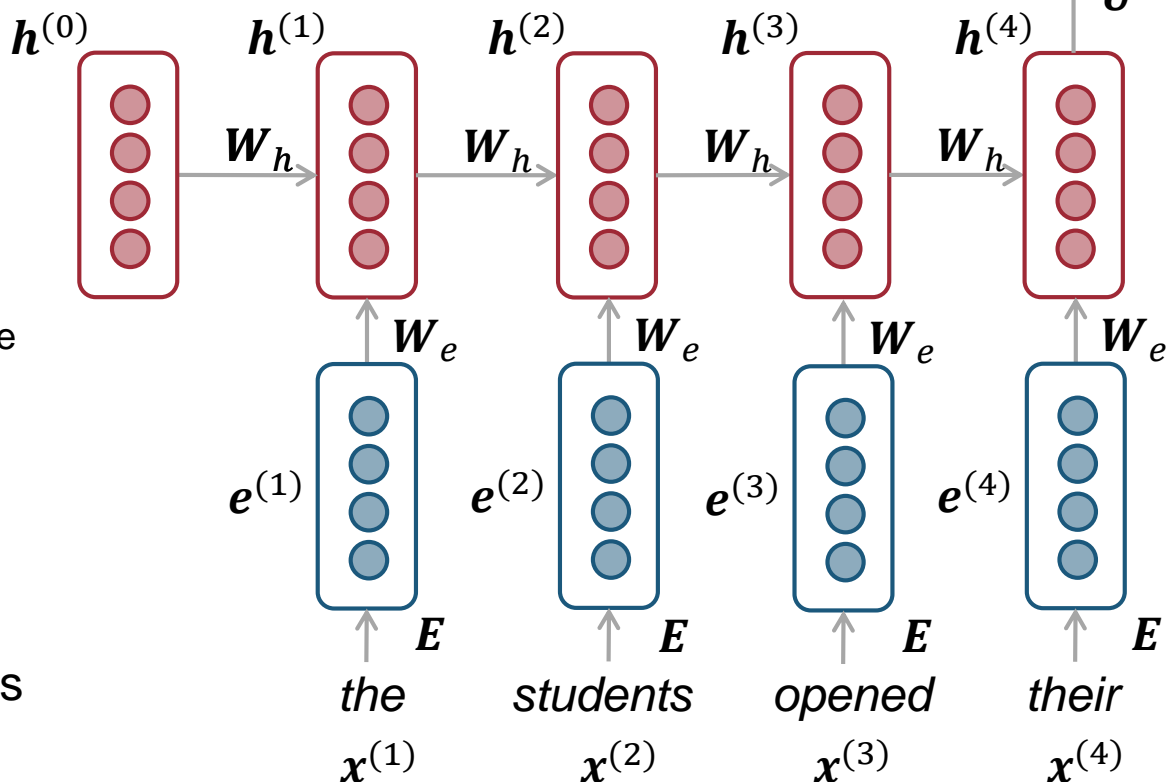
$h^{(0)}$  is the initial hidden state

word embeddings

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

words / one-hot vectors

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



# RNN LM

---

- ▶ Advantages

- ▶ No longer an n-gram (Markov) model
  - ▶ condition each prediction on **the whole history**
- ▶ Moderate model size
  - ▶ smaller than a typical n-gram model



# Generating with an RNN LM

---

- ▶ 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.*



# Training RNN LM

---

- ▶ Maximum likelihood estimation

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

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

- ▶ The overall loss is the average cross-entropy loss for every step:

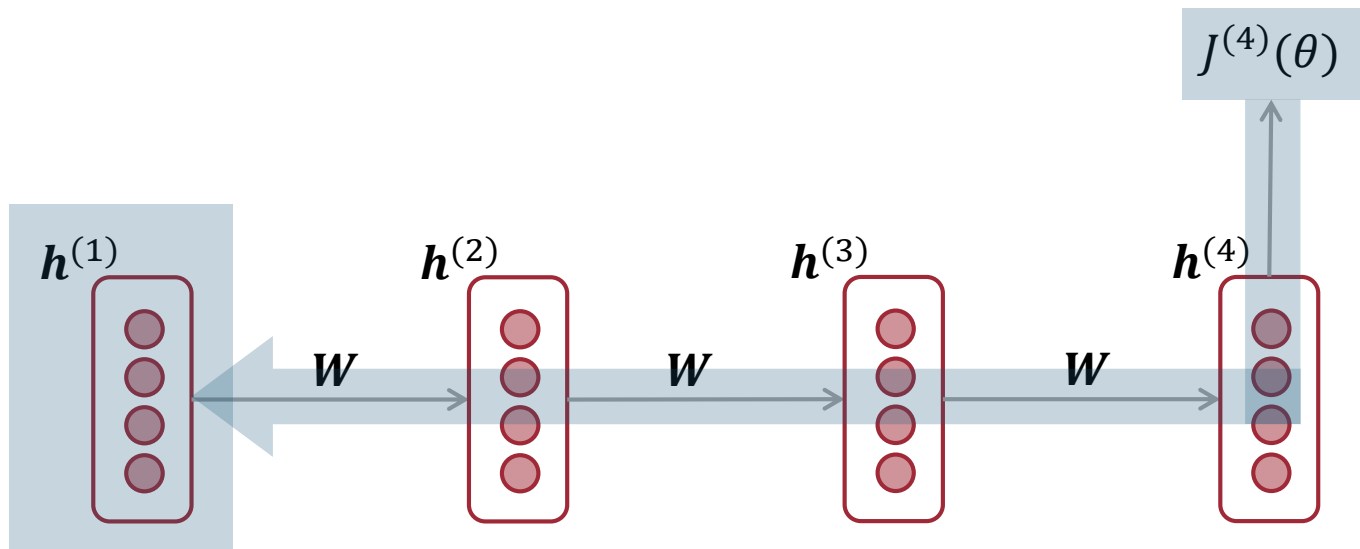
$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^T -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

- ▶ Optimized with stochastic gradient descent



# Vanishing gradient

---

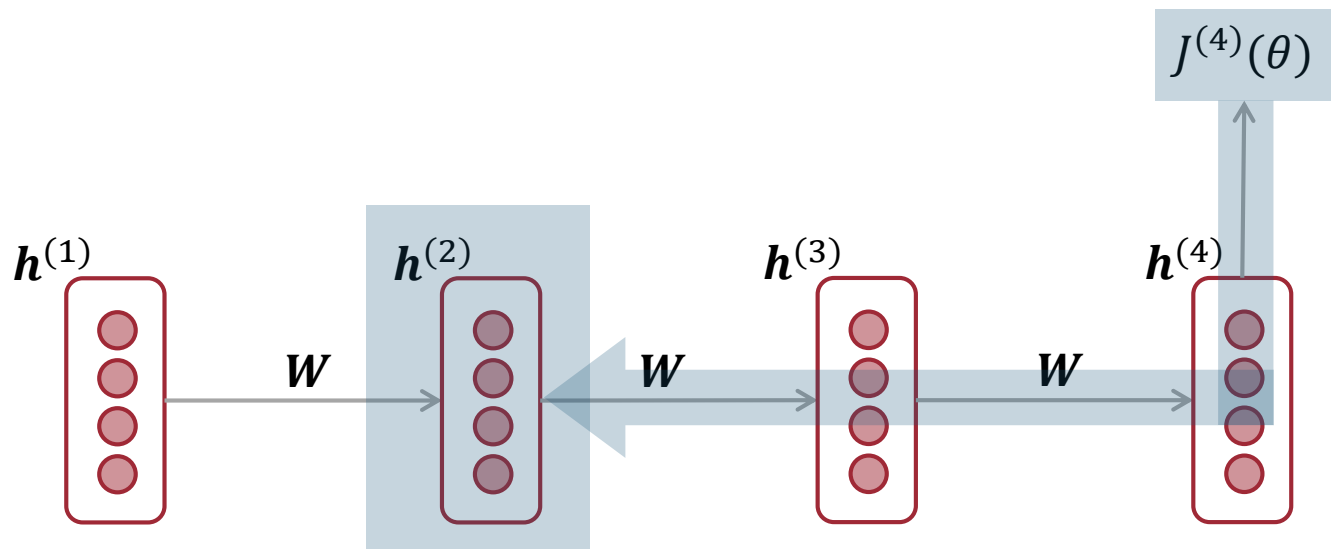


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = ?$$



# Vanishing gradient

---



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

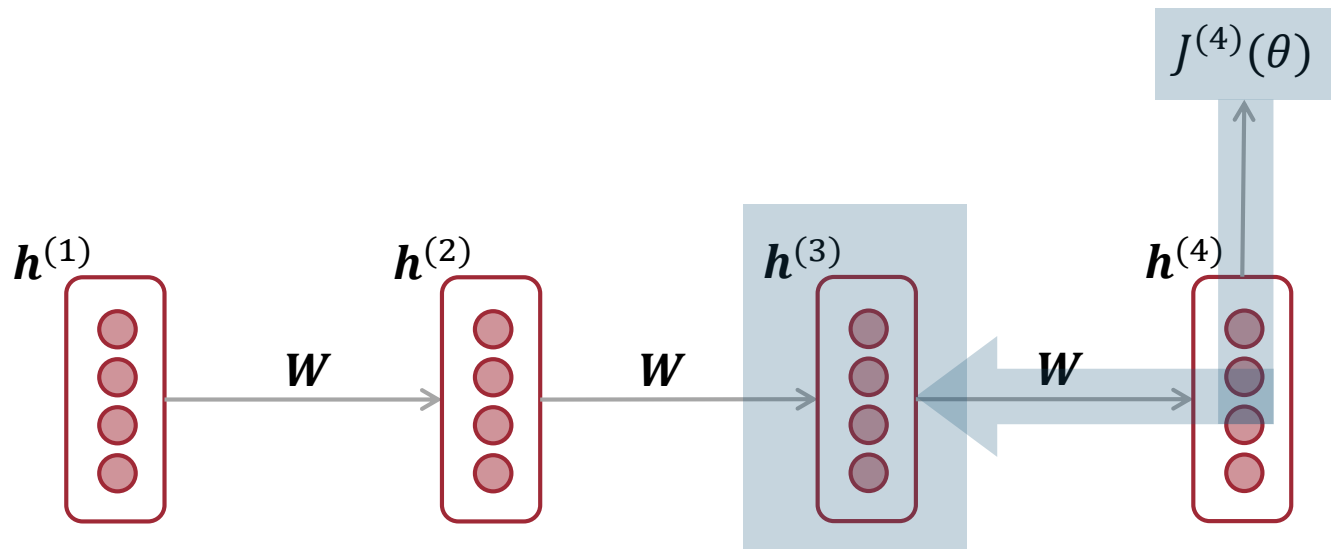
chain rule!





# Vanishing gradient

---



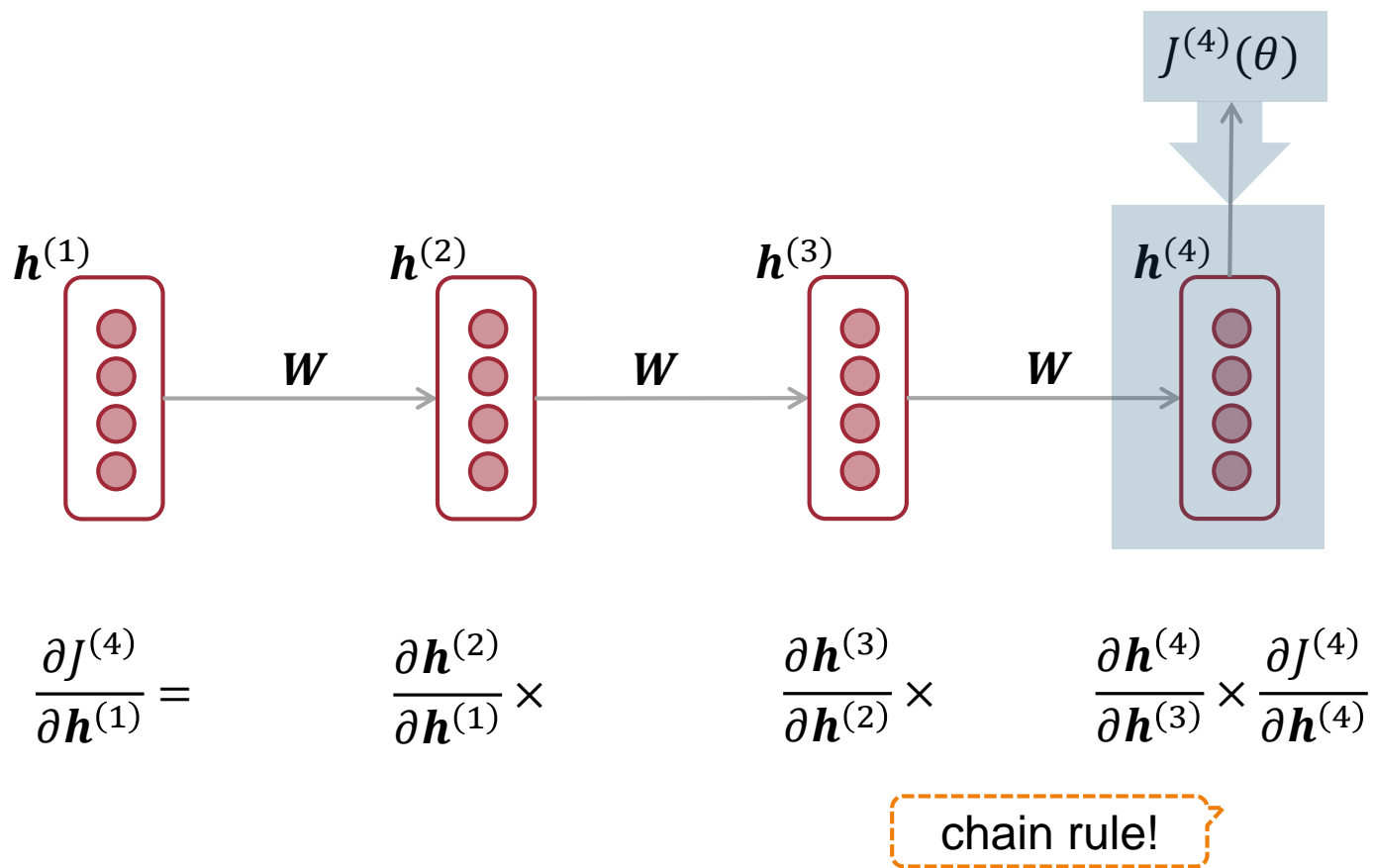
$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(3)}}$$

chain rule!

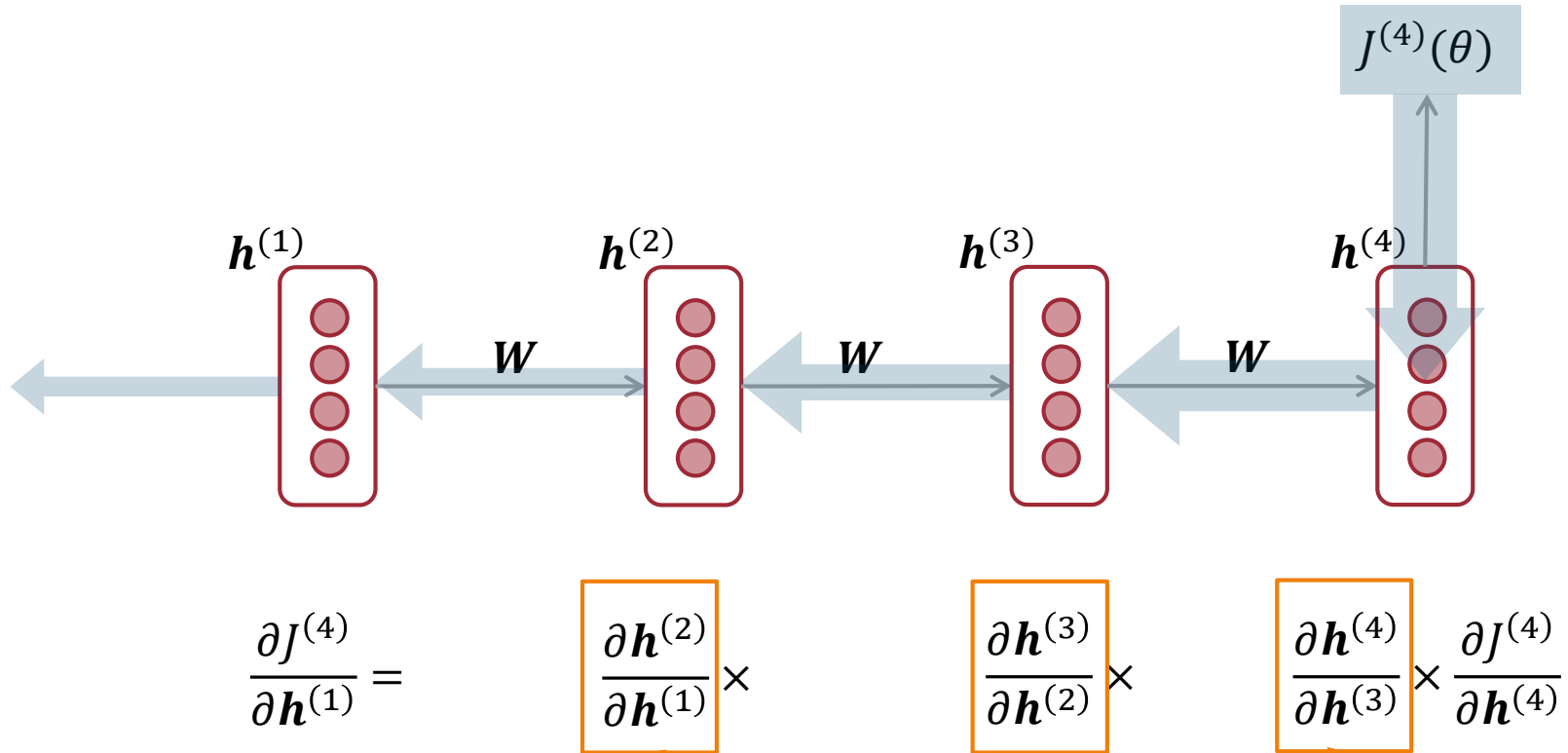


# Vanishing gradient

---



# Vanishing gradient

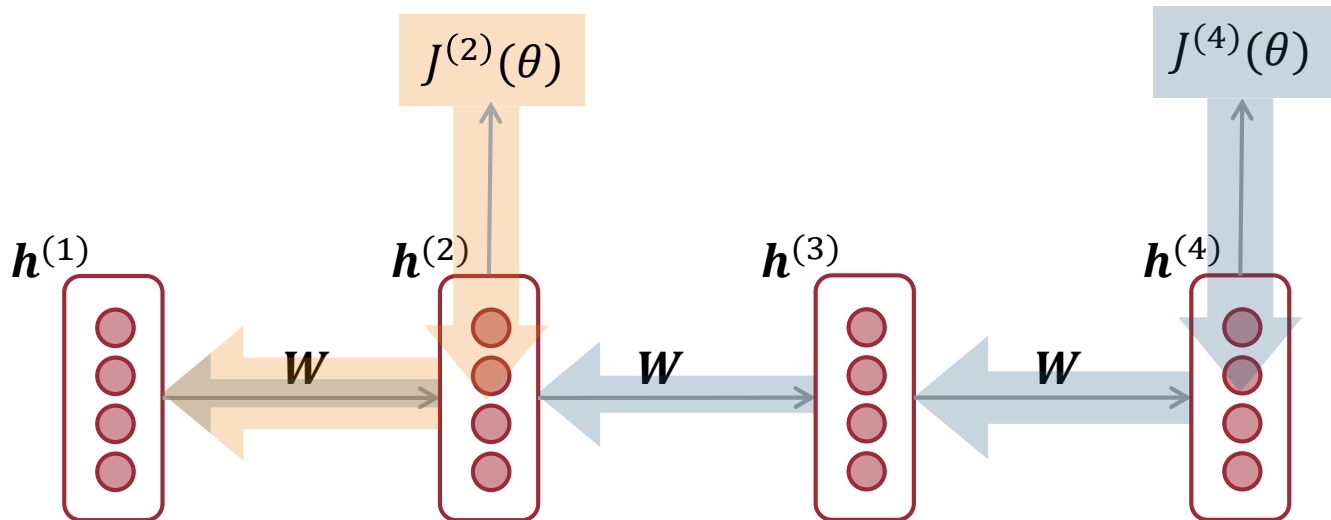


## **Vanishing gradient problem:**

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

# Vanishing gradient



Gradient signal from faraway 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.



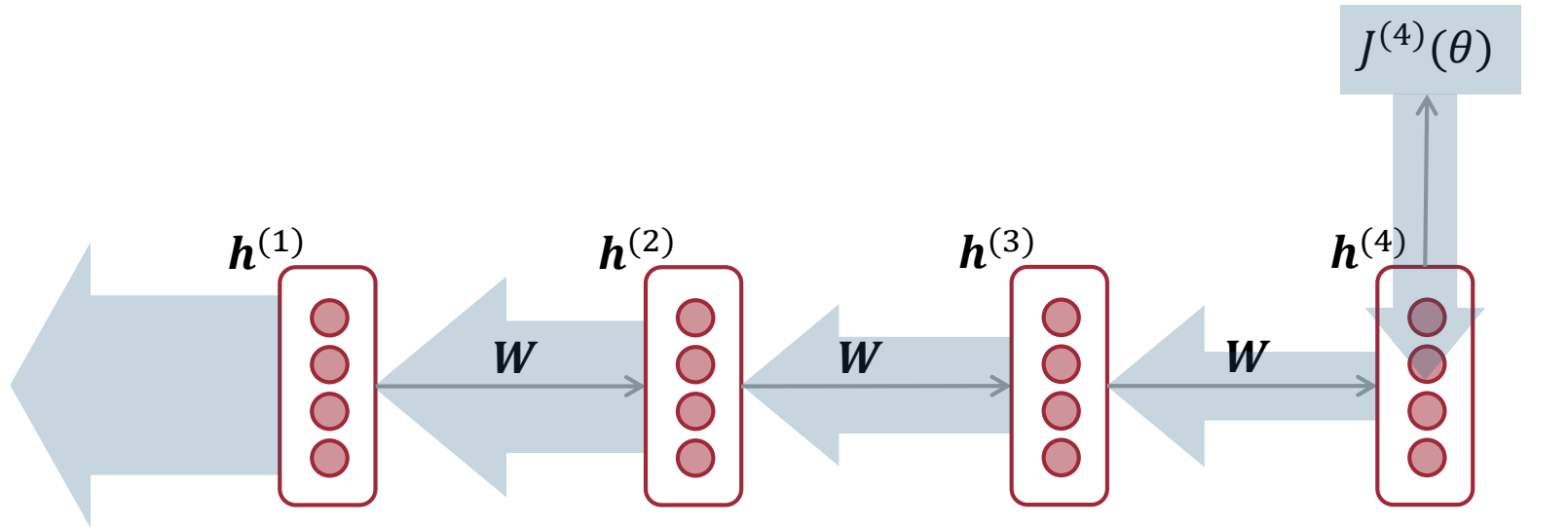
Why is vanishing gradient a problem?

# Vanishing gradient

is? are?

- ▶ **LM task:** The writer of the books \_\_\_\_
- ▶ **Correct answer:** The writer of the books is planning a sequel
- ▶ **Syntactic recency:** The writer of the books is ✓  

- ▶ **Sequential recency:** The writer of the books are ✗  

- ▶ Due to vanishing gradient, RNN LMs are better at learning from **sequential recency** than **syntactic recency**, so they make this type of error more often than we'd like [Linzen et al 2016]

# Exploding gradient



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

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

$$\frac{\partial h^{(3)}}{\partial h^{(2)}} \times$$

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

## **Exploding gradient problem:**

When these are large, the gradient signal gets larger and larger as it backpropagates further

# Exploding gradient

---

- ▶ Why is exploding gradient a problem?
  - ▶ **Bad updates**: we take too large a step and may reach a bad parameter configuration (with large loss)
  - ▶ In the worst case, this will result in **Inf** or **NaN** in your network (then you have to restart training)
- ▶ Gradient clipping: solution for exploding gradient
  - ▶ If the norm of the gradient is greater than some threshold, scale it down before applying SGD update
  - ▶ Intuition : take a step in the same direction, but a smaller step



# Vanishing gradient

---

- ▶ Another view of the problem:
  - ▶ In a vanilla RNN, the hidden state is constantly being rewritten

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

- ▶ Too difficult for RNN to **learn** to preserve information over many timesteps





# Long Short Term Memory (LSTM)

---

- ▶ A type of RNN [Hochreiter and Schmidhuber, 1997] as a solution to the **vanishing gradients problem**.
- ▶ On step  $t$ , there is a hidden state  $\mathbf{h}^{(t)}$  and a cell state  $\mathbf{c}^{(t)}$ 
  - ▶ 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 step, each element of the gates can be open (1), closed (0), or somewhere in between.
  - ▶ The gates are dynamic: their values are re-computed at each position



# Long Short Term Memory (LSTM)

- ▶ We have a sequence of inputs  $\mathbf{x}^{(t)}$ , and we will compute a sequence of hidden states  $\mathbf{h}^{(t)}$  and cell states  $\mathbf{c}^{(t)}$ .

- ▶ 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

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

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

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

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

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

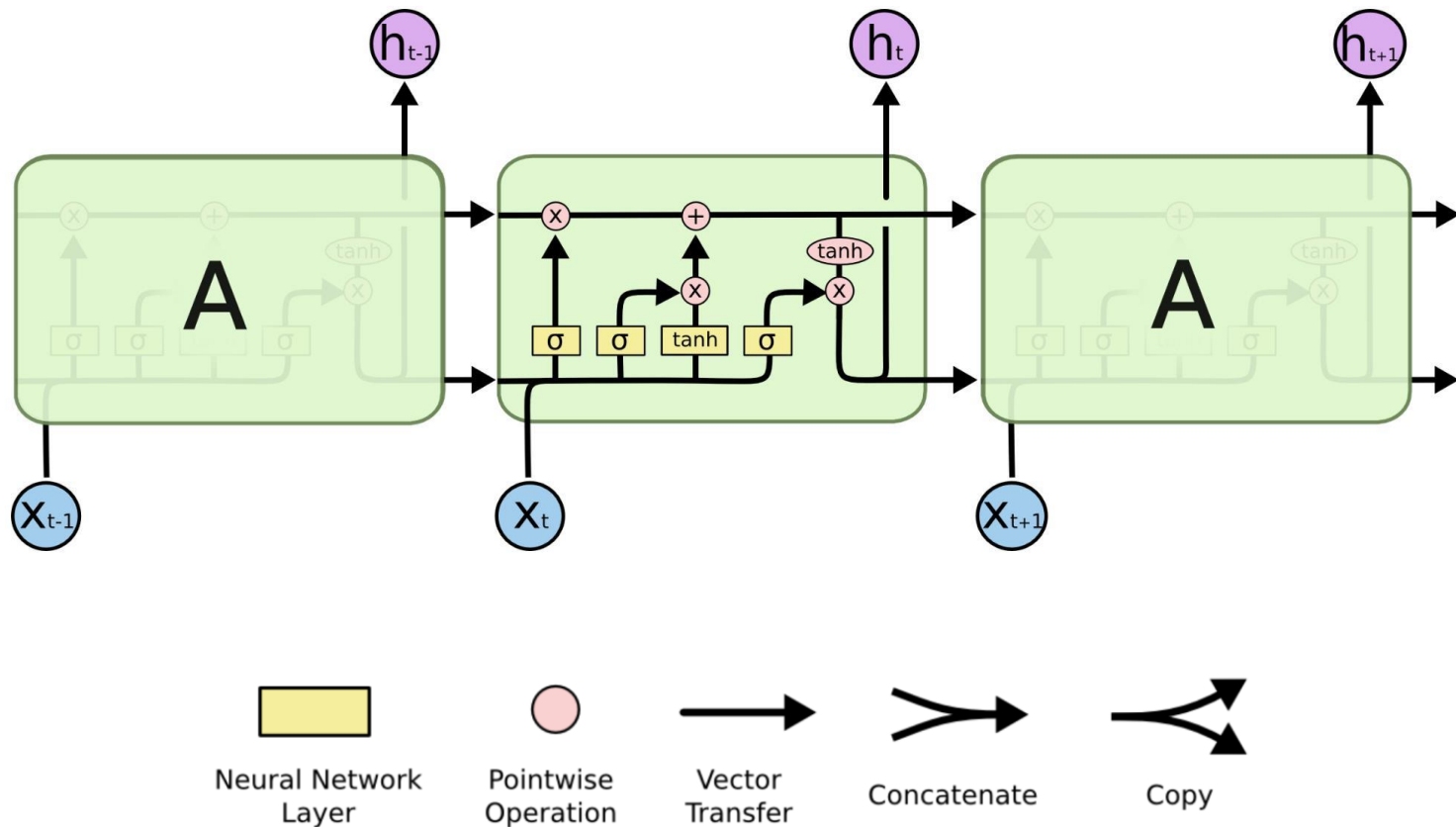
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh(\mathbf{c}^{(t)})$$

Gates are applied using element-wise product

All these are vectors of same length  $n$

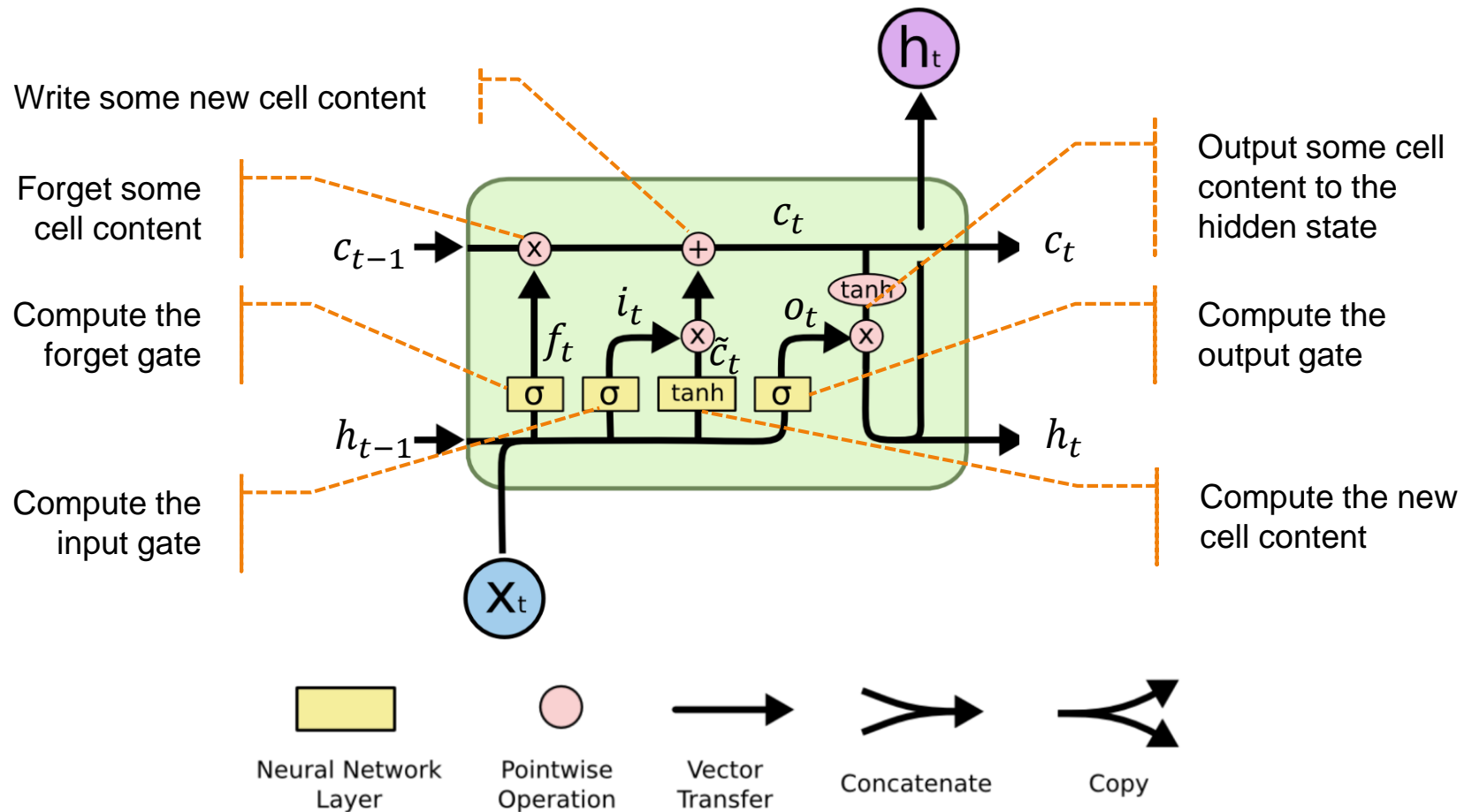
# Long Short Term Memory (LSTM)

- ▶ You can think of the LSTM equations visually like this:



# Long Short Term Memory (LSTM)

- ▶ You can think of the LSTM equations visually like this:



# Long Short Term Memory (LSTM)

---

- ▶ How does LSTM solve vanishing gradients?
- ▶ The LSTM architecture makes it easier to preserve information over many timesteps
  - ▶ Ex. if the forget gate is open and input gate is closed on every timestep, then the info in the cell is preserved indefinitely
  - ▶ By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix  $W_h$  that preserves info in hidden state
- ▶ LSTM doesn't *guarantee* that there is no vanishing or exploding gradient, but it does provide an easier way to learn long distance dependencies



# Gated Recurrent Units (GRU)

---

- ▶ A simpler alternative to the LSTM [Cho et al., 2014]
- ▶ On timestep  $t$ , we have input  $\mathbf{x}^{(t)}$  and hidden state  $\mathbf{h}^{(t)}$  (no cell state).

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

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

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

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

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

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

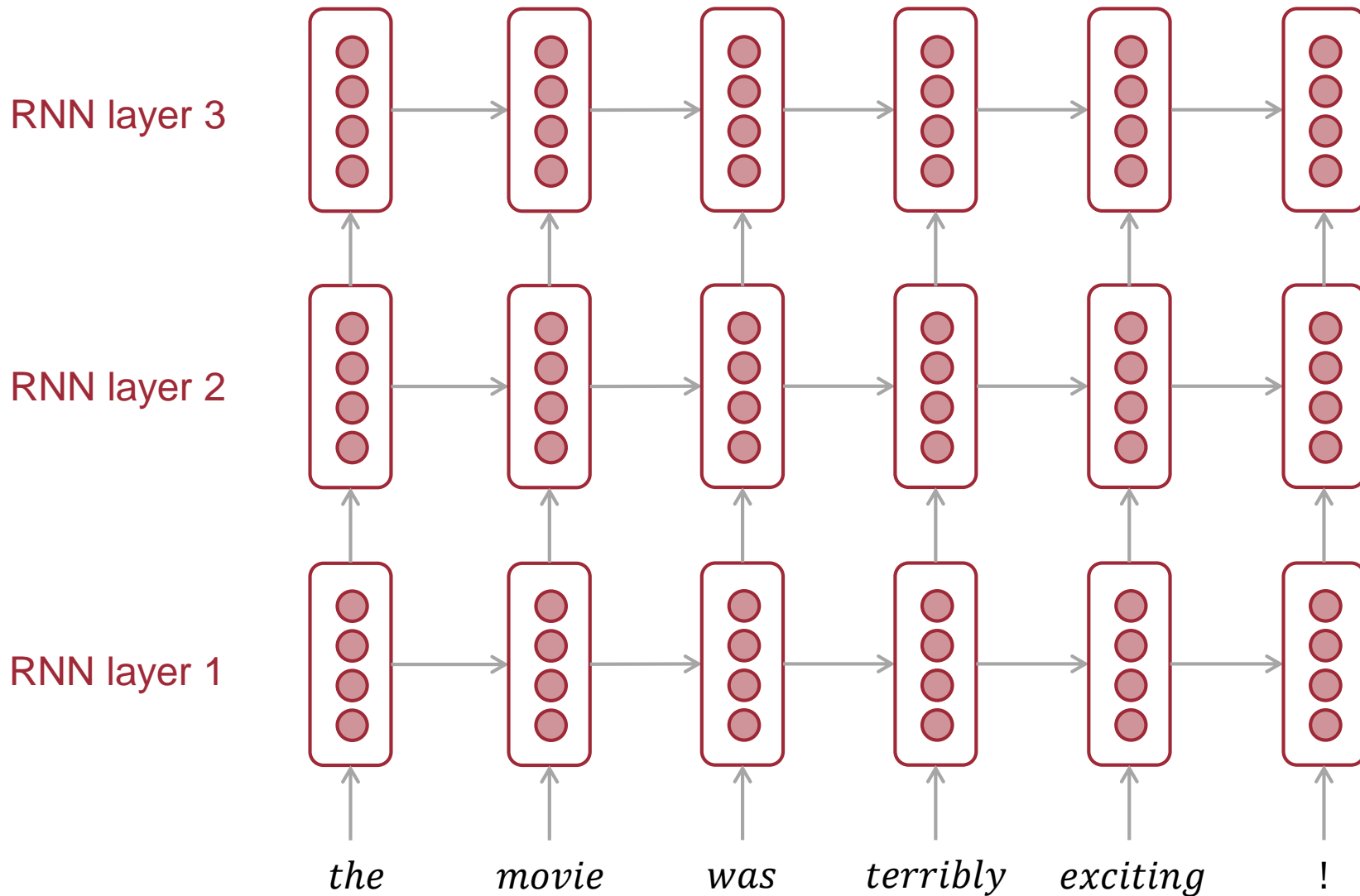
**Hidden state:** update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

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

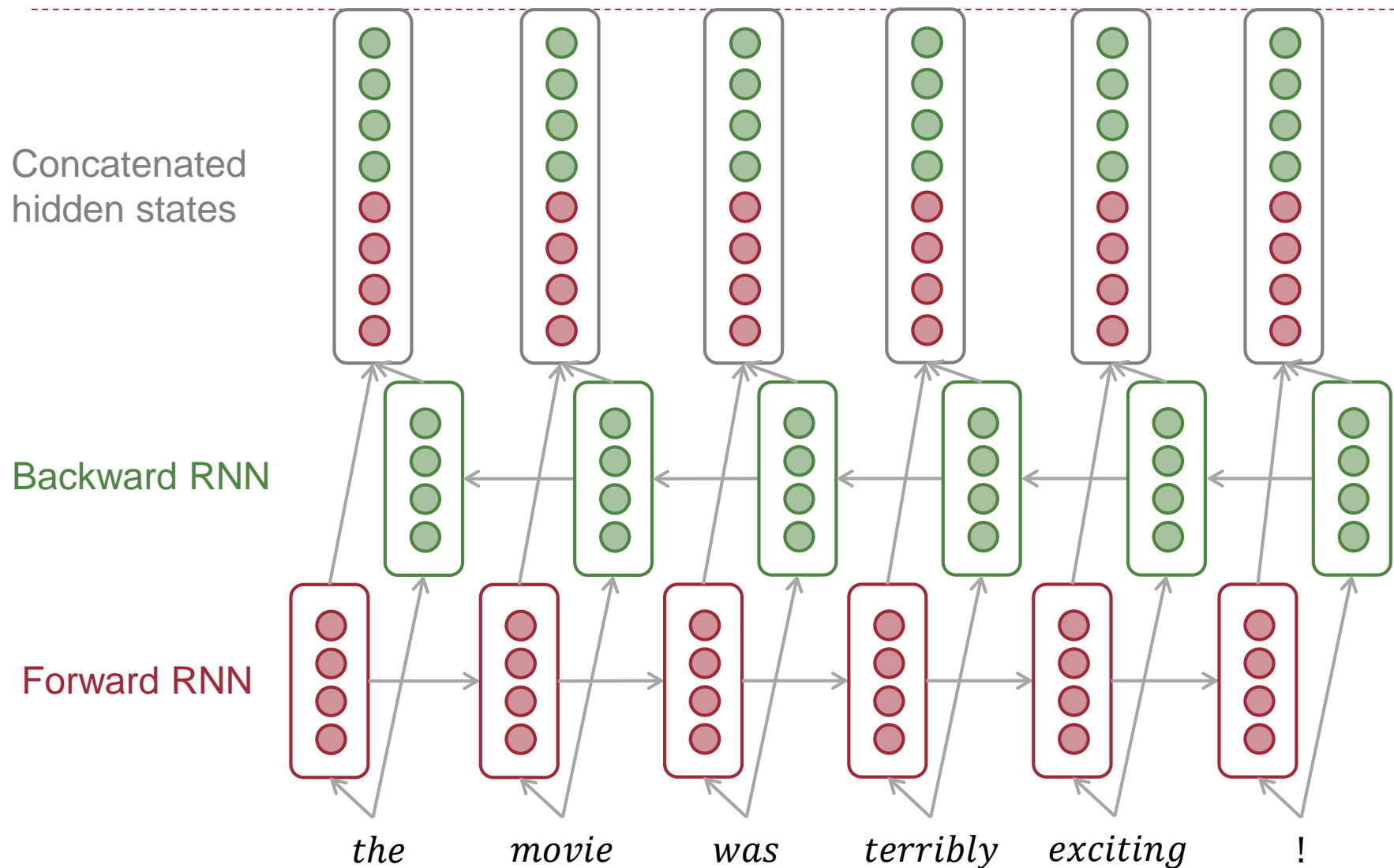


# Multi-layer RNN

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



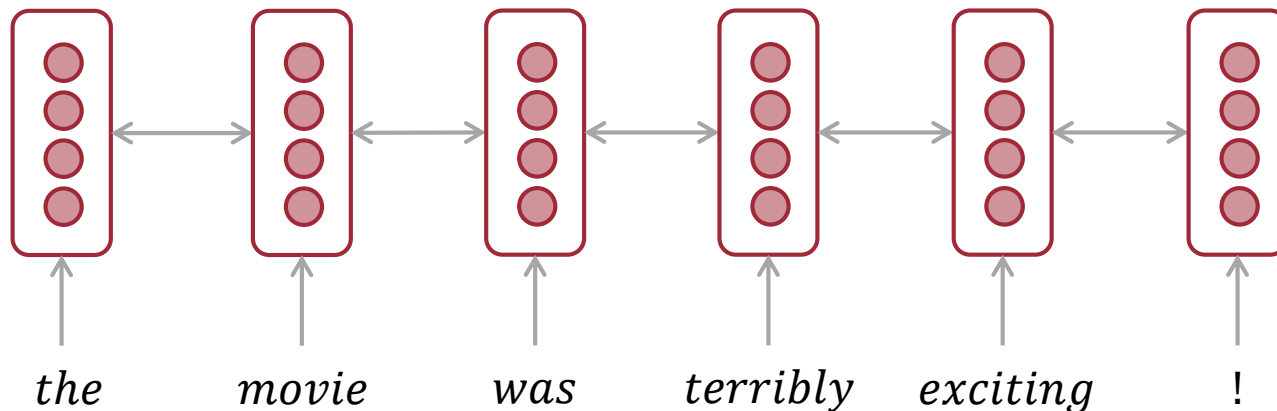
# Bidirectional RNN





# Bidirectional RNN: simplified diagram

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



Can Bidirectional RNNs be used for LM?

No! The bidirectional representation at each position contains information from the whole sentence. In LM, we need to predict the next word conditioned on input to the left.

# RNN Beyond LM

---

- ▶ RNN/LSTM can be used for many other tasks
  - ▶ Word encoding
  - ▶ Sequence labeling
  - ▶ Sentence encoding
    - ▶ Use the last hidden vector as a sentence representation
  - ▶ Sentence classification
  - ▶ Text generation
  - ▶ (Some of these will be discussed later)





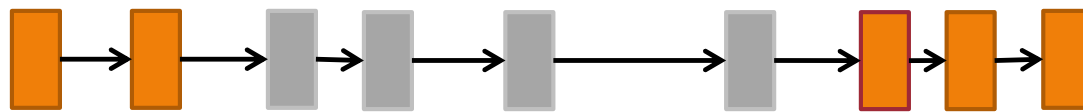
Attention



# RNN: linear interaction distance

---

- ▶ Sequential recency / linear locality in RNN
  - ▶ Nearby words affect each other's representations more than distant words
  - ▶ RNNs take  $O(\text{sequence length})$  steps for distant word pairs to interact.



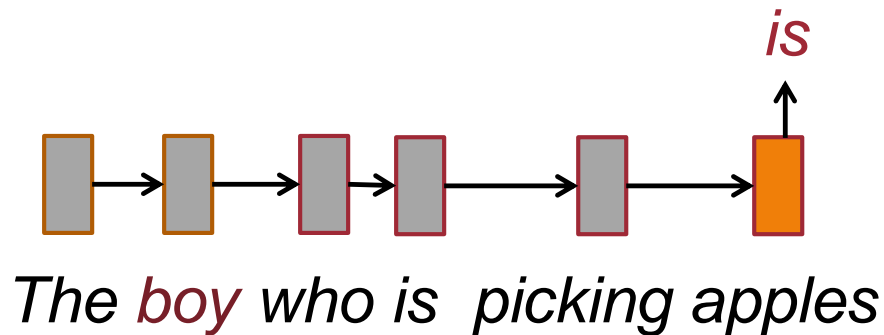
*The **boy** who is picking apples is his son*



# RNN: the bottleneck problem in LM

---

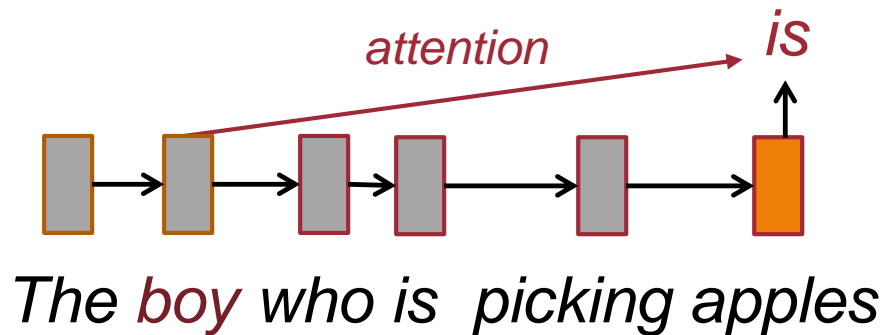
- ▶ The “last hidden state” of RNN needs to capture all information about the previous words to predict next word



# Attention

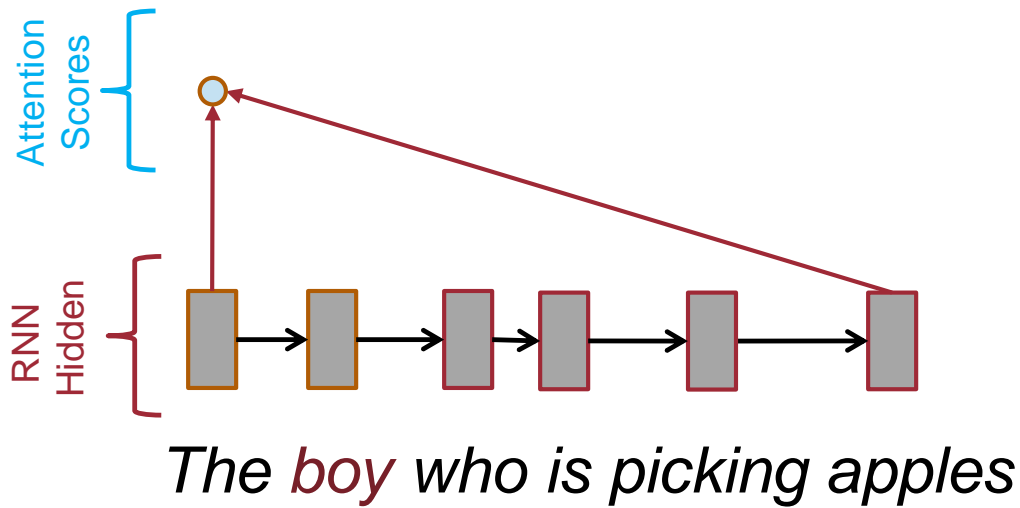
---

- ▶ At each step, add direct connections to a particular part of the history
  - ▶ Ex. attending “**boy**” when generating “**is**”
- ▶ But how do we know which part to attend to?
  - ▶ Attend to everything in the history
  - ▶ But **learn to predict** a different attention score for each word



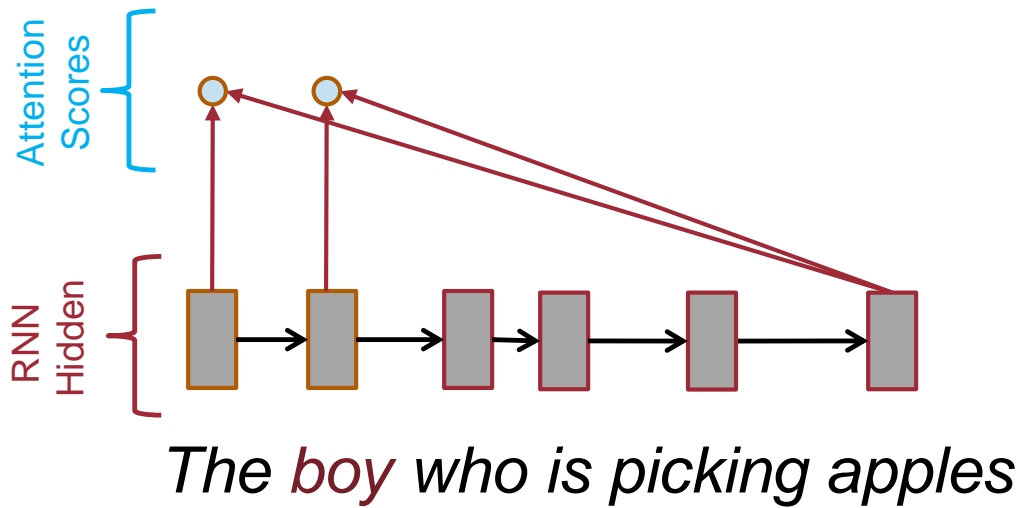
# Attention

---



# Attention

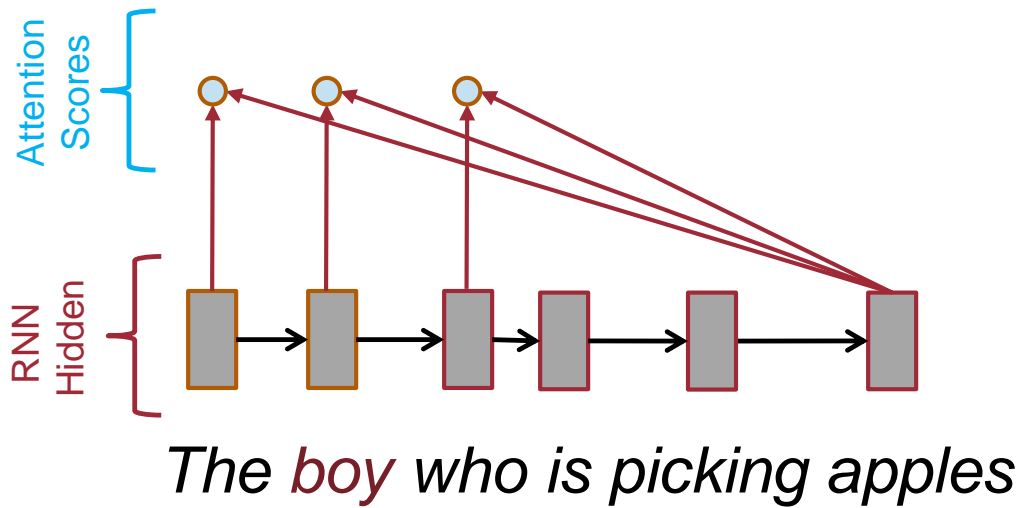
---





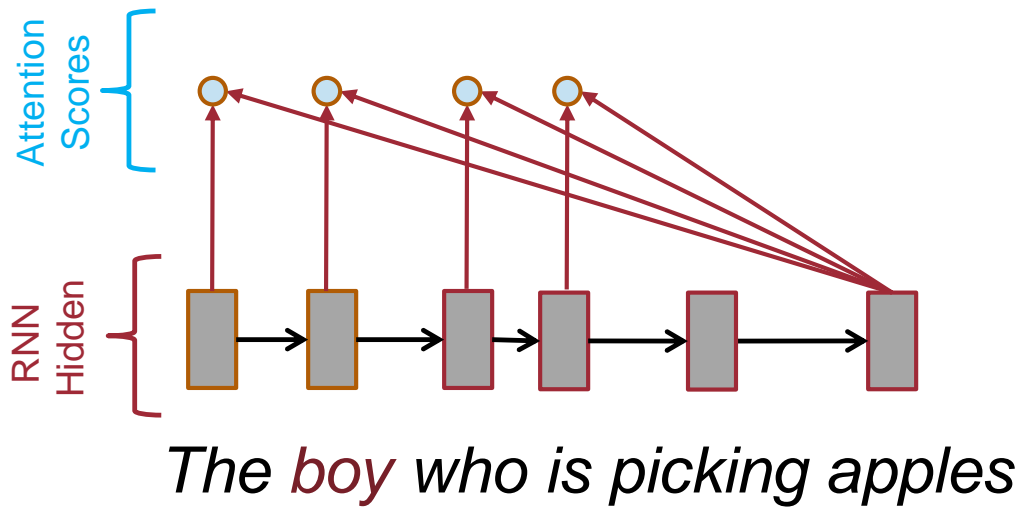
# Attention

---



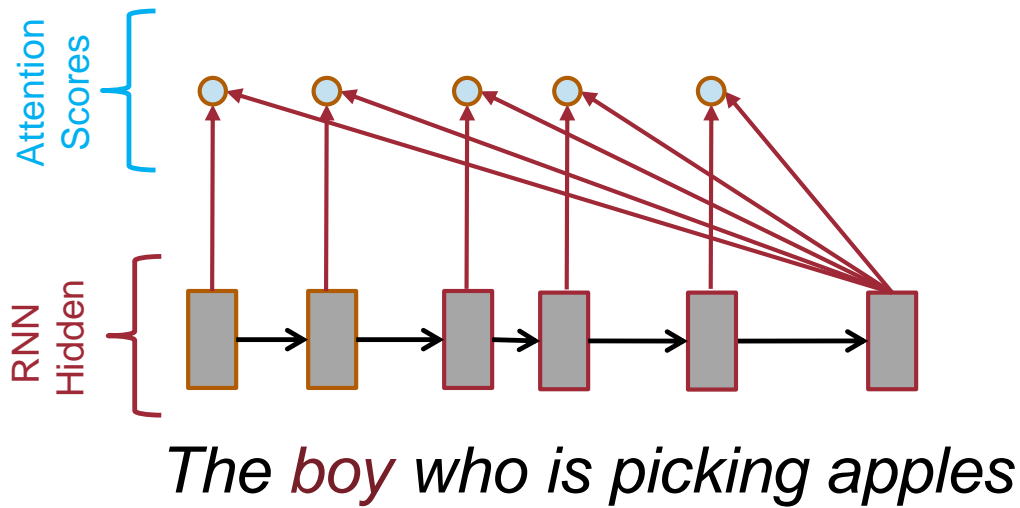
# Attention

---



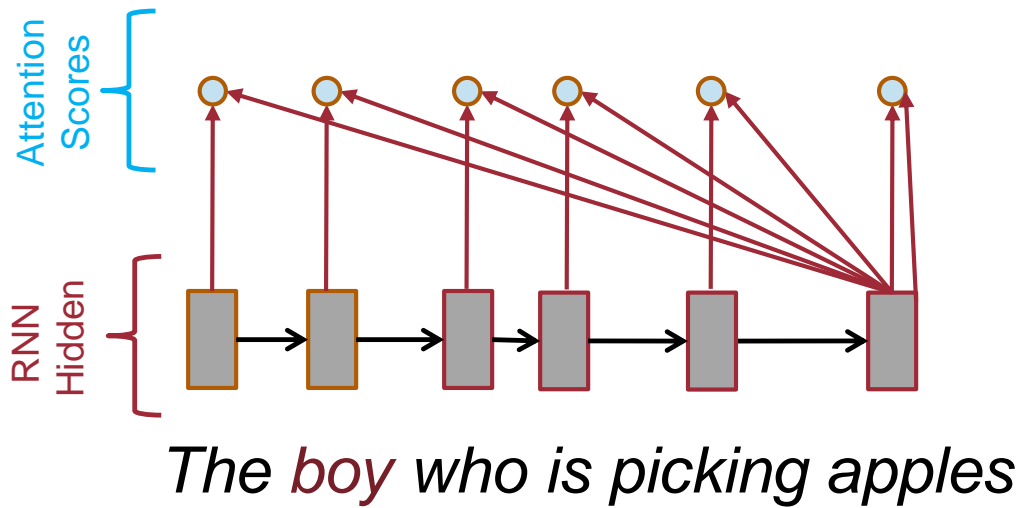
# Attention

---



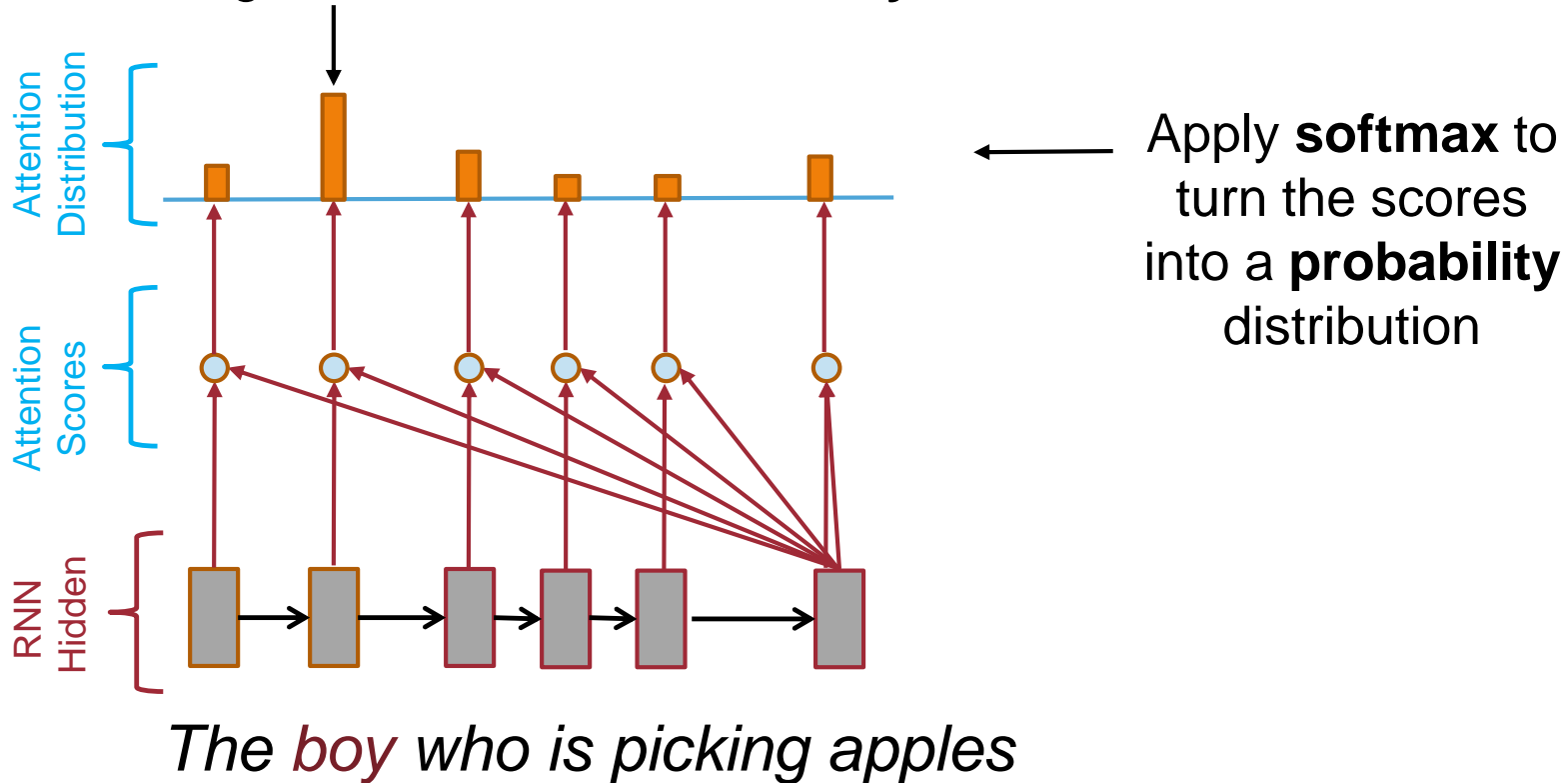
# Attention

---

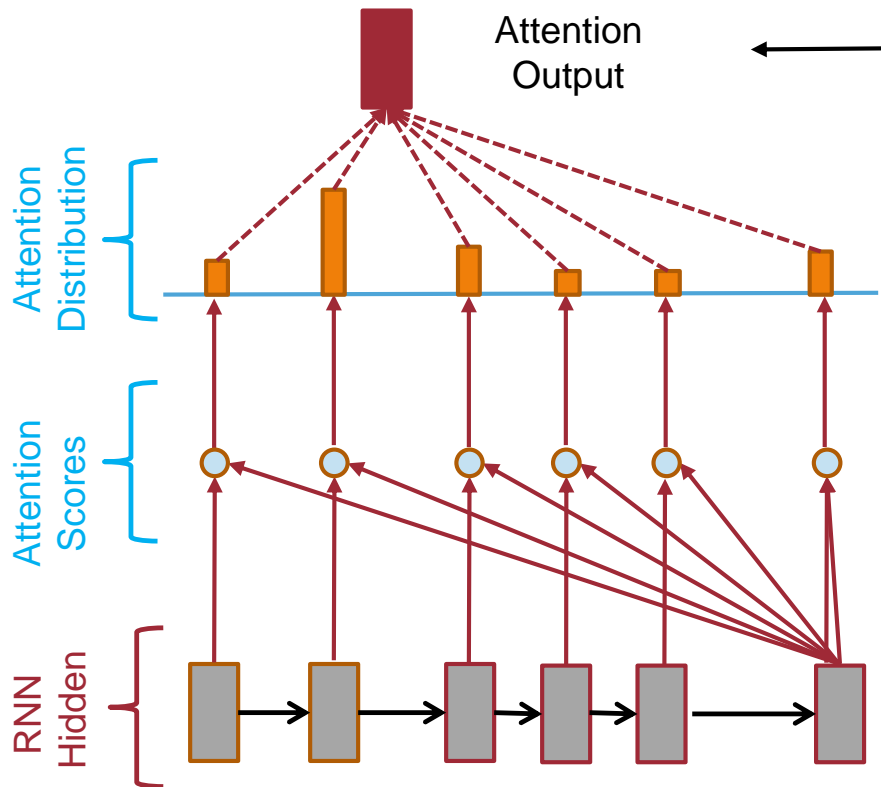


# Attention

When generating “is”, we’re mostly focusing on the hidden state of “**boy**”



# Attention



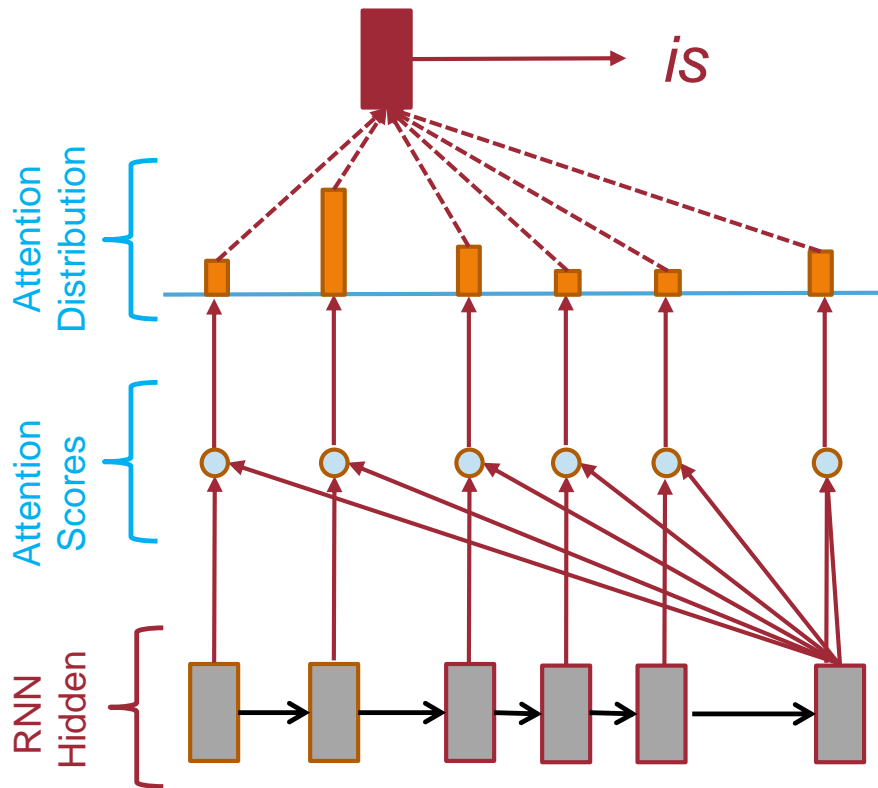
Use the **attention distribution** to take a **weighted sum** of **all seen hidden states**.

The **attention output** mostly contains information from the hidden states that received **high attention**.

*The **boy** who is picking apples*



# Attention



Use the **attention output** to predict the next word

*The **boy** who is picking apples*



# Dot-product Attention

---

- ▶ Inputs: a **query**  $q$  and a set of **key-value** (k-v) pairs to an output
  - ▶ Query, keys, values and output are all vectors
  - ▶ Queries and keys have same dimension  $d_k$
  - ▶ Values have dimension  $d_v$
  - ▶ In previous case, keys/values/queries are all computed from the hidden states of the same sentence (**self-attention**)
- ▶ Output: weighted sum of values
  - ▶ weight of each value is computed by an inner product of query and corresponding key

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$





# Dot-product Attention – Matrix notation

---

- ▶ When we have multiple queries  $q$ , we stack them in a matrix  $Q$ :

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$



$$A(Q, K, V) = \text{Softmax}(QK^T)V$$

$$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$$

Softmax row-wise  $\left( \begin{array}{|c|} \hline \equiv \\ \hline \end{array} \quad \begin{array}{|c|} \hline |||| \\ \hline \end{array} \right) \begin{array}{|c|} \hline \equiv \\ \hline \end{array} = [|Q| \times d_v]$



# Scaled Dot-Product Attention

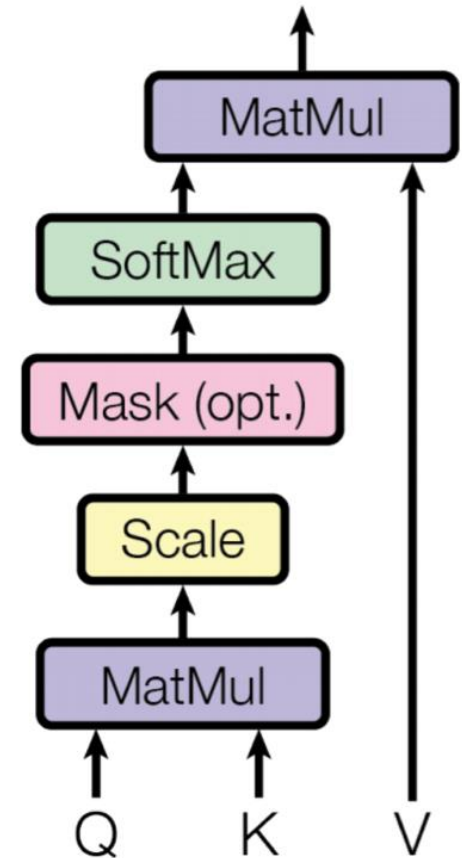
## Problem:

- ▶ As  $d_k$  gets large, the variance of  $q^T k$  increases
- ▶ → some very large (positive or negative) values
- ▶ → numerical problem when applying exp in softmax

## Solution

- ▶ Scale by length of query/key vectors

$$A(Q, K, V) = \text{Softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$



# Attention Variants

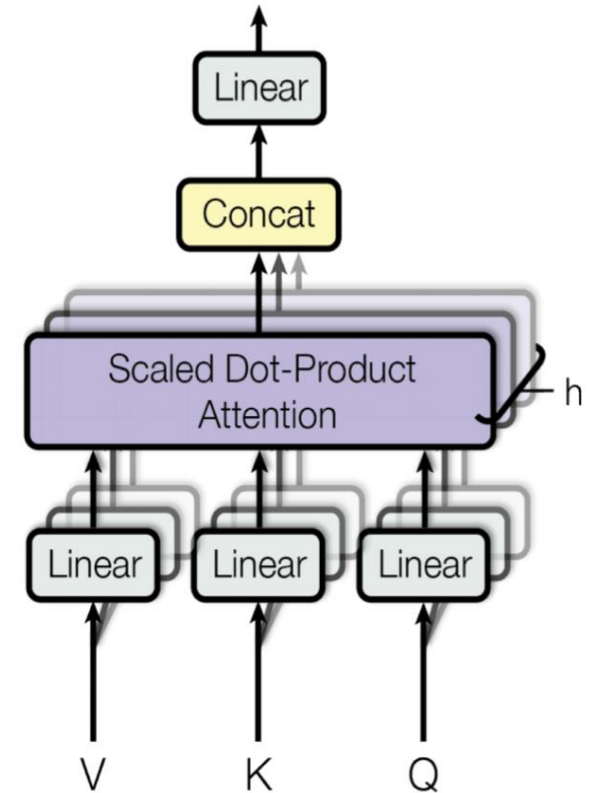
---

- ▶ There are several ways you can compute the attention score of a query vector  $q$  and a key vector  $k_i$ .
- ▶ Basic dot-product attention:  $a_i = q^T k_i \in \mathbb{R}$ 
  - ▶ Note: this assumes  $d_q = d_k$
  - ▶ This is the version we saw earlier
- ▶ Multiplicative attention:  $a_i = q^T W k_i \in \mathbb{R}$ 
  - ▶  $W \in \mathbb{R}^{d_q \times d_k}$  is a weight matrix
- ▶ Additive attention:  $a_i = u^T \tanh(W_1 k_i + W_2 q) \in \mathbb{R}$ 
  - ▶  $W_1 \in \mathbb{R}^{d_u \times d_k}$ ,  $W_2 \in \mathbb{R}^{d_u \times d_q}$  are weight matrices and  $u \in \mathbb{R}^{d_u}$  is a weight vector.
  - ▶  $d_u$  (the attention dimension) is a hyperparameter



# Multi-head self-attention

- ▶ Problem with simple self-attention:
  - ▶ Only one way for words to interact with one-another
- ▶ Solution: Multi-head attention
  - ▶ First map Q, K, V into multiple lower dimensional spaces
  - ▶ Apply attention in each space
  - ▶ Concatenate outputs and pipe through a linear layer



$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$

where  $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

# Range of attention

---

- ▶ RNNs look at the whole history
  - ▶ But with sequential recency / linear locality
- ▶ Attention looks at the immediate history of a fixed length
  - ▶ But without sequential recency / linear locality





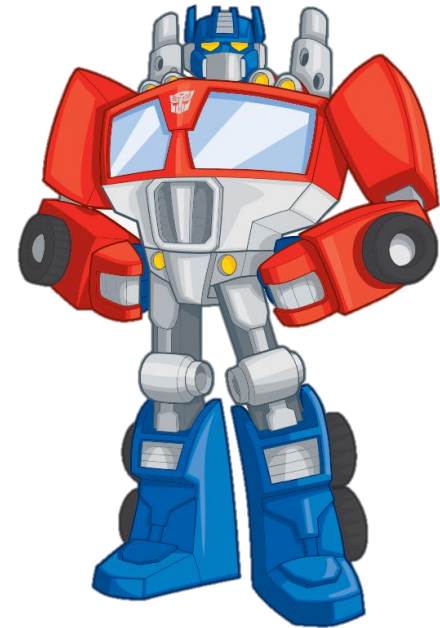
# Transformer



# Recurrence and Attention

---

- ▶ RNN relies on recurrence connections to access history
- ▶ Attention gives us access to any part of history
- ▶ Then why do we still need recurrence?
- ▶ Transformer
  - ▶ Attention is all you need!



# Recurrence and Attention

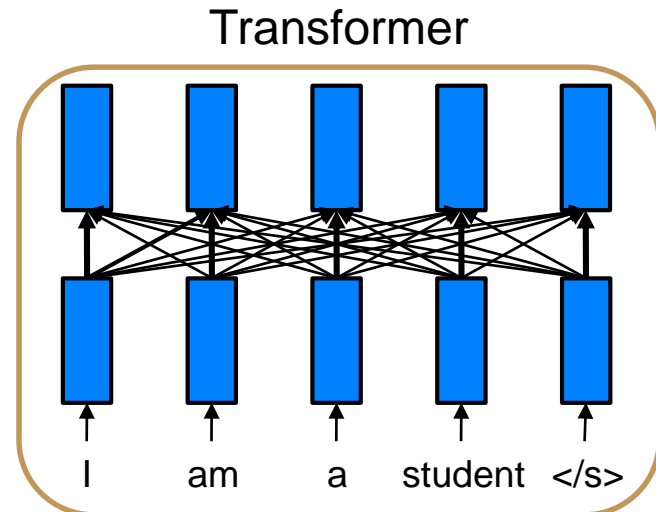
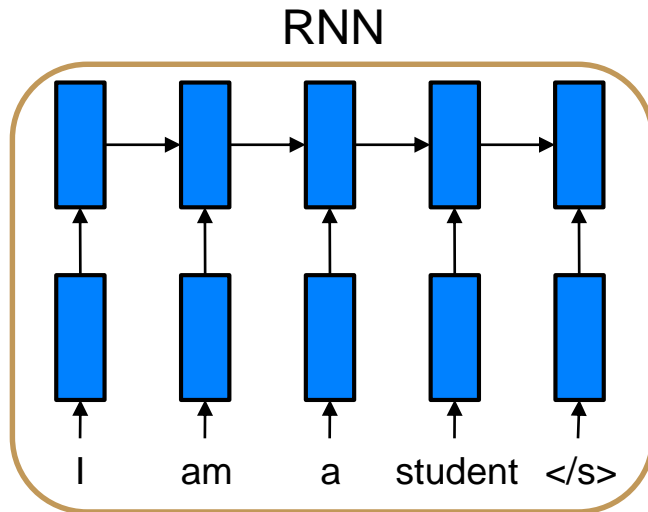
---

- ▶ Parallelizability

- ▶ RNN is not parallelizable: future hidden states depend on past hidden states
- ▶ Transformer can be parallelized over positions

- ▶ What info is missing?

- ▶ Word order!



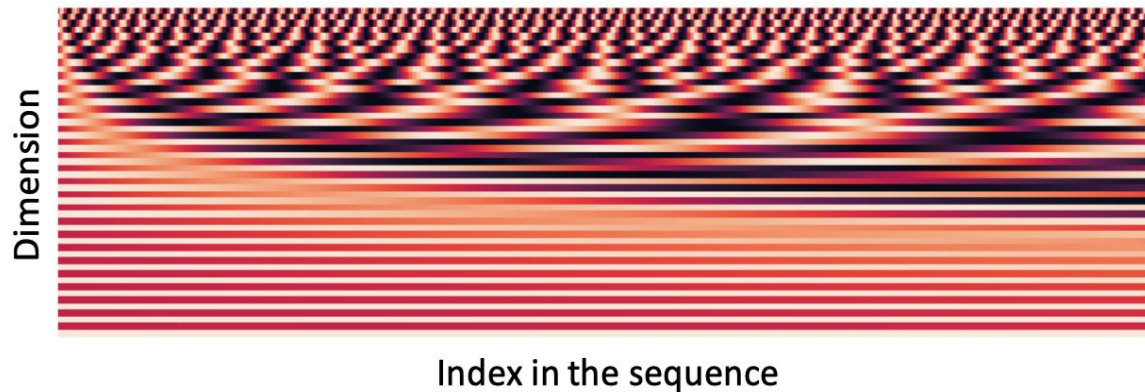


# Position Embedding

---

- ▶ Method 1: Concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



$$p_{i,2j} = \sin(pos/10000^{2j/d})$$

$$p_{i,2j+1} = \cos(pos/10000^{2j/d})$$

- ▶ Method 2: let all  $p_i$  be learnable parameters!
  - ▶ Most systems use this!

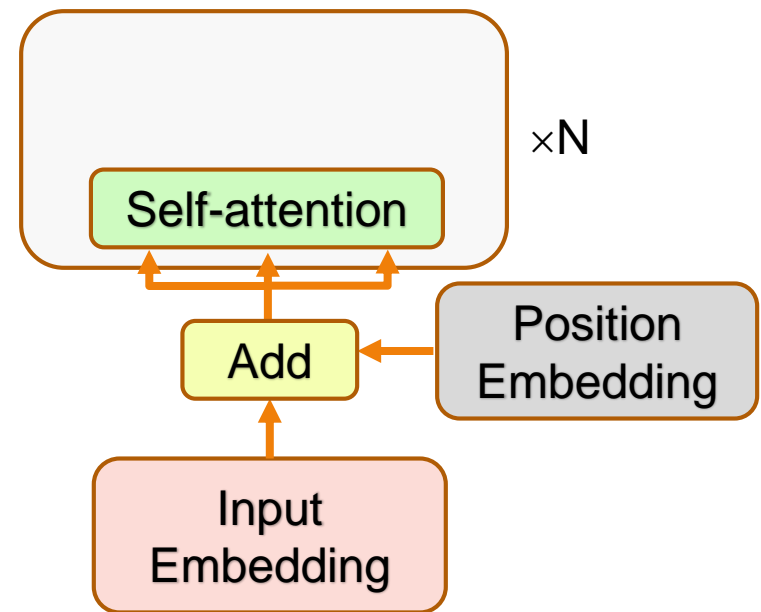


# Transformer

---

- ▶ Add position embedding to input embedding

We only show one self-attention head for simplicity

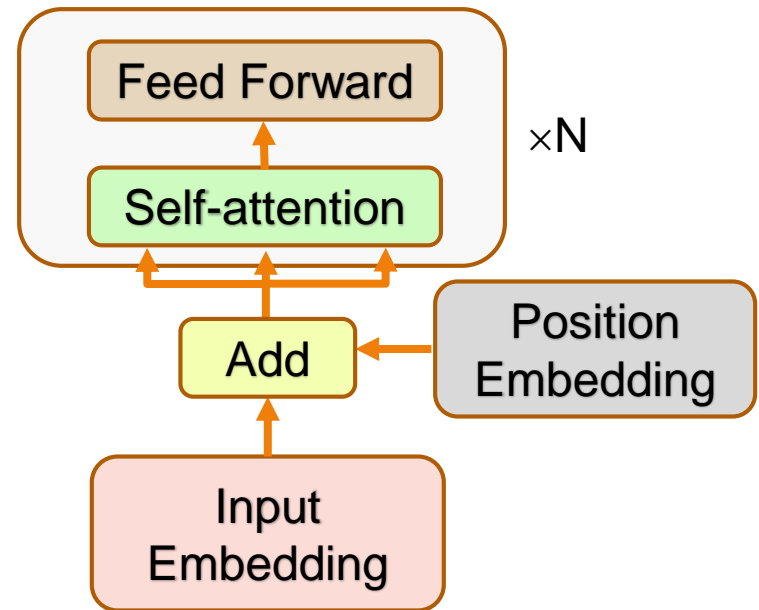
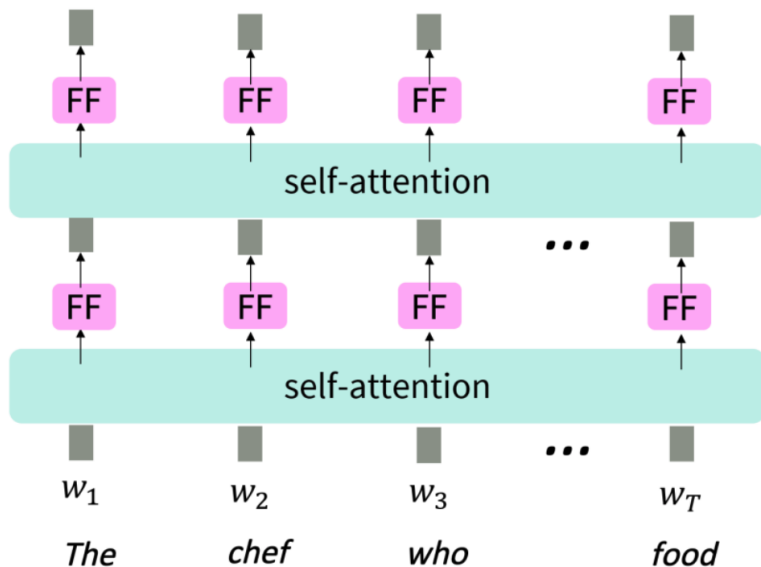


# Transformer - Introducing nonlinearities

- Problem: there is no non-linearities; self-attention is simply performing re-averaging of value vectors.
- Easy fix: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

## Equation for Feed Forward Layer

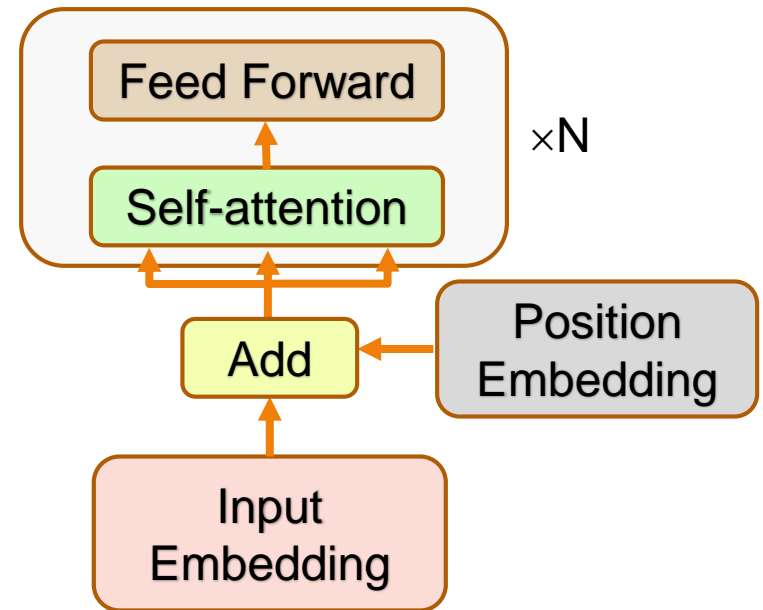
$$\begin{aligned} m_i &= \text{MLP}(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2 \end{aligned}$$



# Training Trick #1: Residual Connections

---

- ▶ Deep networks are surprisingly bad at learning the identity function!

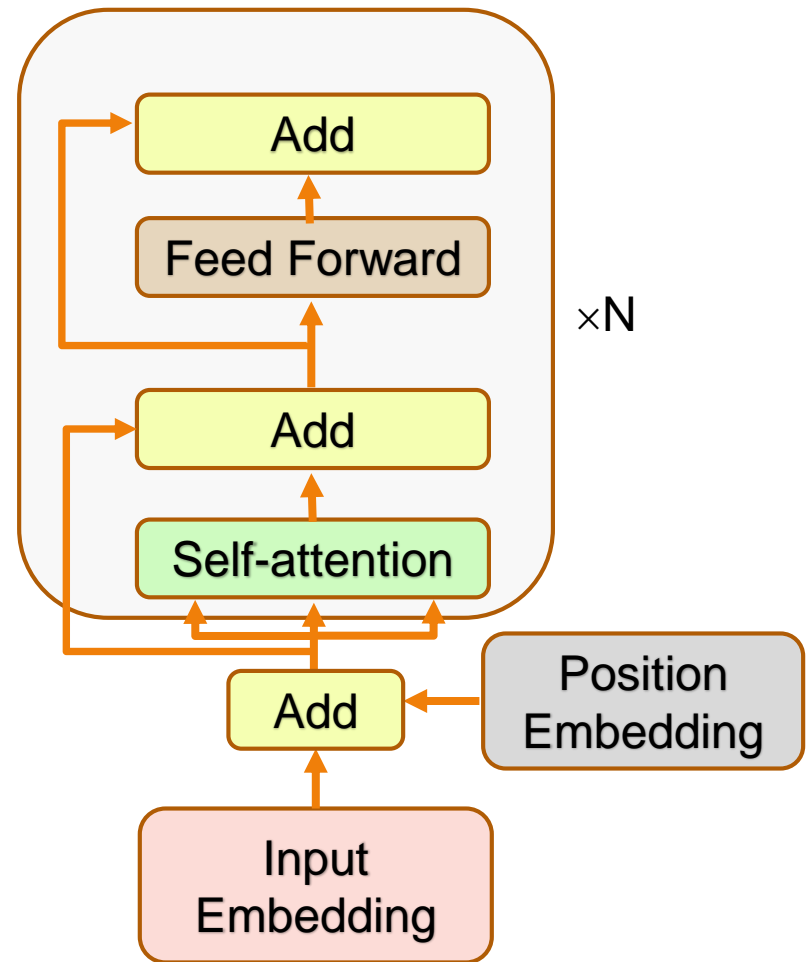


# Training Trick #1: Residual Connections

- ▶ Deep networks are surprisingly bad at learning the identity function!
- ▶ Residual connection: directly passing "raw" embeddings to the next layer

$$x_l = F(x_{l-1}) + x_{l-1}$$

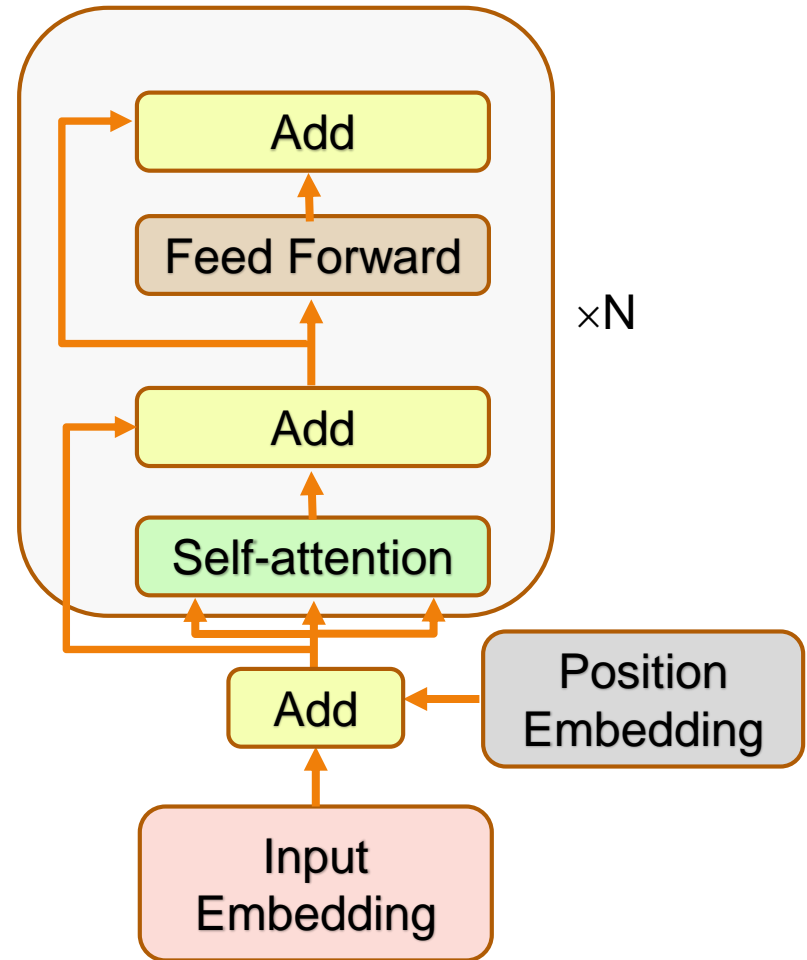
- ▶ This preserves information through many layers
- ▶ It also smoothes the loss landscape and makes training easier



# Training Trick #2: Layer Normalization

---

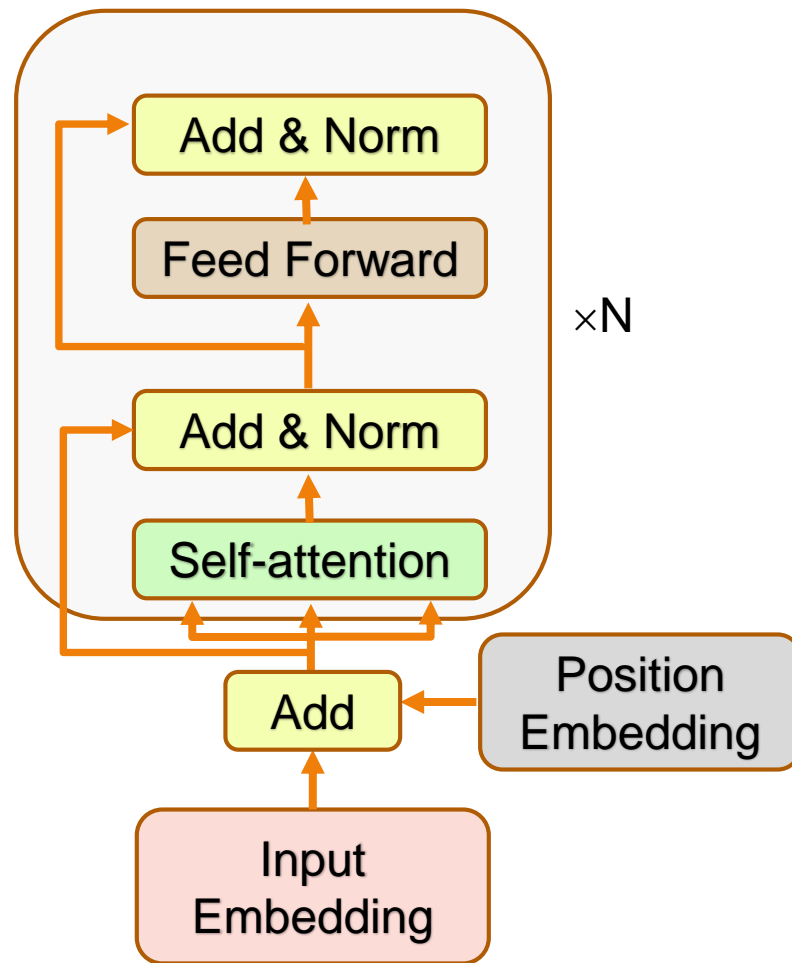
- ▶ Difficult to train parameters of a given layer because its input from the layer beneath keeps shifting.



# Training Trick #2: Layer Normalization

- ▶ Difficult to train parameters of a given layer because its input from the layer beneath keeps shifting.
- ▶ Layer normalization: reduce uninformative variation by normalizing to zero mean and standard deviation of one within each layer

$$\mu^l = \frac{1}{H} \sum_{i=1}^H x_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (x_i^l - \mu^l)^2}$$
$$x^{l'} = \frac{x^l - \mu^l}{\sigma^l + \epsilon}$$



# Transformer for LM

- ▶ In language modeling, we need to ensure we do not peek at the future during training.
  - ▶ Ex. *The boy who is picking apples is his son.* When predicting “*who*”, we should not see “*who is picking...*” and should only see “*The boy*”
- ▶ Masking the future of self-attention
  - ▶ We mask out attention to future words by setting attention scores to  $-\infty$ .
  - ▶ Still fully parallelizable

Attention Mask

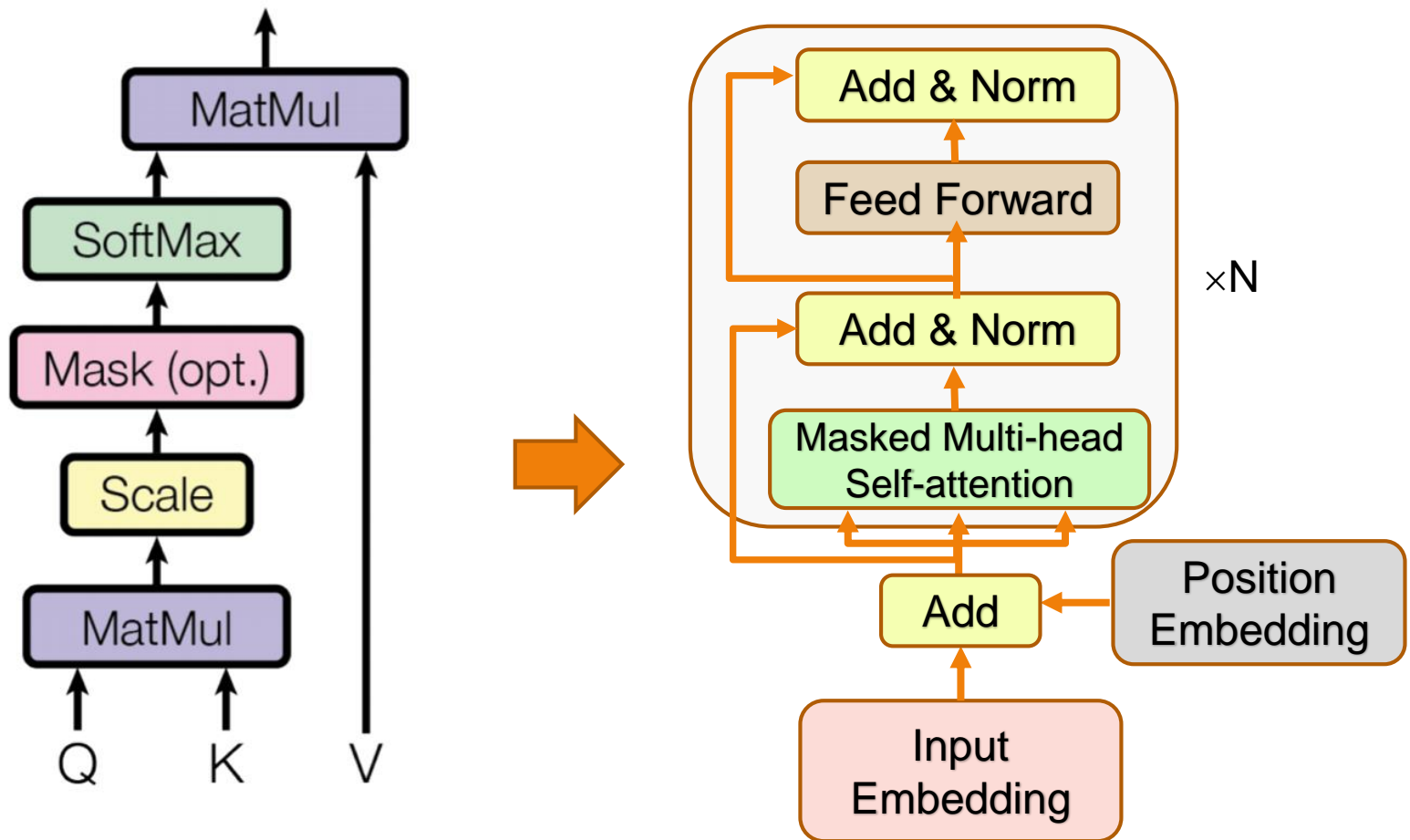
	The	boy	who	is
The	$-\infty$	$-\infty$	$-\infty$	$-\infty$
boy		$-\infty$	$-\infty$	$-\infty$
who			$-\infty$	$-\infty$
is				$-\infty$

$$a_{ij} = \begin{cases} q_i^T k_j, j < i \\ -\infty, j \geq i \end{cases}$$



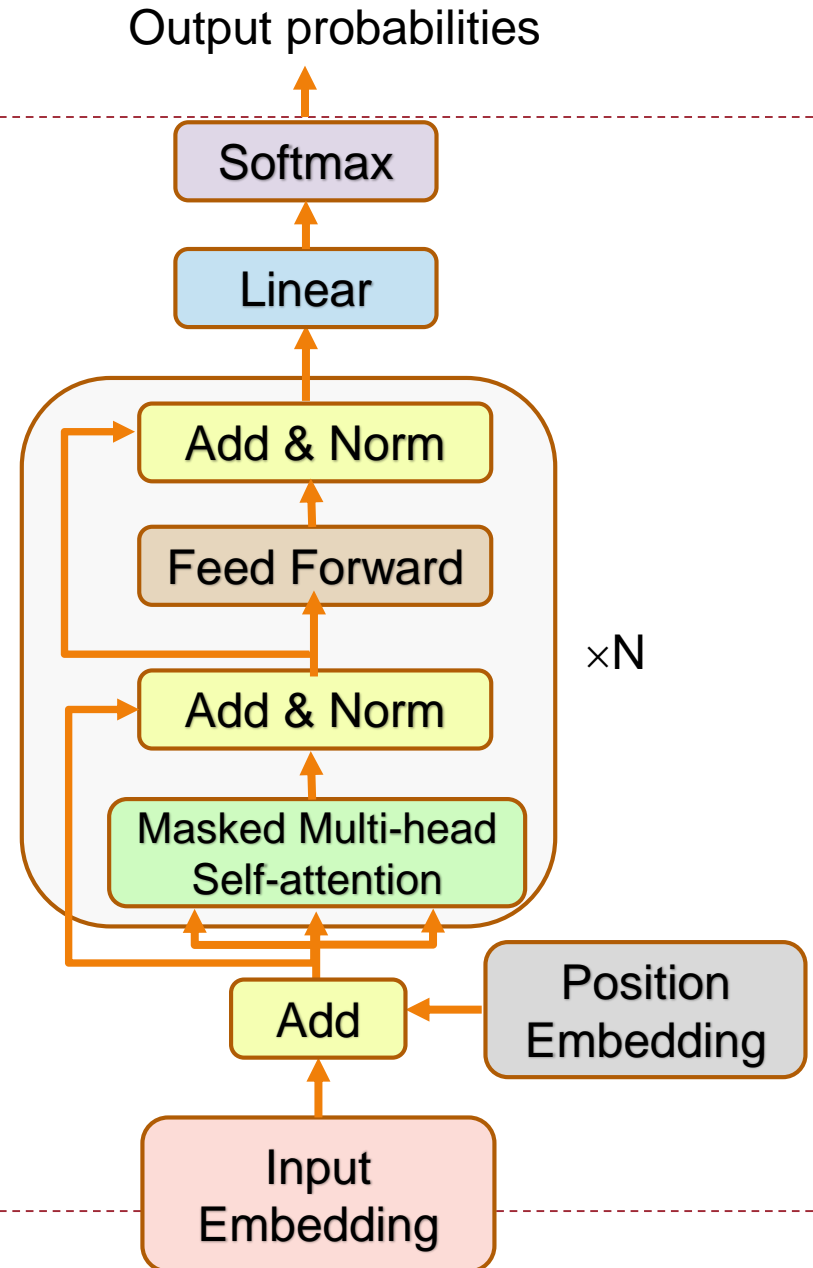
# Transformer for LM

---



# Transformer for LM

- ▶ Add a final layer to predict word probabilities
- ▶ This is the basis of GPT-x



# Generating with GPT-2

---

- ▶ Human prompt

- ▶ In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

- ▶ Model completion

- ▶ The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.  
Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.  
Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.  
Pérez and the others then ventured further into the valley. ...



# Transformer in General



---

- ▶ Originally proposed for machine translation
  - ▶ Including an **encoder** (what we have seen so far) and a **decoder** (to be discussed later)
- ▶ Now widely used for many different tasks
- ▶ The basis of almost all the pretrained language models (PLMs, to be discussed later)



# Extensions and Variants of Transformers

---

- ▶ Quadratic  $\rightarrow$  linear complexity 
- ▶ Relative position embedding 
- ▶ Parameter tying between layers
- ▶ Constrain self-attention with syntactic parse trees
- ▶ Early-exit
- ▶ Residual attention
- ▶ Interaction between attention heads
- ▶ *Many more...*

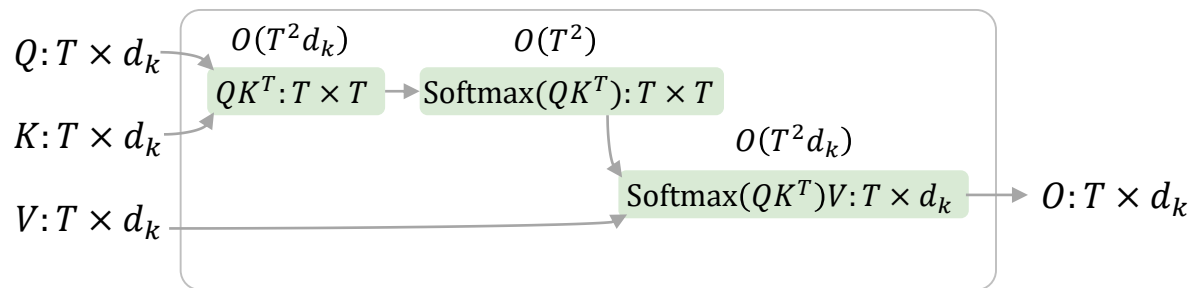


# Complexity of Attention

---

- ▶ Recap: Dot product self-attention

- ▶  $A(Q, K, V) = \text{Softmax}(QK^T)V$



- ▶ Total complexity:  $O(T^2 d_k)$

- ▶ Problem

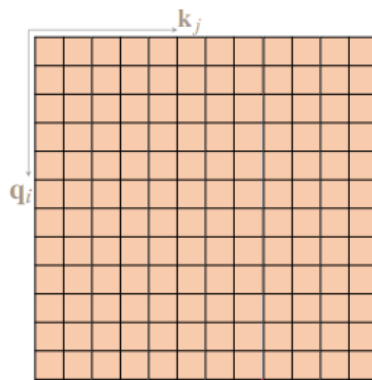
- ▶ Quadratic complexity in sequence length
  - ▶ Unacceptable for long sequences such as documents



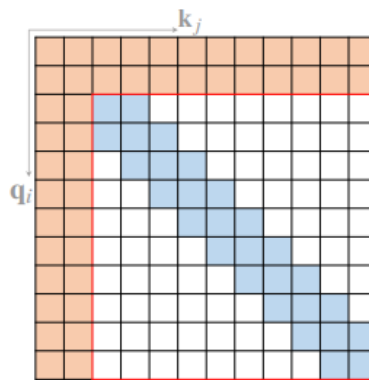
# Solution #1: Sparse Attention

---

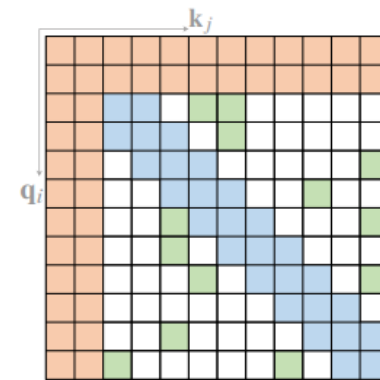
- ▶ Make the attention matrix sparse
  - ▶ Limit the attention matrix according to some pre-defined patterns.
  - ▶ Examples:



Standard attention  
 $O(T^2)$



Global + band attention  
 $O(T)$



BigBird attention  
 $O(T)$

- ▶ Reduce attention complexity from quadratic to linear.



# Solution #2: Linear Attention

---

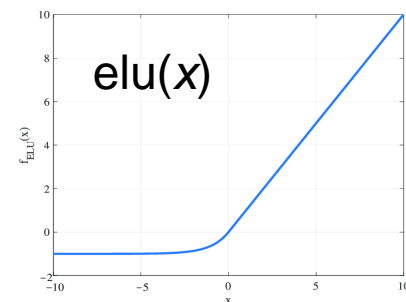
## ▶ What if there is no softmax?

- ▶  $(QK^T)V$ :  $O(T^2 d_k)$ 
  - ▶ Quadratic in length
- ▶  $Q(K^T V)$ :  $O(T d_k^2)$ 
  - ▶ Linear in length

## ▶ Idea: Use $\phi(Q)\phi(K)^T$ to approximate $\text{softmax}(QK^T)$

### ▶ $\phi$ : element/row wise feature map, such as:

- ▶  $\phi(x) = \text{elu}(x) + 1$
  - ▶ Gaussian kernel
  - ▶ Projection to random basis
  - ▶ ...
- ▶ Problem: attention may not be a valid distribution (i.e., sum to one)
- ▶ Another solution:  $\text{softmax}_{\text{row}}(Q) \text{softmax}_{\text{col}}(K)^T$





# Relative Position Embedding

- ▶ Previously, we encode absolute token positions
- ▶ But the meaning of a token is often invariant to its position in the sentence
- ▶ Idea: encode relative positions between tokens when computing attention
  - ▶  $r_{i,j} = \text{clip}(i - j, -k, k)$ ,  $k$  is an upper bound
  - ▶ Add  $r_{i,j}$  into attention formulation
  - ▶ Ex:
    - ▶  $\text{score}_{i,j} = q_i \left( k_j + w_{r_{i,j}}^k \right)^T$
    - ▶  $o_i = \sum \text{attn}_{ij} \left( v_j + w_{r_{i,j}}^o \right)$
  - ▶ Many other designs...

6	$w_{-3}$	$w_{-3}$	$w_{-3}$	$w_{-3}$	$w_{-2}$	$w_{-1}$	$w_0$
5	$w_{-3}$	$w_{-3}$	$w_{-3}$	$w_{-2}$	$w_{-1}$	$w_0$	$w_1$
4	$w_{-3}$	$w_{-3}$	$w_{-2}$	$w_{-1}$	$w_0$	$w_1$	$w_2$
3	$w_{-3}$	$w_{-2}$	$w_{-1}$	$w_0$	$w_1$	$w_2$	$w_3$
2	$w_{-2}$	$w_{-1}$	$w_0$	$w_1$	$w_2$	$w_3$	$w_3$
1	$w_{-1}$	$w_0$	$w_1$	$w_2$	$w_3$	$w_3$	$w_3$
0	$w_0$	$w_1$	$w_2$	$w_3$	$w_3$	$w_3$	$w_3$
	0	1	2	3	4	5	6



# Summary



# Language Modeling

---

- ▶ Compute the probability of a sentence
  - ▶ Chain rule: predicting the next word
  - ▶ Evaluation: perplexity
- ▶ n-gram LM
  - ▶ Probability of each word is conditioned on the preceding  $n-1$  words.
- ▶ Recurrent neural networks
  - ▶ Probability of each word is conditioned on a hidden vector summarizing all the preceding words
- ▶ Transformers
  - ▶ Probability of each word is computed by attending to preceding words

